



UNIVERSITY
of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 6

ARM Assembly Language

6.1	Aims	69
6.2	A Simple Assembler-Only Project	69
6.3	Assembly Language Tricks	70
6.4	Memory Operations	71
6.4.1	Generating Fibonacci numbers	71
6.4.2	Summing an Array	72
6.5	Call and Return	72
6.5.1	More Complexity	73
6.6	Summary and Conclusions	75

6.1 Aims

In this laboratory session, you will:

- Gain experience with the ARM Thumb-2 instruction set, writing standalone assembly-language code in ARMASM syntax

Copy your project files from your H: drive, or wherever you put them last week, to your local C: drive, perhaps in C:\tmp. Remember that the debugger may not function correctly if you don't do this.

 Ensure that the ARM processor board is connected to the PC via a USB cable.

6.2 A Simple Assembler-Only Project

In this section you will experiment with a simple assembler-only project,⁵ using the debugger to inspect register values. You will not need to use CoolTerm.

 Use **Project→Open Project...** to open `Assembly\start\start.uvprojx`.

⁵Calling this an 'assembler-only' project is a bit of a stretch – there's some C in it behind the scenes as well, to initialise various aspects of the CPU. But all the code you'll write will be in assembly language.

❗ The assembly language code you’re looking at uses “ARMASM syntax”. ARMASM is the name of the original ARM assembler, and the toolchain you’re using is capable of interpreting assembly language files written either in this syntax or in the newer “Unified Assembler Language” (UAL) syntax. Both syntaxes are in common use; the auto-generated startup code for the projects you’re using is in ARMASM syntax, and inline assembly language (which we’ll see another time) requires UAL syntax when using this toolchain.

- Look at the code and try to work out what it will do. What does the loop do? Under what conditions will it end? What will the final value of register `r1` be?
- Build the project and download it to the board.

⚙️ Run the code you’ve downloaded by pressing the black button on the processor board to reset the CPU. What happens?

❗ In order to see the project running, it will be necessary to use the debugger to inspect the contents of registers as the program runs.

⚙️ Make sure the code is downloaded to the board, and then start the debugger. The debugger will start and run to the skeleton `main()` function. Select **Debug**→**Step** or press F11 to step into the assembly language file.

- Keep pressing F11, and watch the Registers window to see the values in the registers change.
- Does the code do what you expected? Does it end when you expected? What happens once it’s ended?

❗ Soon we will look at the interoperation of C and assembly language, and you’ll be able to use functions like `printf()` to display the results of computations you’ve made using assembly language.

6.3 Assembly Language Tricks

Writing efficient assembly language code requires a keen insight into the way a CPU works and the fastest way to perform any given operation.

⚙️ Use **Project**→**Open Project...** to open `Assembly\mult\mult.uvprojx`. This is a ‘blank’ assembly language project, to which you can add your own instructions to try things out.

A good source of examples of efficient assembly-language operations is *constant multiplication*. If you were to write a line in C such as

```
b = a * 5;
```

you might imagine that the compiler would generate a ‘multiply’ instruction – after all, the ARM Cortex-M4F has a hardware multiplier. But this is unlikely to be the case. The

multiplier typically takes a few clock cycles to produce a result, and it consumes a fair bit of power. Instead, the compiler is likely to leave the multiplier switched off and to do something like the following:

```
ADD r1, r0, r0, LSL #2    ; r1 = r0 + 4*r0 = 5*r0
```

To multiply by 10, it might use:

```
ADD r1, r0, r0, LSL #2    ; r1 = r0 + 4*r0 = 5*r0
MOV r1, r1, LSL #1        ; r1 = 2*r1 = 10*r0
```

or:

```
MOV r1, r0, LSL #1        ; r1 = 2*r0
ADD r1, r1, LSL #2        ; r1 = r1 + 4*r1 = 5*r1 = 10*r0
```



Work out a way to multiply a number by 13.

- There are lots of ways to do this, but the best available strategies (of which there are many) take just two instructions. Optimal solutions can be tricky to spot though!⁵
- Implement your idea and test it by using the debugger. Can you think of any *other* ways to multiply by 13 using two instructions?

6.4 Memory Operations

In this section you will use the `LDR` and `STR` instructions to move register contents to and from memory.

6.4.1 Generating Fibonacci numbers



Use Project→Open Project... to open `Assembly\fibonacci\fibonacci.uvprojx`. This project contains a small number of assembler directives, but no instructions.



The `AREA` directive tells the assembler where it can put the following assembled content. The `SPACE` directive tells the assembler to reserve a certain number of bytes of memory and initialise it to zero. So as it is, the assembler will reserve 20 32-bit words; the label `array` points to the start of this area, and the label `array_end` points to the next byte after the end.



Write a program to generate Fibonacci numbers iteratively, and store them in the reserved data block. You will need to:



- Load a register with the address of the `array` label

⁵‘Tricky’ is an understatement – this is known as the ‘single constant multiplication’ (SCM) problem and is NP-complete (it can be proved that there is no efficient way to locate a solution in the general case). If you’re interested in this kind of thing, check out the ‘multiplier block generator’ heuristic algorithm in the [Spiral project](#).

– (Hint: Use the `LDR` pseudo-instruction)

- Load another register with the address of the `array_end` label
- Put the first two Fibonacci numbers (1 and 1) into two registers
- Create a loop to:
 - Store a Fibonacci number into the data block, and increment the data pointer by 4 bytes
 - Compute the next Fibonacci number ready for the next iteration
- Add a test and a conditional branch so that the loop ends when the data pointer becomes greater than or equal to the end-of-array pointer


Test your program using the debugger. Verify that it generates the correct results and doesn't overwrite the end of the array.

-  You will need to use the Memory Window (bottom right) to observe changes in memory. If you launch the debugger and step into your code, you will soon see the array address appear in a register when your `LDR` pseudo-instruction is executed. You can then use this to change the address range that the memory window is displaying, if necessary.
-  Your code almost certainly contains a lot of `MOV` instructions, shuffling numbers around. This could be significantly reduced if you were to compute *two* Fibonacci numbers per iteration. But that would only work if the number of requested Fibonacci numbers was even. Can you think of a way to use this idea to make your code more efficient, but also to work with any arbitrary number of requested Fibonacci numbers?

6.4.2 Summing an Array

 Use Project→Open Project... to open `Assembly\array\array.uvprojx`.

-  The `DCD` directive tells the assembler to reserve one or more words of memory, and to initialise them with the given constants. Effectively, it is a read-only static array declaration.

 Write a program to add up the numbers in the array. The result can stay in a register if you like. Verify that when you run the program, the intended register takes the correct value. Step through and ensure that the loop is not overrunning the end of the array.

Try changing the values in the array, and increasing or reducing the number of values. Check that the program still works as expected.

6.5 Call and Return

In this section you'll try using the `BL` instruction to call functions, and interact with the link register to return.

Your first task is to modify the array sum program so that it adds the *absolute values* of

the elements, and to do this by creating a function that will compute the absolute value of a numeric input.



To do this:

- Add a new label to your code, called `mod`. Remember that labels must be left-aligned. The label can go either before or after the `asm_main` label (this is not C!) but you must ensure that it comes between the `AREA` directive above `asm_main` and the `ALIGN` directive below it.
- Write code in the `mod` function to find the modulus of a number. Remember, you can't pass parameters to a function in assembly language; the "call" is merely a branch, and anything that was in the registers before the branch will still be there afterwards. It's up to you how your `mod` function actually behaves; if you want it to take whatever is in `r7` and form its modulus in-place, or if you want to take whatever is in `r3` and write the modulus to `r2`, or whatever, that's up to you.
- Remember to add the instruction `BX lr` at the end of the `mod` function to return from it.
- Modify `asm_main` to call your new `mod` function for each element in the array before adding it to the sum.

Confirm that your modified program reports a sum of 242 for the example list.

6.5.1 More Complexity

The code you've just written is fine but there are a couple more aspects of complexity to look at before we finish this topic. The first is calling functions across translation unit boundaries (calling functions in other files), and the second is nested calls.

When you call a function using `BL` (or `BLX`), the value in the link register (`lr` or `r14`) is overwritten. So if you write a function that calls *another* function, that nested call will overwrite the value in the link register. In general therefore you need to store the value in `lr` on entry to a function, and restore it at the end.



Use **Project**→**Open Project...** to open `Assembly\recursion\recursion.uvprojx`.



For this exercise, you'll once again be generating Fibonacci numbers — but this time you'll be doing so using a *recursive* algorithm.

The basic idea is this: the number at position n is the sum of the numbers at positions $n-1$ and $n-2$, so a Fibonacci generator function can just call itself twice to obtain the two previous values, sum them, and return the result. If the value of n is 2 or 1, the function can simply return the value 1 instead.



Right-click the "Source Files" group in the project pane and select **Add New Item to Group 'Source Files'**. Choose "ASM File (.s)", name it `fibonacci.s` and choose to put it in the `asm` folder in the current project.

Add the following lines to the file:

```
AREA mainarea, CODE
EXPORT fib


END
```

Note that all of these directives must be indented, only labels can be left-aligned.

In `asm_main.s`, add the line

```
IMPORT fib
```

near the beginning perhaps near the existing `EXPORT` directive.

-  The `EXPORT` and `IMPORT` directives in ARMASM syntax control linker behaviour. The `EXPORT` directive makes a symbol visible to the linker, and the `IMPORT` directive instructs the assembler to accept a symbol and allow the linker to resolve it later, much as a function prototype does in C.


Inside the `asm_main` function, add the lines


```
MOV r0, #7
BL fib
```

before the infinite loop.

 You now need to write the `fib` function. Place it between the `EXPORT` and `END` directives in `fib.s`. Some things to think about:

- The index of the requested Fibonacci number will be in `r0` on entry.
- You'll have to decide how `fib` will return the requested value. Should it overwrite `r0` or use a different register?
- If the requested index is 2 or 1, the function can simply return a value of 1. Otherwise, it should call itself twice, add the results of the two calls, and return that sum.
- The value of the link register `lr` will need to be stored before making any function calls and restored again at the end. Are there any other registers whose values need storing across the call? If a function modifies the value of a register then that register is said to be *clobbered* by the call, so check that the calls don't clobber registers containing values you care about, and if they do then preserve and restore them along with `lr`.
- Remember to return at the end of the function! The `END` directive just marks the end of the file...

-  The answer should be 13, given the input of 7.

-  This is not a simple exercise! The code need not be long; my reference solution is 16 lines, including labels, and could definitely be shorter. But the complete lack of any protections

like type checking, automatic preservation of variables across call boundaries, and the inability to refer to variables by name, might make you realise just what a big deal it was when the early high-level languages emerged bearing features like this.

Calling Conventions



As part of this exercise you had to decide how to pass information into and out of your function, and which registers to preserve across the call. If you want to write functions that interact with those written by other people, or generated by compilers, it's really essential that these things are standardised. This is known as a *calling convention*, and ARM specifies a calling convention for its devices. We'll look at that when we look at interworking between C and assembly language.

6.6 Summary and Conclusions

By now, you should:

- Be a little more familiar with ARM Thumb-2 assembly language and the ARMASM syntax;
- Know how to use it to perform basic programming tasks.

If you have any questions about what you have learned, please ask.

Remember to copy your project files to a network drive before you log off.