All individual questions got fully answered by at least one student (typically many more). However, there was quite a variance in the overall performance. All asked questions were very closely related to the course material and the lab assignments. However, answers giving variants from the course material got marks, as long as answers were reasonable.

It was surprising to see quite some students who have not been able to discuss the main factors that impact code size, processor performance and processor power consumption when using standard-32bit ARM instructions and 16-bit Thumb instructions. Here, different answers were accepted as long as they were supported by arguments, like for example:

Thumb typically needs more instructions but is using a smaller instruction memory footprint. Consequently, the performance depends on the problem: if we have a bus that cannot deliver sufficient throughput for fetching 32-bit instructions, Thumb will probably be faster. It can also be faster, if we are able to fit the problem (the complete binary or frequently used parts) into on-chip memory (due to the smaller memory footprint). This way, Thumb is the better option for lower power as we have less off-chip traffic. However, 32-bit instructions allow in many cases to accomplish a job faster, so we could use this to run at lower clock speed and supply voltage or to power-down the system.

The next question asked for discussing the impact of Thumb instructions instead of standard 32-bit ARM instructions on performance and power when considering different (hierarchical) memory subsystems.

This is clearly stated in the course material: the higher up in the memory hierarchy (registers, L1, L2, memory, disk) the faster (throughput) and the lower the latency, but the more power for storage (a cache needs more power per byte to keep the information than a large disk). Furthermore, transfers between hierarchies cost power (off-chip communication is more power costly than on-chip). This can be used, for example, as Thumb reduces code footprint, so less memory transfers to higher parts of the memory hierarchy will save power and less memory transfers to higher parts of the memory hierarchy are faster and have lower latency.

For the question about how multicore CPUs can help reducing power consumption, multiple answers were possible (as long as they were explained), like for example:
- Parallel processing allows a lower clock speed, which allows a lower supply voltage, which in turn results in a power saving as the impact of the supply voltage on (static) power is quadratic.
- Parallel processing allows finishing tasks faster which can be used to power a system temporarily down for power saving.
- Specialized processors in a multicore system might be used to adapt the system to use the entire most power efficient mapping for different load scenarios (ARM big-Little). So during low load, we can switch-off higher performance (and higher power consuming) CPU cores.

For changing the 'Hello World' program from your first lab to start with TEXT (ADR r1, TEXT), you have to remember from the labs that SWINE SWI_ANGEL prints the register pointed to by r1. Consequently, you cannot easily use any kind of auto indexing. Clever programs needed one extra instruction for initial change, while others did the r1 increment after SWINE SWI_ANGEL (both answers got full marks). The last option will slow down the program as we have an extra instruction in a small loop body, but the impact depends of the actual SWI_ANGEL implementation (which will take many more cycles).

The question on why on-chip RAM is used in preference to a cache in some embedded systems could be answered as follows:
The system becomes cheaper and uses less power; and it makes the system more predictable (something that is highly relevant for real-time systems).