UNIVERSITY of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM
Laboratory Scripts

Dr. Andy Pomfret

Laboratory 7

C and Assembly Language

7.1 Aims

In this laboratory session, you will:

- Gain experience of labels, imports, and exports;
- Manipulate `C` data structures from assembly language;
- Undertake programming tasks using a mixture of C and assembly language files;
- Use inline assembly language within C files.

7.2 Exporting Symbols

In this section you will use the `EXPORT` directive to access assembly language from `C`, and practice using the ARM calling conventions.

⚙ Use Project→Open Project... to open `Interwork\export\export.uvprojx`.

Have a look through the project. You'll notice that there are two source file groups. Keeping assembly language and `C` files apart like this helps improve the readability of the project.

The project currently won't link, because there's no code in `asm.s`.[7] Your task is to write the `calculate()` function, in assembly language. The function prototype can be seen at the top of `main.c`.

- You'll need to create a function in assembly language, by using a label and adding an `EXPORT` directive. Remember that instructions and directives must be indented, and labels must not be.
- Your function should return the value of $5x + y$, using whatever instructions you deem necessary.
- Make sure your function obeys the ARM standard calling conventions, as outlined in the video. This includes pushing any scratch registers (`r4`-`r11`) that you use, and remembering to pop them afterwards. Also remember to return from your function!

Implement and test your code.

7.3 Importing Symbols

In this section, in addition to `EXPORT`, you will use the `IMPORT` directive to access `C` functions from assembly language.

Use Project→Open Project... to open `Interwork\import\import.uvprojx`.

As before, this project contains some `C` but no assembly language. There are a couple of things to note:

- The `C` code contains a function called `report()`, which simply prints out its argument. This is because the variable-argument syntax of `printf()` is quite complex when accessed from assembly language.
- The `main()` function calls `fib()`, for which a prototype is given. You'll need to write the `fib()` function.

The `fib()` function will compute Fibonacci numbers. You've already done this, in assembly language, so if you've still got your code from Lab 6 then you can save a bit of time by reusing parts of it for this exercise.

The `fib()` function must:

- Compute n Fibonacci numbers, where n is the value of the argument passed to the function;
- Print out each calculated value by calling the `report()` function.

[7] There are, however, two *assembler directives* which give information to the assembler. The first, `AREA`, gives the assembler information about where to put the code it generates. The second, `END`, is required by this particular assembler to be at the end of any assembly-language file.

⚠ Remember that the ARM ABI requires 8-byte stack alignment across calls in different translation units. You may assume that the stack is 8-byte aligned on entry to your function, but you will need to ensure that it remains so when calling other functions. The easiest way to do this is always to push and pop even numbers of registers.

⚙ Implement the `fib()` function, entirely in assembly language. Build and test your code.

7.4 Working with Structures

As you have seen in the video, when you're working with `C` structures and assembly language you have to be very careful to understand the layout of the structures.

⚙ Use Project→Open Project... to open `Interwork\structs\structs.uvprojx`.

You'll see that there's a structure declared in the `C` file. It contains two signed integers, a void pointer, and a pointer to a function. No other code is defined at the moment.

The following task is a little abstract, but is designed to help you to get to grips with structure manipulation in assembly language and of callbacks and void pointers.

7.4.1 First task: Make it work

⚙ To begin with, write some `C`.

- Create a prototype for a function, `void process(structure_t * s).` You will write this function in assembly language soon.
- Create a function that takes a single argument of type `void *` and returns nothing. Call it '`printInteger()`' or something similar.
- Inside this function:
  - Assign the `void *` argument to a local `int *` variable.
  - Add a call to `printf()` to print out the value of the integer.
- In the `main()` function:
  - Declare a variable of type `structure_t`.
  - Set its `x` and `y` fields to any numbers you like.
  - Declare a variable of type `int32_t`. Give it any value you like.
  - Take the address of this new variable and use it to initialise the `ptr` field in the structure.
  - Set the function pointer `callback` in the structure to point to `printInteger ().`
  - Call the `process()` function, passing it the address of your `structure_t` variable.
  - Print out the values of `x` and `y`.

Now, write some assembly language.

- Create a function labelled `process`.
- Inside this function:
  - The function argument (in `r0`) is a pointer to a structure. Use `LDR` to load the fields of the structure into four scratch registers.
  - Add together the `x` and `y` fields, and write the result back to `y`. You'll need to use `STR` to update the contents of the structure.
  - Use `BLX` to call the `callback` function using its pointer. Pass it the `ptr` field as an argument.
  - Return from the function.
- Export the symbol `process`.

⚠ Remember to push any registers you use – including the link register – to the stack at the start of your function, and to pop them at the end, and to maintain 8-byte stack alignment. Remember also to return from your function.

⚙ Implement and test your code. Does it behave as you expect? You should see the value of the integer whose address you placed in the `ptr` field being printed out by the `printInteger()` function, and then the values of `x` and `y` as modified by your assembly-language function. *Take some time to understand this program if you're not entirely sure of what it's doing and why.*

7.4.2 Second task: A different callback

⚙ Now try creating a different callback.

- Create another function that takes a single argument of type `void *` and returns nothing. Call it '`printDouble()`' or something similar.
- Inside this function:
  - Assign the `void *` argument to a local `double *` variable.
  - Add a call to `printf()` to print out the value of the double. (The floating-point format specifier is '`%f`', in case you'd forgotten.)
- In the `main()` function:
  - Declare another variable of type `structure_t`.
  - Set its `x` and `y` fields to any numbers you like.
  - Declare a variable of type `double`. Give it any value you like.
  - Take the address of this new variable and use it to initialise the `ptr` field in the structure.
  - Set the function pointer `callback` in the structure to point to `printDouble()`.

  - Call the `process()` function, passing it the address of your new `structure_t` variable.
  - Print out the values of `x` and `y`.

Implement and test your code. Does it behave as you expect? You should now see the value of the double whose address you placed in the `ptr` field of the second structure being printed out by the `printDouble()` function, and then the values of `x` and `y` as modified by your assembly-language function.

7.4.3 Third task: Breaking the code

Try swapping the order of some elements in the structure.

- Try swapping `x` and `y`. What happens? Why?
- Try moving the elements around a little more radically, changing the positions of `ptr`, or `callback`, or both. What happens?
- Fix your code by modifying the assembly language to cope with the new element order.

7.5 Inline Assembly

Keeping your assembly language in separate files from your C code has some advantages:

- Maintenance is simpler, because all assembly language is kept separately.
- The syntax is simpler — inline assembly language has to interact with the compiler, so there are a lot more things to consider, as you'll see.
- Portability is improved, because although assembly language is always non-portable, keeping it separate reduces the scope of the task if porting is required.

However, there is one big disadvantage to separate assembly language files: using the functions you've written in assembly language will *always* require a function call and return, and associated stack pushes and pops. This will take time, and could easily negate any speed advantage you were hoping to see through the use of assembly language, or add unnecessary overhead to simple operations that require just a statement or two of assembly language to complete.

The alternative is to use *inline* assembly language.

Note that inline assembly language is *not* part of the C language specification. Any time you use inline assembly language you are relying on compiler-specific language extensions. Although given the lack of portability of the assembly language itself, this may not be considered a huge problem. We will be considering the GNU inline assembler syntax in this module, which is a syntax shared by Clang and the ARM Compiler 6 toolchain.

GNU Inline Assembly

The following description is important, but parts of it may not make sense until you have seen some examples.

The GNU inline assembly syntax consists of the `__asm` statement, which consists of four operands separated by colons:

```
__asm(code : output operand list : input operand list : clobber list);
```

The operands are:

- **code** — a single string containing the assembly language instructions. This may contain references to items in the other operands;
- **output operand list** — a comma-separated list of output operands from the assembly language code, including the names by which they are referred in the code, and their corresponding names in the C code;
- **input operand list** — a comma-separated list of input operands to from the assembly language code, including the names by which they are referred in the code, and their corresponding names in the C code;
- **clobber list** — a comma-separated list of registers whose values are "clobbered" by the assembly language code, and which are not specified as part of any output operands.

Some notes:

- Only the **code** operand is mandatory.
- The code must be supplied as a single string. If multiple instructions are required, these must be separated by newline characters (`\n`) and must be indented with tabs (`\t`). Note that if multiple string literals are provided one after another in C, separated only by whitespace or newlines (or comments), these will be concatenated by the compiler.
- Spaces, newlines and C comments can be used within the `__asm` statement to aid readability.
- The input and output operand lists share a syntax: they consist of a *symbolic name* in square brackets, by which the operand is referred in the assembly language code; a *constraint string*, which we will look at soon; and a *c expression* in parentheses.

⚠ The C compiler will attempt to optimise your code, *including* any assembly language code you have written, if optimisation is enabled. This is often not what you want, and it can be useful to tell the compiler to leave your assembly language alone when optimising.

For this, the volatile keyword can be used following the __asm keyword.

Ok. Time for an example and an exercise to make this mean something.

Use Project→Open Project... to open Interwork\inline\inline.uvprojx.

Have a look in main.c. You'll see that the code is quite short, but if you run it you should find that it multiplies a number by 13. Let's look at how the __asm statement works in this case.

The inline assembly language block is

```
__asm volatile (
  "ADD %[output], %[input], %[input], LSL #3\n\t" /* x*9 */
  "ADD %[output], %[output], %[input], LSL #2" /* +(x*4) */
  : [output] "=&r" (y)
  : [input] "r" (x)
);
```

Taking its elements in turn:

- The code consists of two instructions, specified as a single split string literal, and separated by newline and tab characters. It doesn't contain references to specific registers (though inline assembly code *can*, if necessary). Instead it contains the placeholders %[output] and %[input], which refer to items in the output and input operand list, respectively.
- The output operand list defines a single symbol called output, which will be bound to the symbol called y in the surrounding C code. The constraint string "=&r" indicates that this is an output from the assembly language code, that it will be held in a register, and that its value may be written before the input operands have been read for the last time (so the compiler can not reuse one of the input registers for this output).
- The input operand list similarly defines a symbol called input which will be bound to the symbol called x in the surrounging C code. In this case the constraint string defines it just to be a register.

So the compiler will ensure that x and y are in registers, and will substitute the correct register numbers into the assembly language code. The result is that when the code runs, the value in x is multiplied by 13 and placed in y.

Open the code in the debugger. Identify the lines in the disassembled code that are due to the inline assembly language. Which registers are being used for x and y?

Constraint Strings and the Clobber List

There are many options for constraint strings — see the documentation here — but in general you will always want to use the character `r`, indicating a register.

For output operands, prefixing this character are zero or more *constraint modifiers*. These could be:

- `=` for an operand that is only written to, and is written only after all input operands have been read for the last time. The compiler may allocate this operand to the same register as an input operand during optimisation.
- `+` for an operand that is written to and read from.
- `=&` for an operand that is only written to, but may be written before all input operands have been read for the last time.
- `+&` for an operand that is written to and read from, but is written before all input operands have been read. Known as an "early-clobber operand".

The clobber list usually takes register numbers, in double quotes, separated by commas. There are two other clobber specifiers that can be useful though:

- `cc` indicates that the CPU flags are clobbered by the code (e.g. the zero or carry flags are affected);
- `memory` indicates that memory is written by the code, and that the compiler should not rely on memory having the same state before and after the code execution.

7.5.1 Modifying the example

Modify the code to remove the variable `z`, and to *add* 13 times `x` to whatever is already in `y` and print the resulting value of `y`.

To do this you will need to modify the `output` symbol so that it can be read from as well as written to. Leave it as an output argument, but change its constraint string to `"+&r"`.

You may also find that you need to use an additional register for an intermediate result. It is best practice *not* to hard-code a register for this and include it in the clobber list, but instead to create a C variable for it and pass it in as another output argument using the `"=&r"` constraint string.

Test your code, both by running it and by examining the disassembly listing to check that the generated code is sane.

Try replacing your intermediate variable with a hard-coded register, and adding that register to the clobber list. How does that affect the disassembly listing?

7.6 Summary and Conclusions

By now, you should:

- Be more familiar with `IMPORT`, `EXPORT` and using linker symbols in assembly language;
- Have seen first-hand how important it is to control the order of fields in a structure;
- Know more about the GCC/Clang inline assembly syntax for ARMv7 devices

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