QUESTION 1.

As always, many people lost a significant proportion of the marks by not doing what the question required. For instance, I didn't count them, but I suspect that most of the answers didn't include examples of addresses or explanations of how they work for parts (a) or (b), which together make up about a quarter of the marks for the question.

The next most common, and more important, error was that many people didn't distinguish between memory model and implementation – so I had to read quite a lot of stuff about base and limit registers, and about searching memory, where it really wasn't relevant. It isn't a particularly good idea to dump everything you know about a topic on the off chance that some of it might be right; the result is usually to demonstrate that you understand so little about the topic that you can't select the appropriate bits by yourself.

Another was confusion between memory models and memory management. A segmented memory model is not the same as segmented memory management. The first is seen by the programme, the second is implemented by the system. They are both concerned with segments, but – as is common with different parts of operating systems (which, I suppose, is why they're different parts) – deal with them in quite different ways. (No, it wasn't a trick question – it never occurred to me that there would be any confusion, any more than there is between roast chicken and roast chestnut.)

One revealing comment is worth noting. (It appeared in part (b), but I've promoted it because I think it's of general significance.) I think only one person said it in as many words, but it came through by implication in many other answers. It was that the flat memory is obvious, because it was "the one I am the most used to". Apart from that not being a very good reason for anything, I very much doubt if it's true. Every time you use a variable's name instead of its memory address, you are assuming a memory model in which there are many objects which are essentially separate, and don't have any relative positions. Every time you use a one-dimensional array, you are using a segment model (the address includes a name and a displacement). Every time you use a procedure, you assume some sort of segment model in that you think of items in the procedure as existing in an area inaccessible to those outside. If you use a procedure recursively, you are risking all on a stack model. (It is quite hard to implement a recursive procedure in a system without a stack model.) Now, hands up, all those who really are most used to a flat memory model. This is why I always urge you, in lectures and in the notes, to apply the topics you meet in the course to your experience of using computers.

(a)

Flat model : There is a single memory area of arbitrary size, addressed sequentially.

The address is an integer, being the serial number of the memory location. Example : (1234).

The example identifies the 1234th (or maybe the 1235th) location in the memory.

Segmented model : Memory is composed of several memory areas, called segments. Each segment can be identified, and is of arbitrary size, and locations within segments are addressed sequentially.

The address is a name (commonly represented as an integer) and an integer. The name identifies the segment; it is a name rather than a number because the sequence of segments is not significant. (Numbers are usually used because it is convenient to impose an order on the segments for implementation purposes, but the order isn't part of the model.) The integer identifies the serial number of the memory location within the segment. Example : (X, 987).

The example identifies the 987th (or maybe the 988th) location in the segment named X.

Linda model : Memory is associative. It is composed of a collection of tuples, each an ordered set of an arbitrary number of components. For each component, the value and type are arbitrary, but known.

The address is a tuple with the types of all components specified; values for tuples may or may not be given. Example : (integer, "Rabbit", logical).

The example identifies any triple in memory with an integer as the first component, the text string "Rabbit" as the second component, and a logical value as the third component.

There were five marks for this part, and three parts in the part, so I allowed two marks for each part of the part. Because the total couldn't exceed 5, people who wrote completely acceptable (not necessarily perfect !) answers have marks { 2, 2, 1 } for the parts. That doesn't mean that there's anything wrong in your Linda answer – just that I had no more marks to give away.

Why did so many people say that segments were ***equally-sized*** *chunks of memory ?*

The several people who didn't know what the Linda memory was had presumably not attended lectures, not read the notes, and not looked at the old tests and examination papers.

(b)

Memory is composed of several memory areas, called segments. Each segment can be identified by a pair of numbers acting as coordinates, and is of arbitrary size, and locations within segments are addressed sequentially.

An example of an address is (7, 64, 334). This identifies the 334th (or maybe the 335th) location in the segment identified by the coordinates (7, 64).

There was a strong tendency to talk about implementation in this part, where I didn't want it; the memory model just says what happens, not how it works. It's important to separate implementation from the abstract data structure. (So, for example, if you want a structure which is a stack of files (to take a case at random), you can implement it by encoding the stack position in the file name – then you don't need a physical array of pointers or anything like that.)

Having explained on the question paper that "the most obvious feature of a memory model is the form of the address which it requires", I quite expected that answers to this part would concentrate on addresses. They didn't.

Several people suggested Linda. That was a surprise, but it's by no means a silly idea. It's easy to formulate the address and data parts of the cell (13, 25, "string"), there is absolutely no difficulty in extending the array whenever you want to, and it offers an instant search function (?, ?, "string"), which could be very handy. (You'd presumably use a version of Linda without types, as you always know that the first two items are integers and the data could be anything.) But it falls down a bit on details, particularly access to items within the data (the "i" of "string"). If you never wanted to get inside the cell data, that wouldn't matter, but in fact you do : the question mentions expressions specifying computations. (And see below.) You might also have a minor problem with Linda's ability to accept two or more items with the same coordinates – but you could define a modified model which wouldn't allow that.

Various answers mentioned the "page model", the "virtual memory model", etc. Neither of these is a memory model – both are implementation techniques which can in principle be used with any model at all.

The idea of defining a memory model for some sort of computational activity is, more or less, to identify the easiest way to refer to memory when you're writing the *programme. So why did quite a large number of people start by saying something like "A flat memory model could be used ....", then spend some time (both theirs and mine) going on about how to calculate where the data are ? No one who's sane seriously wants to do all that work; the memory model defines the transformation that you want the system to do for you.*

Most answers said nothing about identifying locations within the cell data. Something has to address the locations within, say, a string in order to put the characters there and read them back again, to say nothing of searches, editing operations, evaluating expressions, and so on. It may be that the operations will be implemented by copying the cell datum somewhere else to do the internal work (though that's clumsy and unnecessary), but that's beside the point.

A number of people favoured a linked structure. The reasons given (if any) were various, but a fairly common opinion was that it was easy with the links to identify rows and columns, and to find the immediate neighbours of a cell. Is it really easier to find the data of cell (2,3) by something like (down, down, right, right, right) than by looking it up directly in a two-dimensional array ? If you think it is, what about the data of cell (200, 300) ? (And please observe that a 2-dimensional array is NOT the same as a 1-dimensional array of 1-dimensional arrays : a cell in a 2-dimensional array has the same sort of relationship with all its four nearest neighbours.)

(c)

Use the segments to represent the cell data. The displacements within the segments will be handled satisfactorily by the segmented memory implementation, as will the variable sized data, so we needn't worry about those possible sources of trouble.

The spreadsheet's segment coordinates must be mapped onto the segmented memory's segment names. The easy way to arrange that is to maintain a two-dimensional array of segment names, used in the obvious way to identify the required segment. The size of the two-dimensional array must be adjustable to cope with the definition of new cells outside the current array boundaries.

I think that's good because :

- The underlying segment management will look after memory management for the arbitrarily sized segments.
- For empty cells, no space is wasted except for the location in the two-dimensional array. (One could eliminate the "wastage" within the array in various ways, but it's probably counter-productive; any distortion of the array will make it much harder to implement spreadsheet operations on whole rows and columns. That isn't required by the question, of course.)
- There is a cost in time associated with changing the array dimensions as new cells are defined. This could be avoided in several ways : for example, by using linked structures for the implementation (with a concomitant cost in space, and probably in search time to find the required coordinates), or by using a hash table (which may save time when setting up the spreadsheet, as it avoids the resizing, but will probably take a little longer in execution and won't save much, if any, space). In most cases the requirement to resize the array only happens while the spreadsheet is being set up, so doesn't affect execution. Retaining the array of segment names also maintains the simple structure in which rows and columns are clearly defined.

A serious difficulty with this part of the question was that many people had at least partially answered it under (b), where it shouldn't have been, and therefore hadn't much left to talk about. I gave some credit for stuff in (b) if it seemed appropriate, but that wasn't very often.

The question stated that you were given a segmented memory. It didn't actually say that you had to use it, but I think there was a fairly strong implication. (As there was no *suggestion that you were given anything else, you didn't really have any choice.) Nevertheless, several answers made no reference to the segmented memory at all.*

I haven't said anything about how the array of names is stored, because that's not the problem at this level. We don't usually worry about how to store a segment table when constructing a segmented memory manager; once we've made the memory work corectly, the details will look after themselves. (We may have to tidy things up a bit afterwards in the interests of efficiency – for example, we may want to lock a segment table in memory while it's in use – but that's a different sort of problem.)

Several answers contained curious references to "processes". (They turned up in all parts of the question, but were perhaps most prominent in this part.) Sometimes it made sense if I assumed that "process" meant "process's memory"; in other cases, it seemed that a spreadsheet cell was thought to contain a process. I didn't understand what was intended by this use of "process". Certainly the cells could contain code which could be used by a process, but that's not the same as a process.