# Subroutines & Parameters

Prof. John McLeod

ECE3375, Winter 2023

This lesson continues the discussion of assembly language, with specific emphasis to the code base for the ARM®Cortex-A9 processor. Additional forms of branching are introduced, and the stack, and subroutines, and passing/returning parameters are discussed.

As previously mentioned, the basic categories of assembly language instructions are:

1. Data movement instructions.
2. Data manipulation instructions.
3. Conditionals and test instructions.
4. Branch instructions.
5. Subroutine instructions.
6. Interrupt instructions.

This lesson will examine branching in further detail, and introduce subroutines.

## *Branching*

We already discussed the basics of branching: The preferred way to create branches in your program is to use the equivalent of a `GOTO` command to jump to a label somewhere else in your code.

*Mnemonic:* To perform a **direct branch** to a known location in your program, use the `b` mnemonic. This requires a single label as an operand.

*Mnemonic:* To perform an **indirect branch** to a variable location in your program, use the `bx` mnemonic. This requires a single register as an operand, the register should hold the memory address of appropriate instruction.

Like every mnemonic, both `b` and `bx` can have appended conditional codes.

## *The Link Register*

Branching and looping is undeniably cool, but it is a bit rigid. What if we want to write a subroutine that can be called, and then return to the original spot in code? This is where **indirect branching** comes in. Consider the following example:

```
/*some code here*/
mov r3, pc
b pause_for_input
/* more code */
mov r3, pc
b pause_for_input
/* even more code*/
pause_for_input:
   ldr r1, [r4]      @read switches
   cmp r1, r0        @check if switches changed
   beq pause_for_input
   bx r3         @return to original code
```

In this example, `pause_for_input` is now a subroutine.

- By storing the state of `pc` before branching to the subroutine, `r3` now contains the address of the instruction that comes immediately after the branch instruction. [1]
- By exiting the `pause_for_input` block using an indirect branch, the program returns to the appropriate place in the code.

This procedure is so common that a special register and some mnemonics are available to streamline the process, and avoid the

[1] The ARM has the odd property that any time you move `pc` into a register, it actually moves `pc`+4. This always trips me up: when the command `mov r3, pc` executes, `pc` was already updated to point to the *next* line of code, however `r3` actually holds the address of the *line of code after that*.

cludgy and confusing "`mov r3, pc`"-type code.

*Definition:* The **link register** `lr` is a special register used to store the address of an instruction immediately *after* the current one (i.e., the address `pc + 4`).

*Mnemonic:* To save the next instruction before branching to a direct label, use the **direct branch with link** mnemonic `bl`.

*Mnemonic:* To save the next instruction before branching to a indirect location, use the **indirect branch with link** mnemonic `blx`.

This cleans up the above example, as shown below. Note that there is now no need to explicitly save the program counter, or to appropriately increment it.

```
/*some code here*/
bl pause_for_input
/* more code */
bl pause_for_input
/* even more code*/
pause_for_input:
   ldr r1, [r4]      @read switches
   cmp r1, r0        @check if switches changed
   beq pause_for_input
   bx lr         @return to original code
```

## *The Stack*

The above discussion of the link register demonstrate how simple subroutines can be implemented in assembly. However there is still a lot of functionality missing:

- There is only one link register, so subroutines cannot be nested. [2]
- There is a finite set of registers, many of which are probably needed in the main program. But the subroutine needs registers too! Your subroutines have to know which registers it can overwrite and which it must preserve.

A more flexible way of approaching subroutines is to store the **state** of the program prior to the subroutine call to memory, than restore it just before the subroutine returns. This method is common enough that a special area of memory and a special set of mnemonics are available to simplify and standardize writing subroutines in assembly.

*Definition:* The **stack** is a special area of memory used to store "snapshots" of the registers. It is a *last-in-first-out* data structure.

*Definition:* The **stack pointer** `sp` is a special pointer that holds the most recently-used memory address in the stack.

*Definition:* When data is written to the stack, we say it is **pushed** onto the stack.

*Definition:* When data is read from the stack, we say it is **popped** off the stack.

There are a few variations in how a stack is constructed. First, there are two options for how data is written to the stack.

[2] Well, technically these kind of subroutines *can* be nested, if you use a different register to hold the previous `pc+4` address for each level of nesting. But this is clumsy and requires advance knowledge of how many levels of nesting are possible.

*Definition:* A **descending stack** starts at a high memory address, and *decrements* the stack pointer as data is written to the stack. The stack is described as *growing downwards*.

*Definition:* An **ascending stack** starts at a low memory address, and *increments* the stack pointer as data is written to the stack. The stack is described as *growing upwards*.

Second, there are two options for what the stack pointer addresses.

*Definition:* In a **full stack**, the stack pointer addresses the last memory location which contains data that was written to the stack.

*Definition:* In an **empty stack**, the stack pointer addresses the first empty memory location in the direction of the stack's growth.

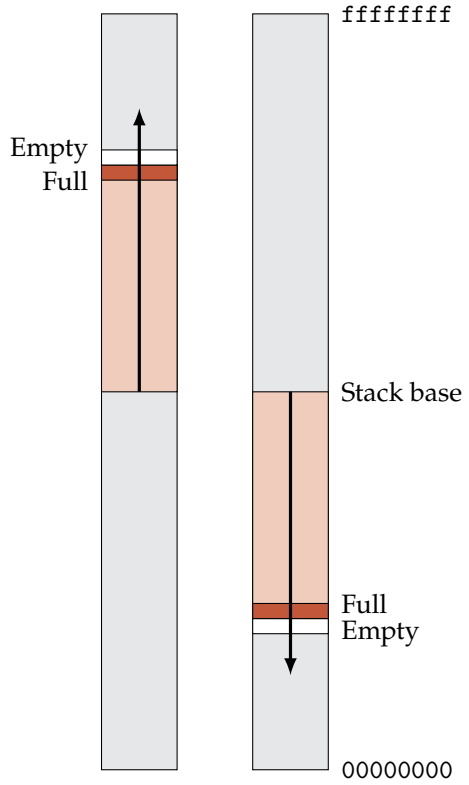


Figure 1: Schematics of different types of stacks. Left: an ascending stack, right: a descending stack. The stack pointer would address the dark red cell for a full stack, or the white cell for an empty stack.

## *Accessing the Stack*

An ARMv7 system uses a **full descending stack**. Similar to other concepts we've discussed (like endian-ness), the architecture of the stack is only important if you try to access it directly. Consider the following examples:

```
str r1, [sp, #-4]! @push to stack
```

The above is the correct way to manually push data onto a full descending stack.

- This uses pre-index addressing to change the value of the stack pointer *before* writing the data, so the data is written to the next empty memory cell.
- The pre-index is $-4$ bytes, since the stack is descending. A pre-index is necessary because a full stack is used (`sp` addresses a full memory element).

```
str r2, [sp], #-4 @push to stack - whoops!
```

The above is an incorrect way to manually push data onto a full descending stack.

- This uses post-index addressing. With a full stack, that will overwrite the last element in the stack before incrementing the stack pointer.
- However, this is the correct way to manually push data onto an empty descending stack.

Please note that this code will still compile and run without error. The CPU doesn't really "know" what is special about a stack — it is just an ordinary place in memory.

```
str r2, [sp, #4]! @push to stack - whoops!
```

Similarly, the above is also an incorrect way to manually push data onto a full descending stack — rather this is for a full ascending stack.

```
ldr r3, [sp], #4  @pop from stack
```

The above is the correct way to manually pop data out of a full, descending stack.

- This uses a post-index of 4 bytes which is appropriate for a full, descending stack.
- Post-indexing is necessary because the stack pointer for a full stack always points to the last data element.

```
ldr r4, [sp, #4]!  @pop from stack - whoops!
```

The above is an incorrect way to manually pop data out of a full, descending stack.

- By pre-indexing a shift of 4 bytes this will actually skip over the last element in the stack, and return the *second-last* element.

## *Don't Re-Invent the Wheel: Use Push and Pop*

You can avoid all the hassle of trying to remember how the stack architecture is defined by using special mnemonics designed to streamline using the stack.

*Mnemonic:* To **push** data onto the stack, use the `push` mnemonic, followed by a list of registers within curly braces.

*Mnemonic:* To **pop** data out the stack, use the `pop` mnemonic, followed by a list of registers within curly braces.

With this in mind, you can push and pop data without worrying about the correct way to update the stack pointer — that will happen automatically. [3] However when pushing/popping multiple registers at once, it is important to recognize the order this occurs.

- If you `push {r0,r1,r2}`, then `r2` is pushed on the stack *first*, and `r0` is pushed to the stack *last*.
- Consequently, the first item removed when you pop data off the stack will be whatever was formerly in `r0`.
- Similarly, if you `pop {r0,r1,r2}`, then the stack is popped into `r0` *first*, and into `r2` *last*. [4]
- This way, `push {r0, r1, r2}` followed by `pop {r0, r1, r2}` does not change the contents of the registers.

As an explicit example:

```
mov r0, #3
mov r1, #15
mov r2, #255
```

[3] *Warning!* In the simulator, you can always `pop` from the stack even if there is nothing in it! This will cause the stack pointer to "wrap around" and start reading the opcodes in the program itself.

[4] It is worth reminding everyone that ARM compiler always sorts the registers such that the lowest valued register is associated with the lowets memory address, so `pop r2, r0, r1` does exactly the same thing as `pop r0, r1, r2`. I don't like it — why use Assembler if you can't do stupid things? — but that's the way it is.

```
push {r0, r1, r2}
pop {r2}
pop {r0, r1}
```

Here the registers are initialized with some arbitrary numbers, then pushed to the stack.

- In an ARMv7 system, the first (word) address in the stack is `0xfffffffc`. As the stack is a full descending stack, *before* anything is on the stack the stack pointer is `0x00000000`.
- When the registers are pushed to the stack, `r2` goes in first, to address `0xfffffffc`. Then `r1` goes to address `0xfffffff8`, and finally `r0` goes to address `0xfffffff4`.
- The contents of the registers is unchanged after pushing them to the stack.
- When the stack is popped to `r2`, the last stack value (the number pushed from `r0`) is moved to `r2`, and the stack pointer is incremented up to `0xfffffff8`.
- When the stack is popped twice, registers `r0` and `r1` are overwritten with the remaining two values.

This process is shown schematically in Figure 2. It is worth stressing, again, that the "stack architecture" is just a *programming convention*, not a *hardware implementation*. If you are coding in assembly language, you don't need to use the stack as defined by the system: You can set aside any arbitrary block of memory and use `str` and `ldr` with the appropriate pre- or post-index addressing to act as a custom stack. Furthermore `sp` can act as a general purpose register,

so as long as you *don't* call `push` or `pop` you can use `sp` as the pointer for your custom stack.

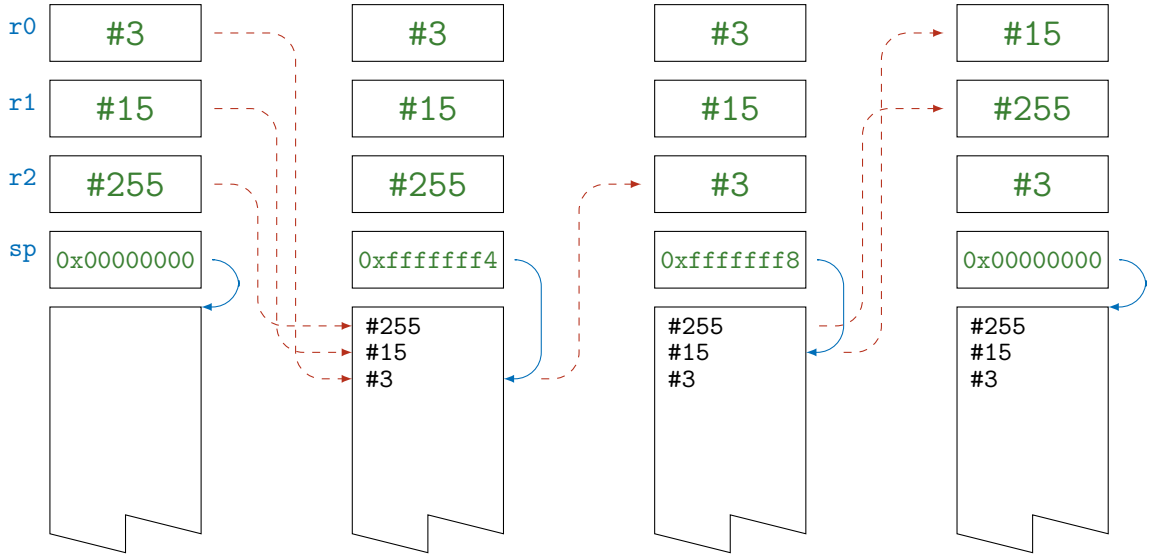


Figure 2: Representation of the code example showing how the stack contents, stack pointer, and the register contents change as data is pushed and popped from the stack. Note that in an ARMv7 system, the first word in the stack is address `0xfffffffc` (the first byte is address `0xffffffff`).

## *Subroutines*

Good, reusable, and portable code uses the concepts of *structured programming*, and often relies on **subroutines**. [5]

*Definition:* A **subroutine** is a block of code that is called from the main code, performs some task, then returns to the same point in the main code. Subroutines often take one or more data values as inputs, and often return one or more data values after completion.

For simple programs, implementing a subroutine is as easy as using a **direct branch with link**, then after completing the subroutine, returning to the main code with a **indirect branch** using the link register, as was discussed above. Simplicity is often a good idea in programming, but this method doesn't always produce a *reusable* or *portable* subroutine. Some issues are:

- The CPU has a limited set of registers. How many registers were holding critical data from the main program, and how many are needed by the subroutine?
- If the subroutine is returning one or more values, which registers or memory addresses will be used for these data?

Note that the first issue is something that is foreign to programmers only familiar with high-level programming: in those languages, the set of possible variables is basically infinite, and subroutines often exist within their own **scope**, so new variables for manipulating data can be defined within the subroutine as necessary. But in assembly, where everything has to go through a limited of registers,

[5] In various languages these are also called **functions**, **methods**, **procedures**, **routines**,... and who knows what else.

data organization can be a important issue.

With this in mind, a "good" subroutine is one that *preserves the main program's status*, and *follows a standard protocol for input and output parameters*.

1. The first is accomplished by the subroutine code immediately **pushing** all relevant registers to the stack as soon as it is called, then **popping** these registers off the stack just before it exists. This ensures the data used by the main program is preserved, while still allowing those registers to be used temporarily by the subroutine.
2. The second is accomplished by just making sure all the subroutines you write follow the same protocol.

For writing subroutines, is *probably* a good idea to follow the same protocol everyone else uses:

- The first four registers (`r0` to `r3`) are used to pass parameters into the subroutine, and return values from the subroutine: when the subroutine is called from the main program any data in those registers can be assumed to be an input parameter, and when the subroutine returns to the main program any any data in those registers is assumed to be an output parameter. The main program's state is *not* preserved with these registers, they are *not* pushed or popped to the stack.
- The remaining registers (`r4` and up, as well as `sp` and `lr`) should be preserved after the subroutine returns to the main program: if any of these registers are needed by the subroutine, the original value should be pushed to the stack, and

restored before the subroutine exits.

This is especially important if you are writing assembly code that may interact with code written using higher-level programming (C code, in particular) — the higher-level code will automatically follow this protocol after it is compiled. Following these standards also makes it easy to have arbitrarily nested subroutines.

- As the labs in this course will use C code compiled into Assembly language, this process is largely transparent to you — the C compiler will handle converting a C function into an Assembly subroutine, with all the necessary **pushes** and **pops** to the stack.
- However it is good to understand this process so you can read the Assembly code generated by compiling your C program.
- And, of course, we like to ask exam questions about arbitrary (and contrived) subroutines.

## *Contrived Example*

Consider the example below. This is a subroutine that does the valuable chore of reading from an analog-to-digital converter (ADC), then adds some constant to it. How valuable is this? So valuable! Such value as cannot be evaluated. There is only one problem:

- I am a bad student, and don't know how to use the ADC yet! So that part is hidden in another subroutine that I will write later.

To implement this subroutine, I will adopt the following protocol. This is consistent with convention, but has some of my own signature flair.

- In my programs, I only use registers `r4` to `r8` for local variables — I don't trust registers higher than `r8` as they are shifty and of poor personal character [6] — so I only need to push those registers, and the link register, to the stack.

[6] Your opinion of these registers may vary.

To ensure everything plays nicely with everything else, I need to remember to preserve the state of the system.

- I am preserving the state of the stack pointer by ensuring that I always pop for every push made during this subroutine.
- I only use `r4` in this example as a local variable, so technically I don't need to push `r5` to `r8` to the stack — this subroutine doesn't touch them, so their value will be preserved — but I want to keep the same standard "state preservation" for all of my subroutines. If I *really* cared about making my code as fast and efficient as possible, I would only push the registers that

are needed by the subroutine.

- This subroutine expects the input parameter to be in `r0`, and the return value is also stored in `r0`.

This subroutine calls two nested subroutines: `read_AD` to read the ADC, and `fix_big_number` that somehow "fixes" when an unsigned addition generates a carry-out. I don't know how either of these two subroutines actually work, but that is not important: as long as they preserve the state and put their output in `r0`, the actual details of their code is irrelevant to this program.

```
/* my super subroutine
   this reads input from an A/D converter
   and adds a constant to it.
   the constant should be in r0
   the output is returned in r0*/
read_AD_add_X:
   @ preserve state
   push {r4, r5, r6, r7, r8, lr}
   @ pass constant to r4
   mov r4, r0
   @ get A/D input
   @ I don't know how to do this!
   @ I will put it in a separate subroutine
   bl read_AD
   @ ok, somehow the A/D input is now in r0
   @ add constant to A/D input, save in r0
   adds r0, r4
   @ is result valid?
   @ a carry-out means hs condition is met
   blhs fix_big_number
   @ restore state
   pop {r4 - r8, lr}
   @ return to main code
   bx lr
```

## *The Stack Frame*

More complicated subroutines may need to be passed, or return, complex data sets as parameters. Furthermore, while that subroutine is executing, it may need extra space for local variables. The stack should be used for all of this. To facilitate using the stack in this manner, a general-purpose register is used as a **frame pointer**.

*Definition:* The **frame pointer** contains the address in the **stack** that is at the start of the local storage for a subroutine.

**Frame pointers** are particularly useful in high-level programming languages — if you write a program in C that involves subroutines, you will see the **frame pointer** being used.

- The C compiler uses register `r11` as the **frame pointer**.
- Register `r11` is a general-purpose register, which can be used for arbitrary purposes in Assembly, so be careful if you are mixing Assembly code and C.

A **frame pointer** is used to set up a well-defined block of memory for *temporary* use by the subroutine.

- This memory can be accessed using the **frame pointer** as the base address, avoiding using the **stack pointer**.
- This is useful because the stack pointer normally should pop data for every push (and vice-versa) in order to always point to the end of the stack, however temporary variables can often be discarded after the subroutine is complete. [7]
- This memory is also useful because it exists at the same location, *relative to the stack pointer at the initial function call*, for

[7] Meaning that there is no need to pop them off the stack, just "rewind" the stack pointer over them.

every subroutine. This allows code external to the subroutine to access that data. [8]

To use stack frames, the protocol for writing subroutines should be as follows:

1. Save the *previous* frame pointer and link register to the stack.
2. Set the frame pointer to the stack pointer.
3. Move the stack pointer ahead by some arbitrary amount to leave some empty space for temporary variables.
4. Now preserve the state by pushing all remaining registers to the stack. Remember the link register was already pushed to the stack in the first step. Also, the stack pointer does not need to be saved as we are using the frame pointer instead.
5. Write the main code for the subroutine. Use addressing with offsets to access the temporary storage space at the start of the stack frame.
6. Restore the state by popping all the general purpose registers.
7. Rewind the stack pointer over the temporary storage block by simply setting the stack pointer to the frame pointer.
8. Pop the *previous* frame pointer and link register from the stack.

The following code provide an example of using frame pointers. Here 32 bytes of memory space are reserved for temporary storage — the exact number doesn't matter, and probably should depend on the needs of the subroutine. [9]

[8] Maybe this last part is kind of a dodgy practice, I dunno...

[9] Also, note the use of `sub` to move the stack pointer "ahead", because the ARM has a **descending** stack.

```
/* my super subroutine using frame pointers */
make_world_peace:
   @ save link register and previous frame
   push {lr, r11}
   @ set up frame pointer
   mov r11, sp
   @ move stack pointer ahead
   sub sp, sp, #32
   @ preserve state as normal
   @ (but lr already saved)
   push {r4 - r8}

   /* main code goes here */
   /* do bunch of stuff */
   @ oh, I want to put something
   @ on the stack
   push {r2}


   /* do more stuff */
   @ save temporary data
   str r3, [r11, #-4]

   /* do more stuff */
   @ read temporary data
   ldr r3, [r11, #-4]
```

```
/* do more stuff */
@ oh, I need that value again
ldr r1, [r11, #-4]

@ let's get the old frame pointer
ldr r0, [r11]
@ let's sneak a peak at the
@ previous subroutine's private data
ldr r2, [r0, #-8]

/* ok, I'm done */
@ gotta pop that old variable though
pop {r2}

@ now restore the state
pop {r4 - r8}
@ rewind stack over temporary storage
mov sp, r11
@ restore old link and frame
pop {lr, r11}
@ exit subroutine
bx lr
```

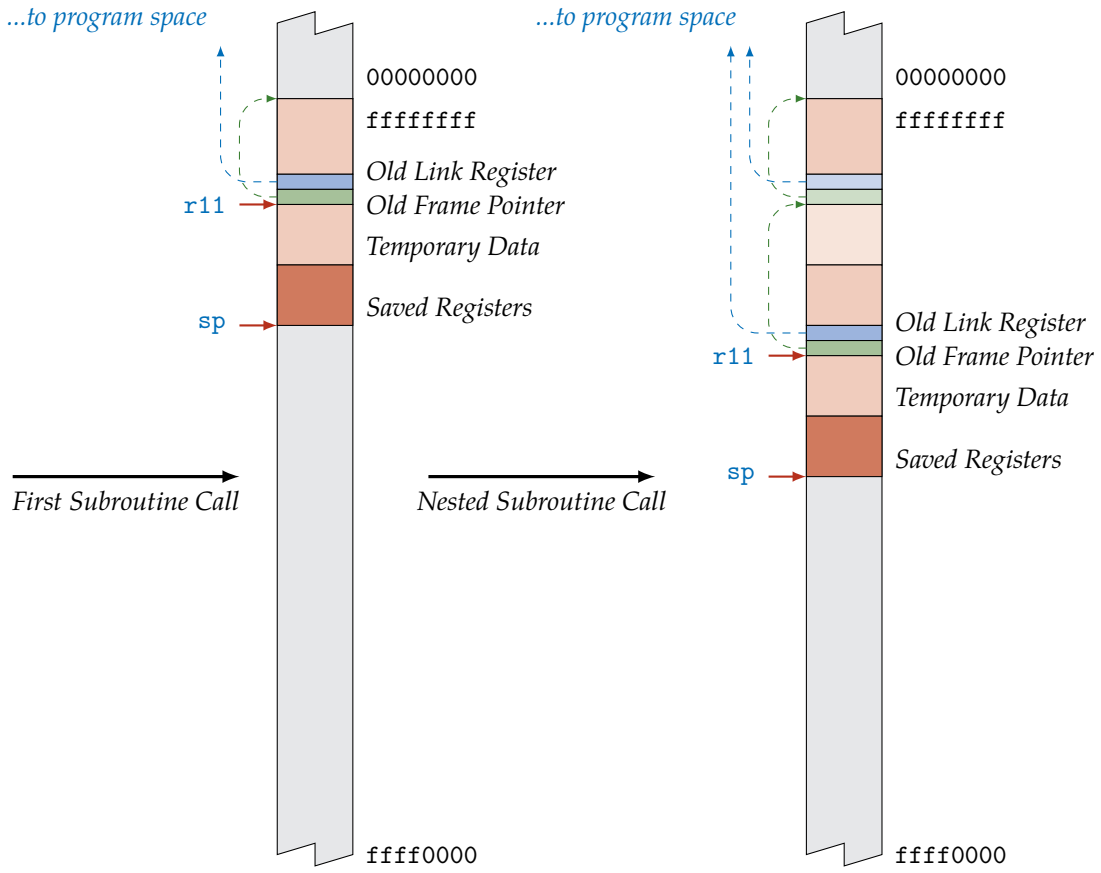


Figure 3: Representation of frame pointers and the stack. Note how the frame pointer effectively creates a **linked list** via subsequent subroutine calls.

As the ARM has a full descending stack, the framepointer `r11` holds the address of the old frame pointer. [10] This *probably* isn't considered good programming practice, but I think it is fun to do dubious things in Assembly language. A visual representation of frame pointers and subroutine calls is shown in Figure 3.

[10] And `[r11,#-4]` is the address of the old link register. Again, the order of the registers doesn't matter because the compiler sorts them, so `pop r11, lr` does the same thing.