**SHOT Assignment 2 - What’s the ARM in it?**

**ARM Architecture – The Rat of the Processor World**

**Questions (13 questions in total) SAMPLE ANSWERS**

1. The Cortex M4 is a **RISC** processor – what in ARM’s history lead to this design?  
   **(5% marks)**

Started with small embedded systems’ controllers, with simple ISAs. This was designed to be low power, low cost, and with small binaries (memory foorprint) which implies a RISC approach. This continues to this day. RISC instructions consume less power as the pipelines are simpler and can be shorter, and a complexity of CISC wasn’t required for much of the targetted market (such as the early BBC Micro).

Here’s the Wikipedia entry, to check for plagiarism, but it does sum it up:

“ARM, previously Advanced RISC Machine, originally Acorn RISC Machine, is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments.

Processors that have a RISC architecture typically require fewer transistors than those with a complex instruction set computing (CISC) architecture (such as the x86 processors found in most personal computers), which improves cost, power consumption, and heat dissipation. These characteristics are desirable for light, portable, battery-powered devices—including smartphones, laptops and tablet computers, and other embedded systems.[3][4][5] For supercomputers, which consume large amounts of electricity, ARM could also be a power-efficient solution.[6].”

Another frequently cited source is ARM’s own:

<https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1>

1. What does the ARM instruction subset called **Thumb2** bring to the microcontroller party? What are its key features? **(5% marks)**

Thumb ISA 16-bit subset of ARM instruction set allows denser code, so can use smaller memories. Helps with I-cache where used. All instructions are 2 bytes long, simplifying the pipelines. Narrow memory buses (16 bit) can be more easily utilised by Thumb code. Limited functionality (implicit operands and smaller register bank). Thumb2 extended functionality to 32-bit-like capability. Each Thumb instruction is decompressed to an equivalent ARM instruction, so doesn’t benefit code speed.

1. What **endianness** does ARM employ in the Cortex M4? Should we be thanking Von Neumann or Harvard for the memory organisation on the M4, and why?  
   **(5% marks)**

Either, but the SETEND instruction can swap endianness. Many chips are specified and manufactured with one or the other. We see big-endian in the debugger. The M4 has a Harvard architecture with separate buses and regions of memory for code and data.

1. The ARM architecture includes an **LR** register. What benefit does this bring to the performance of code?  
   **(5% marks)**

**LR** is the **Link Register** and is used to store the return address when a function is called (rather than the stack). Calling is achieved with a Branch instruction ‘BL’ or ‘BLX’ – *branch with link* which copies the return address to LR before jumping to a label. This LR value may have to be stacked (or saved in a spare register- better!) by the called routine if it’s not a leaf function, and it needs to call something in turn, but for leaf functions that are called frequently, it gives a performance boost as returning is much quicker (simply move the LR to the PC).

1. (a) Why are **FPUs** and their cousin technology, **NEON**, frequently optional in ARM architectures?  
   **(5% marks)**

Not always needed by applications, they cost $$$, and take power and space on the chip. Often simpler to re-factor code to use integer math.

(b) Executing the fast inverse square root code (see below) on the Cortex M4 with the **FPU turned off** still gives the correct answer of 629.9937 when run in a loop adding up 100,000 'x' values – how does it manage that? **(5% marks)**

Uses software emulation (softFloat) – with commensurate performance implications. For example, to multiply it will invoke the function,  **float \_\_mulsf3 (float a, float b)** or similar. Some of you expliained the Fast Inverse magic number code – that’s irrelevant – how are the FP calculations done without a FPU was the question.

1. The highlighted line from the fast inverse square root algorithm above is implemented in the Cortex M4 with the following single line of assembly:

**SUB r0,r1,r0,ASR #1**

Assuming that **r1** contains the magic number **0x5f3759df** and **r0** contains the value of ‘**i**’ explain what the **ASR** component is doing, and thus explain the general syntax of the operand list. **(10% marks)**

Flexible second operand (Operand2’) is being employed. ASR is **arithmetic shift right** by #1 bit – this changes the value of the second register operand before the SUB takes place. So the instruction reads r0 = r1 – (r0 shifted 1 place to the right). The flexible second operand gives much flexibility with certain instructions.Other shifts are available to provide fundamental operations on a register contents before being applied, such as integer multiplying and dividing by powers of 2.

1. Consider the following C code fragment containing some inline assembly written in ARM code for the Cortex M4 processor; it includes a conditional execution (**IT***, “if-then”*) block.

(a) What role do IT blocks play in the optimisation of code execution on a Cortex M4 processor?  
**(10% marks)**

The M4 doesn’t have branch prediction hardware because it’s a low-power, low-cost processor - using IT blocks mitigates the problem of mispredicted branches (only static prediction is operating). Conditional execution reduces the need for branch prediction and pipeline flushes, both of which increase complexity and can significantly impact (reduce) performance.

Predicated instructions in an IT block are fetched as normal, but quickly flushed if they don’t match the current condition, thus the overhead is low. However, they are limited to up to 4 instructions per IT block.

(b) Write out the sequence of assembly instructions that would be executed by this code and thus state what final value will appear in register **R3** when the assembly block exits.

Include an explanation of the semantics of the **ITTET NE** instruction.

**(15% marks)**

**ITTET** **NE** means the 1st instruction following the IT (‘if-then’ instruction) executes if the condition is **NE** (arising from the CMP instruction). The next three instructions execute if the **NE** condition is TRUE (**T**hen), FALSE (**E**lse) andTRUE(**T**hen) respectively, reading left to right. The instructions used have to match in their (otherwise optional) appended condition codes ‘NE’ or ‘EQ’, which must be boolean opposites. Up to 4 instructions can thus be in an IT block. The executed sequence with the given values of ‘R1’ and ‘R2’ would be:

mylabel: CMP R1,R2 creates condition NE (as R1<>R2)

MOV R3,R1 R3 = 5

ADDS R1,R3,R1 doubles R1 to 10 (=1010 binary)

B mylabel so repeats the IT block…

mylabel: CMP R1,R2 creates condition EQ (as R1=R2 now)

AND R3,R1,#0x03 AND off everything in R1 (10**10**b) except

for bottom 2 bits so **R3 = 2** finally.

Note that this example also appears in your ARM notes, almost verbatim. However, I saw cut and pasted copies of this:

**“ITTET** **NE** means the 1st instruction following the **IT** (‘if-then’ instruction) executes if the condition is **NE** ("*not equal*", arising from the previous CMP instruction). The next three instructions execute if the **NE** condition is TRUE (**T**hen), FALSE (**E**lse) andTRUE(**T**hen) respectively, reading left to right. The instructions used have to match in their (otherwise optional) appended condition codes **"NE"** or **"EQ"**, which must be boolean opposites.”

* straight from my notes. Be careful, that’s *plaigarism*.

The Instruction Set summary also contains this, pasting it would also be plaigarism:

“Makes up to four following instructions conditional, according to **pattern**.

**pattern** is a string of up to three letters. Each letter can be **T** (Then) or **E** (Else). The first instruction after **IT** has condition **cond**. The following instructions have condition **cond** if the corresponding letter is **T,** or the inverse of **cond** if the corresponding letter is **E**. See **Condition Field** table below.”

1. (a)… **calculate what value would end up in register r0** after the highlighted instruction:

**ADD r0,r6,r0,LSL #1**

has been executed, given the current register and memory values shown above.

**(10% marks)**

**LSL** = shift left logical by #1 position doubles value in right-hand register, so the value 5 in r0 changes to 10.

r6 contains -1 in 2’s complement (= 0xFFFFFFFF), so, 10 + (-1) = 9.

**So, r0 = 9**

(b) Identify which particular part of the given C++ source code this one instruction is implementing, and briefly explain how it is doing so.

**(10% marks)**

It implements part of the FOR statement on line 41: (branchLine \* 2) - 1

The **LSL** (left shift logical) does the multiply by 2 and adding the -1 currently in r6 (showing as 0xFFFFFFFF = -1 in 2’s complement) does the rest.

(c) Which C++ identifier is register **r1** being used for: **leaves**, **LEAF**, **putchar**, or **branchline**?

**(5% marks)**

leaves

(d) Explain the semantic difference between these two similar looking load register instructions, and give an example of a data access scenario where the second would be useful.

**(10% marks)**

The second one has an auto-update decoration ‘**!**’. This means that the addition inside the square brackets (the *effective address* calculation) is used, then made permanent in the address register, **r4** in this case; so **r4** becomes **r4+4** here. This is referred to as “Pre-Indexed addressing”.

In the first version without the ‘!’ the value of **r4** would remain the same after execution (also called “Immediate offset addressing”). A few of you called this “Post-Indexed addressing, incorrectly – the position of the closing square bracket ,“]” is important here!10/10

The second version avoids having to use an extra add instruction to increment the address register r4, in particular when sequentially accessing data in array or other linear data structures. The immediate value (‘offset’) used would reflect the data element’s size in bytes (4 = 32 bits, and so on). It’s effectively an implementation in hardware of the ‘**++**’ in languages like C++.