Questions from Assignment Manual

Question 1: How to access the elements in the array from an assembly function? Given the n-th element's address in x[N] is ADDR, what would be the address of the (n+k)-th element in x[N]? Given that n and k are two constant decimal integers, and ADDR is a constant hexadecimal integer.

- To access elements in the array from an assembly function, we need to obtain the address of the first element of the array (the pointer of the array).
- By offsetting the pointer by **(index * #4)** (which is **sizeof(int)**), we can load that address into a register to be used.
- Address of the (n+k)-th element is **[ADDR + (#4 * k)]**
- An example assembly code is shown on the right.

```
mov r1, #4
ldr r2, =k
mul r2, r2, r1
ldr r1, =ADDR
add r1, r2
ldr r0, [r1]
.equ k, ...
.lcomm ADDR 32
```

Question 2: Describe what happens with and without PUSH {R14} and POP {R14}.

- Without **PUSH** and **POP** of the **LR** (**R14**), after executing the **BL** instruction, the **LR** will be overwritten.
- In asm_fun(), this LR would point to the line after "**BL SUBROUTINE**". Thus, if there is no push and pop, **bx lr** will constantly call itself, causing an infinite loop.

Question 3: What can you do if you have used up all the general purpose registers in your assembly function and you need to store some more values during processing?

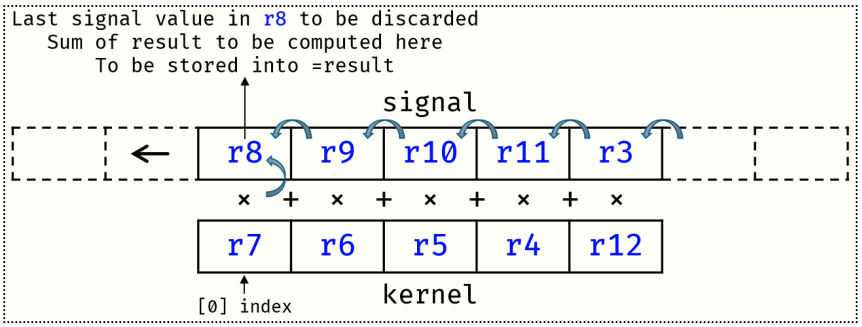
- Method 1: Push all current GP registers to stack. After utilising the registers, we can pop back all GP registers from stack.
  - Method 1.5: Instead of using the stack, reserve memory with **.lcomm** and store GP registers in those addresses, so that the registers are freed up to load and process other values.
- Method 2: Instead of loading all the required memory in the registers and processing it all at once, perform each operation individually, and continuously access the memory every time the operation requires a new value (**ldr**, **str**).

Program Overview

We designed 3 different implementations of this convolution program, which will be selected and run depending on the kernel size in order to optimise for **speed** efficiency.

- **convolve_kernel5.s**: If **len(kernel) ⩽ 5**, we are able to load all kernel values and the 5 relevant signal values in the registers all at once.
  - This allows us to load kernel values and each signal value only once, thus decreasing memory access operations (total load operations **= M+N**).
  - Number of loops will also be minimised to **lenY = M+N-1**
- **convolve_kernel16.s**: If **5<len(kernel) ⩽ 16**, we are also able to load all kernel values and the 16 relevant signal values in the registers by splitting each 32-bit register into 4 separate bytes.
  - Similar to above, this decreases memory access operations and loop count since each kernel and signal value will only be loaded once.
  - However, this implementation will only run when **len(kernel)>5** since it requires a lot of shifting operations, which makes it less efficient than the above implementation
- **convolve_generic.s**: If **len(kernel)>16**, to work with large array sizes, we need to access and process each kernel and signal element one by one, through a nested **for** loop.

convolve_kernel5.s Implementation



The concept of this implementation was to decrease load / store operations through loading and keeping all the kernel values in registers **r4-r7** and **r12**.

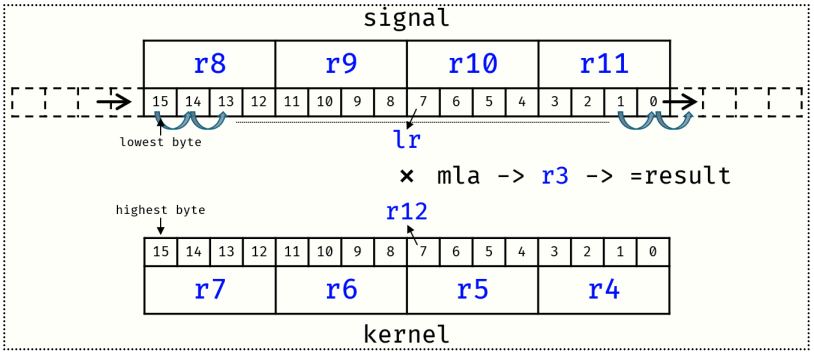
Since kernel values do not shift throughout, we can load new signal values into **r3** and shift existing values from **r3 > r11 ... > r8**, constantly multiplying the same registers together (e.g. **r7** with **r8**, **r6** with **r9**, etc.). This simplifies and shortens our loop. In total, this allows us to restrict the number of load operations to **len(kernel)+len(signal)**, and the number of store operations to just **len(result)**.

Register Management: In order to save on the number of registers and allow us to maximise the number of kernel values we could store, we aggressively reused registers whenever they became available. For example, in the diagram above, since **r8** (the last signal value) would have been discarded after **mul** with **r7**, we reused it to act as the accumulator (**mla Rd**) that stores the current result being computed.

Hardcoding: Towards the end of this implementation when the last signal value is loaded, we also broke out to a "**hardcode**" branch rather than continue the loop described above. Essentially, instead of shifting the remaining values down the registers, we re-targeted each kernel register to be processed with a different signal register (e.g. **r7** would multiply with **r9** instead of **r8**).

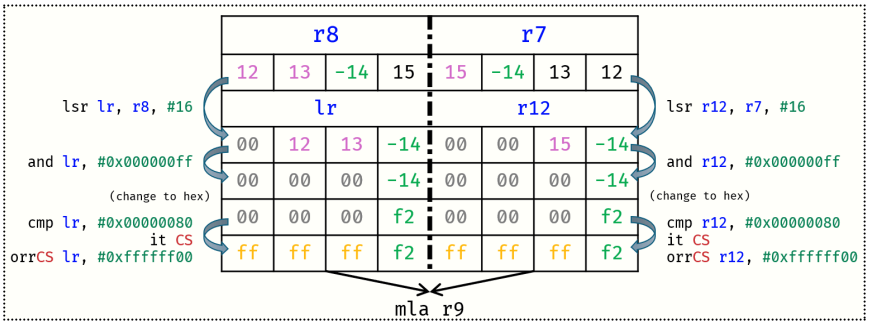
- While this increases code size and makes it slightly unreadable, a kernel size of 5 was small enough for us to "hardcode" this part, and allows us to save on unnecessary **mov** operations.

convolve_kernel16.s Implementation



This program flow is similar to the previous one, fixing kernel values while moving signal values in the registers. However, to store more values in the registers, we split each 4-byte register into individual bytes, hence multiplying the available storage by 4. We assumed that each input value is within the reasonable range of -128 to 127, which is the maximum size that 1 byte is able to store.

Managing Signed Numbers: Unlike unsigned or positive integers, the two's complement of signed 8-bit negative integers are different from that of signed 32-bit negative integers.



When we **lsr** and isolate the required byte by masking all except the last byte (through **AND #0x000000FF**), we need to ensure that the sign of the 8-bit number is preserved.

We do this by checking if the resultant 32-bit value **>#0×80** (128 in decimal, or -127 in signed 8-bit integer). If this is the case, we fill the front 3 bytes with **#0×FF**, thus flipping the sign for the 32-bit integer.

Register Usage: The concept of this implementation was to maximise the number of kernel values available to be stored. Thus, it was a conscious effort to use and reuse every register possible.

Register	Purpose
r0	1st Address of Signal -> Current Result Address (after post increment offset)
r1	1st Address of Kernel -> Travelling Signal Start Address
r2	**len(signal)** -> Last Signal Address (to know when to stop loading signal)
r3	**len(kernel)** -> Last Kernel Address -> Accumulator result (**mla Rd**)
r4 - r7	Loaded Kernel Values. Each register contains 4 kernel values.
r8 - r11	Current Signal Values. Each register contains 4 signal values.
r12	Kernel Value currently being processed.
lr	Signal Value currently being processed.

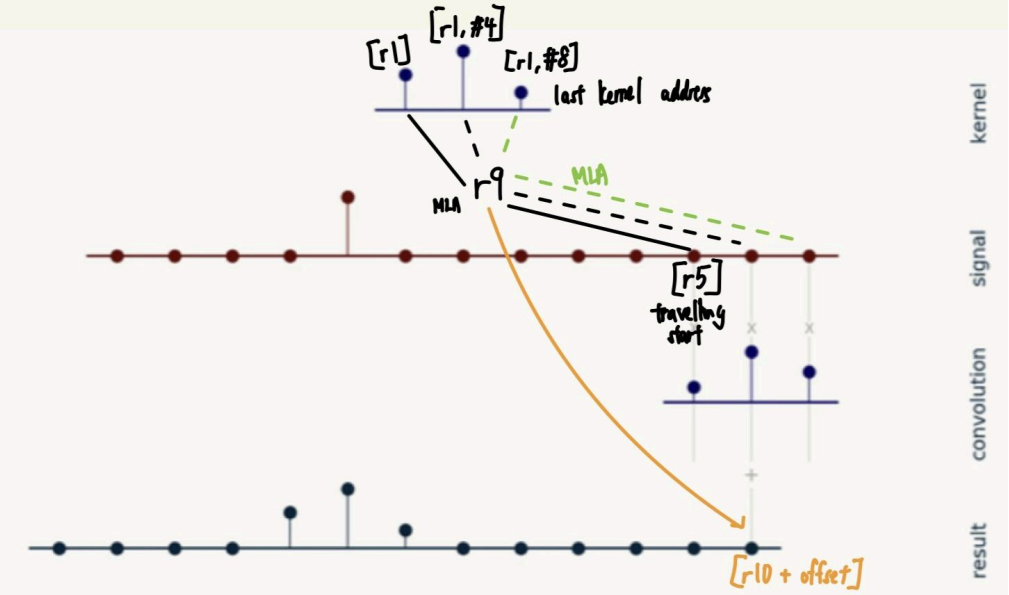
Usage of **r12** and **lr**: Compared to the previous **convolve_kernel5.s** implementation, we used separate registers for our accumulator result and currently-processed signal and kernel values. This is because values were being stored as single bytes within 4-byte registers, and inline barrel shifters were not available for the **mul** or **mla** instructions to manipulate the existing **r4-r11** inline without another register.

Usage of **lr**: As shown above, each register is constantly being used, and every result cannot be discarded. If we did not use **lr** as a general-purpose register, we would either have had to **push**/**pop**, **ldr**/**str**, or decrease the number of kernel values to 12, all of which were not ideal since they defeated the speed efficiency and purpose of this implementation.

- Since we had to use **lr** as a register to store values, and we did not have registers to use as a loop counter, we were unable to branch to any function inside our main signal processing loop. This resulted in bad readability and many repetitive functions. However, we chose to make these trade-offs so that our speed efficiency can be maximised.

convolve_generic.s Implementation

If there are more than 16 kernel values, our last implementation would have to be generic and adaptable enough to kernels of any size. In this case, there would definitely be insufficient space within existing general-purpose registers to store the kernel values. Hence, we decided to perform each multiplication operation individually, loading kernel and signal values one at a time. This is similar to the C code provided, and resembles a nested for-loop.



Comparison to optimised C code: Though the number of load operations would be significantly greater than our previous two implementations, we still made sure to minimise these expensive memory access instructions as much as possible. Our resultant code achieved roughly the same efficiency when compared to the given C code (after GCC compilation to Assembly, with flags **`-mcpu=cortex-m4 -g0 -O3`**).

- Both only utilised 2 **`ldr`** (1 for kernel and 1 for signal) every inner for-loop.
- Both only utilised 1 **`str`** instruction (for the output value) every outer for-loop.

Inner for-loop in GCC	Nested for-loop in our implemented code

`.L4:`
`ldr r6, [r2, #4]!`
`ldr r3, [r4, #-4]!`
`cmp ip, r2`
`mla r5, r6, r3, r5`
`bne .L4`
`str r5, [r1]`
	`large_loop:`
	`...`
	`small_loop:`
	`cmp r4, r2`
	`bGT exit_small`
	`ldr r7, [r4], #4`
	`ldr r8, [r6], #-4`
	`mla r9, r8, r7, r9`
	`cmp r6, r1`
	`bGE small_loop`
	`exit_small: str r9, [r10], #4`
	`add r5, #4`
	`b large_loop`

Future Improvements

We might be able to combine **`convolve_kernel16.s`** and **`convolve_generic.s`**, through loading >16 kernel values only once into the registers, then loading the signal values one by one to be multiplied with the fixed kernel values. This essentially halves the number of load operations compared to the current **`convolve_generic.s`** implementation.

Appendix

Both of us contributed equally to our project :)

While the conceptualisation of all implementations were done together, Cedric worked on the `convolve_generic.s` implementation, while Jing Heng worked on the `convolve_kernel5.s` and `convolve_kernel16.s` implementations. Both of us were involved in cross-checking and debugging each other's code.