the limits of a byte (in other words, the range of decimal numbers 0-255). Once we encountered the meaning and appropriateness of binary arithmetic, the limitations of the decimal system for dealing with the world of Assembly language became apparent. In exploring the idea of paged memory we saw how the size of the logical pages must be a function of the number base, and in a binary system that means that the page size must be a power of two. Two to the power of eight gives 256 — the magic number in an eight-bit microprocessor system.

Binary very quickly became too unwieldy and too prone to error for use as a numbering system, and we passed on to hexadecimal (number base 16) arithmetic. We saw how the eight-bit byte can be fully represented by two hex digits, from $00 to $FF, one digit representing the state of the lower four bits, and the other standing for the upper four bits of the byte.

The way that BASIC programs are stored in the program area was exhaustively examined. By describing tokenisation as another form of machine code, we gave a useful insight into the operating system. Our discussion of end-of-line markers showed how the BASIC interpreter handles the difficulty of telling where one piece of code ends and another starts, and the Commodore's link addressing introduced both the lo-hi address convention and the idea of indirect addressing.

From there we moved directly into Assembly language itself. We started from the primitive operations of the CPU as directed by the eight-bit op-codes that constitute its program instructions. With the idea of coding so thoroughly explored, it was a short step to Assembly language mnemonics. Once we had made that step it became clearer that programming in machine code or Assembly language or BASIC was still just programming, and that what counted was solving the logical problem before worrying about how to code the solution. Problem-solving has been the central theme of the course. But the obscurity of some of Assembly language's concepts forced our attention first to clearing the haze of confusion that besets most people on first contact with low-level languages.

The course proceeded to spend some time on the practicalities of loading and running machine code programs on computers that were more or less dedicated to running BASIC programs. We looked at system variables and operating system pointers on the BBC Micro, Spectrum, and Commodore 64, and learned how to 'steal' space from BASIC.

We glanced at the architecture of small computer systems and the Z80 and 6502 CPUs, and moved on to begin writing Assembly language programs that manipulated memory and the accumulator. Assembler directives or pseudo-ops were introduced here, a step towards practicality and the real world, but also a step away from machine code, manual assembly, and the laborious detail of low-level programming.

The need for the logical constructs of a programming language was now obvious, and we turned to considering the processor status register (PSR). Its role as a recorder of the results of CPU operations was immediately illustrated in an introduction to binary arithmetic, using the 'add with carry' instruction. The central role of the PSR and, in arithmetic, of the carry flag, was obvious as soon as it was seen. The course has concentrated on the processor status register and the associated instructions since then.

We briefly examined the various addressing modes; indexed addressing was given most attention because of its importance in handling loops, lists and tables. The need for a class of instruction to change the flow of control in a program is evident once these structures are introduced, so we began to examine the conditional branch instructions while still exploring the potential of indexed and indirect addressing. With conditional branching, primitive arithmetic and array-type structures, we have almost all the bones of any programming language. Fleshing out the form through practice and systematic investigation is the remaining task.

The Assembly language subroutine call and return was examined both for itself and as a way of introducing the last unexplored area of the operating system — the stack. Seeing how it works, what it is for, and how we might use it introduced some new ploys to the repertoire of machine code programming, while a more searching look at the CPU registers and their interactions introduced new possibilities in the manipulation of memory and the microprocessor.

Finally, with a working knowledge of the architecture of the microprocessor and a vocabulary of op-code instructions, we approached binary arithmetic. The oddities of subtraction and two's complement, and the complexities of multiplication and division have all been covered in detail. Looking ahead, we will investigate the practical craft of machine code programming by investigating and exploring specific tasks for the processors we have initially concentrated our attention on (the Z80 and the 6502), as well as other processors, such as the 6809 CPU used by the Dragon 32 and 64.

Answers To Exercises On Page 299

1) The fastest-running solution is certainly a routine written specifically for 16-bit multiplicands, on the same lines as the eight-bit routine in the last instalment. On the other hand, if you split 16-bit multiplication into two separate eight-bit multiplications (multiplier by lo-byte, followed by multiplier by hi-byte), then you can call the existing eight-bit routine twice, adjust for a carry out of the lo-byte, and store the results in the product bytes.

2) A multiplication routine using repeated addition consists simply of a loop whose counter is the value of the multiplier; each time the loop is executed, the multiplicand is added into the product.