MEMORY SPACE

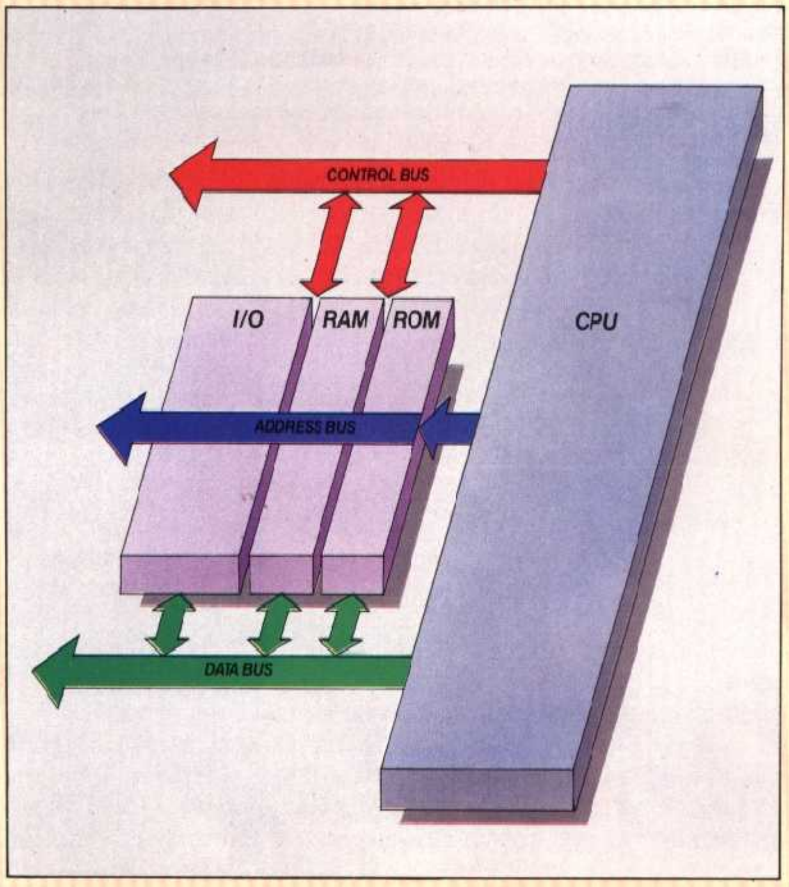
At this stage in the course, we look at ways of finding or reserving space in memory to store our machine code programs. We also take our first look at how we can perform a simple arithmetic task by using machine code instructions to manipulate the contents of the accumulator register in the Central Processing Unit.

In the last instalment of the Machine Code course, we developed a very simple Assembly language program, translated (assembled) it into machine code, loaded it into memory and executed it. We used the Monitor program on page 118 for the latter two tasks. If the program was a more sophisticated package containing an assembler, we could have used it for assembling our machine code program as well. At this stage in the course, it's no great hardship to do the assembly by hand — indeed, it's very educational. But once you've grasped the principles of the process, and as your Assembly language programs get longer, there won't be any point in concerning yourself with the actual machine code translation. In fact, with larger programs, assembling by hand gets very tedious, and is prone to error. Consequently, when you reach this stage in learning machine code, you may want to acquire an assembler program suitable for your machine.

There were many significant points about using the short machine code program that we gave (see page 117). We used one of the CPU registers to manipulate memory, we had to decide where in memory to store the machine code, and we caused the microprocessor to execute it. These are all aspects of Assembly language programming that particularly puzzle a beginner, and it's worth looking at them more closely. Let's start with the question of where to store the machine code.

To the CPU the only difference between one byte of memory and the next is whether they're read-write memory (RAM), or read-only memory (ROM). ROM chips contain system programs and data that must be protected from accidental or deliberate overwriting, and therefore can only be read. ROM can't be written to, so we can't load a machine code program into ROM. Those areas of memory apart, there's theoretically nothing to stop us loading a program into any other part of memory, but there are practical considerations that prevent us using some areas.

The CPU uses certain sections of RAM for temporary storage in the course of its operations, and if we load a program there, then either it will simply be corrupted by the CPU's overwriting it, or (and this is more likely) the CPU will read our machine code as if it were some of its own data. The operating system also uses large parts of RAM for storing its working data, and for running the computer system. Loading machine code programs (or anything else for that matter) into any of these areas would be unwise or impossible for the same reasons that prohibit use of the CPU's workspace. Furthermore, BASIC programs can take up all the remaining RAM — partly as program text and partly as variable storage areas. Once again, it's unwise to tamper with these areas, and so the programmer



Small System Architecture

A typical computer system, in its most schematic form, comprises memory and a CPU. The former is made up of ROM chips (containing system programs), RAM chips, and specialist chips dedicated to input/output operations.

Data and control signals flow into and out of the CPU and around the system along *buses*. These are routes — very similar to ribbon cables — which can carry a byte or more of data at a time. These buses may be uni-directional like the *address bus*, which only transmits in one direction, or bi-directional like the *data bus*, which can transmit in either direction. The *control bus* carries switching information around the system, opening and closing logic gates to direct the flow of data. The *address bus* carries a 16-bit address from the CPU to select one byte of memory, allowing data to flow along the eight-bit *data bus* into or out of the byte