In the BASIC Program Text Area this line:

```
10 REM**********************
```

has 25 consecutive bytes containing $2A, the ASCII code for '*'. These bytes are never inspected by the operating system or the BASIC interpreter because to them the command REM means 'ignore the rest of this line'. Once the line has been entered into a program in this form, a machine code subroutine can be loaded into the asterisk bytes, where it will reside untroubled by the interpreter. The big advantage of this apparently messy method (often used in programs for the expanded ZX81) is that when you SAVE, and subsequently LOAD, the BASIC program, then the machine code subroutine goes along with it. Using the other methods described usually means having to save the machine code separately from the BASIC program. The trouble with this method is that LISTing the line causes the operating system to interpret the machine code bytes as ASCII character data, which may corrupt the screen display. This explains why there were warnings embedded in the Demonstration programs on page 19 for the BBC Micro and the Spectrum. The Commodore 64 version of that program loads the machine code subroutine into the cassette buffer, whereas the BBC and Spectrum versions load it into their opening REM line — hence the warnings about not LISTing these versions of the program.

Once you've written an Assembly language program, assembled it into machine code, and LOADed it into the RAM of your choice, then you can proceed to execute it. This is done through the BASIC commands CALL (BBC only), SYS (Commodore 64 only) or USR (all three machines). Each of these commands is followed by the address of the first byte of the machine code program, wherever it's stored. All three commands mean the same thing to the interpreter: 'execute the machine code program starting at the address given, and return to execute the next BASIC instruction when the RET or RTS op-code is executed'. It is similar to the GOSUB command in BASIC.

In the last instalment we wrote a program to copy one byte's contents into another byte by loading the accumulator with the contents from one address, and then storing the accumulator's contents in the other address. This illustrates the centrality of the CPU's role in the entire system: data and control must flow from memory, through the CPU, and back to memory. Whereas in BASIC we can write LET X=Y (meaning 'copy the contents of Y directly into X'), in Assembly language we have to copy into the CPU from memory, then out of the CPU back into memory. The CPU registers (see the accompanying panel) are the bytes of RAM inside the CPU itself where data from memory is stored or manipulated. Both the Z80 and the 6502 have a register called the *accumulator*, which is referred to by a majority of the Assembly language instructions, and is the register in which arithmetic is chiefly done.

Suppose we want to add the two numbers $42 and $07 (remember that the symbol $ means the number is hexadecimal). We simply put one of them into the accumulator, and add the other in on top of the first — their sum will literally 'accumulate' in the register. Here are the instructions to perform this:

Z80		6502	
LD	A,$42	LDA	#$42
ADC	A,$07	ADC	#$07

Here the Z80 instructions both refer to the numbers to be loaded and added, whereas in the 6502 version the numbers are preceded by #, which shows that they are actual numbers rather than addresses. Thus, LDA #$65 means 'load the number $65 into the accumulator', whereas LDA $65 means 'load the contents of the byte whose address is $65 into the accumulator'. Similarly, the add instruction, ADC (it happens to be the same mnemonic in both Z80 and 6502) means in this case: 'add an actual number into the accumulator'. The numbers $42 and $07 are said to be 'immediate data', and LDA #$42 may be read as 'load the accumulator with the immediate data #$42'.

After these two instructions have been executed, the sum of the numbers will be contained in the accumulator. It is 'invisible' to us there, so we must store the accumulator contents in a byte of RAM where it can be inspected. The program must end with a return instruction, and must begin with an instruction to put an associated CPU register into the correct state, so the full programs read as:

Z80	
Machine Code	**Assembly Language**
A7	AND A
3E 42	LD A,$42
CE 07	ADC A,$07
32 ?? ??	LD BYTE1,A
C9	RET
6502	
18	CLC
A9 42	LDA #$42
69 07	ADC #$07
8D ?? ??	STA BYTE1
60	RTS

We won't bother about the meaning of the first instruction in either program at the moment, but notice that the fourth instruction contains the symbol BYTE1, rather than an actual address. The value of BYTE1 will be different from machine to machine, so we'll just use the symbol here, and replace it by a real hex number when we come to assemble the code.

Now we must decide where to locate the machine code, and what address BYTE1 represents. Choose a place to store the machine code and then make BYTE1 equal to the address of the byte after the end of the program, and put that address in lo-hi form into the machine code. After that use the Monitor program on page 118 to load and execute the machine code, and to inspect the byte where the result — $49 — should be stored.

Accumulator
This is really the central register of the CPU. Arithmetic and logic operations, as well as data transfers, are chiefly conducted via this register — more so in the 6502 than the Z80

Arithmetic And Logic Unit
Comprises a binary adder and logic gates, which permit access to individual bits of the registers and data bus. Properly controlled, it enables addition, subtraction and Boolean operations

Processor Status Register
Whenever a CPU operation is performed, the individual bits of the PSR show some of the effects of the operation — does it cause a zero result, for example, or is there a carry bit from an addition operation?

Program Counter
This points to the address in memory where the next op-code to be executed by the CPU is stored. The BASIC function USR(address) causes the address specified to be loaded directly into the program counter, so that CPU execution proceeds from that point

Stack Pointer
Carries the address of the next byte in RAM of free workspace for the CPU's use. Whenever the CPU writes some data to the stack, the stack pointer is changed to point to the next free byte

Index And General Purpose Registers
Index registers are used by the CPU to address memory in a variety of ways, while the General Purpose Registers are used as general workspace, and for specific CPU purposes

KEVIN JONES