**Q1a. dark silicon (discussed in lecture #29/30)**

The canonical reference for works related to Dark Silicon is the paper titled "Dark Silicon and the End of Multicore Scaling" [1]. You can skim through the abstract and the conclusion to get an idea of what the paper talks about. The primary issue for dark silicon are issues with scaling symmetric processors, which has led to the creation of (SMT) multicore processors. Even with more cores, the power consumed is increased disproportional to the workload demands, which is where dark silicon can help by powering off unwanted parts. Complexities lie in how these parts should be designed or controlled, which requires special circuitry.

In the attempted answer, the word asynchronous is used, which you must thoroughly understand before you can apply it in this context. For hardware circuits, asynchronous circuits are those which do not depend on clocks, but rather on (completion) events. They may or may not be applicable to the dark silicon processors depending on how someone designs them.

The answer does veer in the right direction by making the comparison of the big.LITTLE architecture (in reality, any vSMP or asymmetric multiprocessor architecture will do) to demonstrate how different 'silicon' structures can be powered on demand. The question asks what are the issues in creating and executing such structures.

Remember the magic words: power, efficiency, performance. There are costs and complexities associated with each in this case. The circuit is more complicated, scheduling and optimisations would need to be made more specific to the variable processors, which is even more complicated. Finally, the instruction set that is used may need to be updated for the architecture to take advantage of new features.

While the answer does provide a good overview, one would expect more discussion of these issues and clarity into its applicability. The idea to use an example use-case from the real-world (ARM in this case) is a good approach.

**Q1 b. Compare and swap**

The interpretation of the question is incorrect (slightly) in the given attempt. The question provide C code for the C&S instruction (which is ONE machine instruction).

Here, the address refers to the address of the lock, and testval and newval are used by any thread/process used to acquire and release the lock.

If we assume that value at address refers to state of the lock variable, and 1 means the lock is acquired, and 0 means the lock is free, the C&S algorithm works somewhat like this:

1. no thread/process has the lock, so the function is called as (address, 0, 1) --> if lock is free, set it for me

2. in the function, oldval = 0 = testval because the lock is free, so newval (1) is set at the address

3. other threads/processes try to call the function to acquire the lock as (address, 0, 1) but fail because now oldval = 1 and testval = 0 which do not match

4. only the thread/process which has the lock knows it is locked (responsible behaviour), so it calls (address, 1, 0) when freeing the lock, which sets the value of address = 0

The question does not relate to cache coherency, or any cache management. Nothing prevents or stops a thread/process from ignoring or not using locks and freely manipulating data. But locks are provided so that the resource can only be accessed by one execution unit at a time, which is a voluntary design pattern. The C&S alogithm is used to implement locks, which are used in code as:

acquire lock(x);

// do something

release lock(x);

**Q1 c.**

1. code is clearly vectorisable; however, you 'assume' that the address is n-byte aligned (4,8,16 depends on instruction set), which you must mention since \_load\_ps is used and not \_loadu\_ps. Also, va is an unused variable in the code.

2. sum is not a scalar value, it is constantly changing and is depending on the previous iteration of the code, and thus cannot be used in an vector instruction; If you change the algorithm (asserts? \_hadd\_ instructions?) you must mention the approach or atleast put comments. The problem can be partially vectorised only for the multiplication instruction (as correctly stated).

**Q2.**

If you use qsort (what is qsort? is it stated in the question? in the answer?), why isn't that parallelised? Is that not a good candidate for parallelization? If you believe so, did you state why? The reasons/justifications/notes provided are not enough to explain these. Use the assignment as a guideline for what issues are important for parallelisation of algorithms.

**Q3.**

1. OoO Superscalar

The choice of words - "begins execution of multiple instructions at once" is very misleading and implies underlying parallelism, which superscalars (or out-of-order) does not provide, so I would be careful of using such sentences as it changes the viewpoint of what the hardware is doing. Instead, once could say, executions can happen as soon as (functional) units as free or results are available if based on previous dependencies. Both ILP and out-of-order are based on a single core processor.

Another choice of words - "if an instruction takes a large amount of cycles to execute, the other instructions will be done before it. on condition that they are not dependent on the currently executing instruction.. Leading to out of order code execution." is also misleading. Do instructions that take longer to execute get relegated to the back of the queue? (no, they don't). It is only when there are dependencies that instructions are pushed back in the queue, allowing instructions without dependencies to be executed earlier.

The code will see (slightly, maybe) benefit from both OoO and ILP. To get an idea, write the assembly code for the above code and see if you can identify dependencies and re-order execution to execute at least one instruction out of the intended order for OoO and if instructions can be combined for ILP. The answer should justify whatever your line of reasoning is.

2. VLIW

The approach seems to be along the right lines of reasoning

3. Vector

Correct reasoning, however, you should probably also mention how to vectorise the code. Both lines are candidates for vectorisation.

4. SMT

Correct reasoning, especially about dependencies

5. Shared memory multicore processors

Same line of reasoning about SMT, you need to be clear about overhead for sending instructions to different units and synchronising their results.

6. multi-chip symmetric processor

Imagine executing multiple threads on each of these processors. IF your answer differs for SMT and SMP, then you usually have a problem in either of those. In most cases, they will be the same.

7. NUMA

No real speed up because there is no access non-uniform memory locations, and also, depends on your answer about using multiple processors. If multiple processors can be used, and memory locations are an important performance point, NUMA will be helpful.

8. distributed memory multicomputer

A more demanding version of #6, so the answer will be largely the same with more independence for each computer, and more overheads on synchronisation. Also, this requires a master-slave or job-master type arrangement, which is additional overhead.

Do note that all of the answers try to explain what the architecture is (which you can safely assume that the lecturer who will also be correcting the exam papers already knows) without explaining what impact this will have on running the code. Focus on the code, and explain using important points about the architecture. For example, for vectors, the main thing is multiple instructions can be executed into one vector, and so, how? SSE instructions for example, and which code can be vectorised (point it out in the given code); and the underlying assumption about dependencies, etc.

The formatting, grammar, and structure of this email does not construit an invitation to submit similarly poorly worded answers ;)