bytes. The machine code can be saved as a file for later use, or loaded immediately into memory for execution.

In performing the assembly for us, the assembler can be made to perform other tasks that we've been doing by hand — attaching the location addresses to each line of the program, for example. Another pseudo-op, ORG, does this for us. It is added to the program like this:

Z80		
0000		ORG $A000
A000		BYTE1 EQU $0009
A000	A7	AND A
A001	3E 42	LD A,$42
A003	CE 07	ADC A,$07
A005	32 09 00	LD BYTE1,A
A008	C9	RET

Notice that the location address attached to the first line of the program is $0000, but the address of the following line is $A000, which reflects the effect of the ORG statement. Furthermore, notice that no machine code bytes appear on the lines containing pseudo-ops, precisely because they are not parts of the program and are not to be translated into machine code. Because they are features of the assembler program rather than elements of the CPU instruction set, pseudo-ops do differ from one assembler program to another. EQU, for example, is sometimes replaced by '=', and ORG by '.='. The effect is the same, however, and we shall continue to use ORG and EQU as if they were standard.

It may have occurred to you, while reading about assembler directives, that we've been using a pseudo-op almost from the start of the series: '$', the hex marker. This is no more than a directive to the assembler that what follows is to be treated as a hexadecimal number. Similarly, '#', introduced in the last instalment, is the 'immediate data' marker, signifying that what follows is an absolute quantity rather than a pointer or a symbol. Taking this a little further we could in fact regard Assembly language itself as no more than a series of pseudo-ops. Indeed, there's nothing to stop you inventing your own mnemonics for the machine code instruction set, provided they correspond one-to-one with that set. One very popular assembler program for the Vic-20 does just that: it uses a non-standard version of 6502 Assembly language, largely for the sake of formatting the Vic's 22-column screen.

In this course, we shall continue to use what we've been using so far — the Assembly language mnemonics published by the chip manufacturers — but it does no harm to be reminded from time to time that everything that we call machine code is symbolic. The CPU is indifferent to everything except voltage patterns on its input/output pins, so how we describe those patterns is entirely a matter of convention.

Having finished with pseudo-ops for the moment, let's return to inspecting our program for other points of interest. In particular, let's compare the translations here to the LDA and LD A instructions with their translations in our earlier programs. Previously we wrote:

```
AD ?? ??     LDA $????          (6502)
3A ?? ??     LD  A,($????)       (Z80)
```

meaning 'load the accumulator from the byte whose address is ????'. The load-the-accumulator op-code in translation is $AD (6502) and $3A (Z80). Compare this with the second line of the current program, which incorporates the instruction 'load the accumulator with the immediate value $42'. Here the op-codes are $A9 and $3E, for the 6502 and Z80 respectively. But why are they different? Possibly you've figured it out for yourself. Although we're doing the same class of operation (transferring data into the accumulator) in both programs, the source of that data differs. Therefore, to the CPU, they're different operations, and have different op-codes.

In the first version, data is to be loaded from a byte whose address is given. Nothing is stated or implied about the contents of that byte; the CPU is instructed simply to copy those contents into the accumulator. The three machine code bytes, AD ?? ?? and 3A ?? ??, are decoded by the CPU to mean 'interpret the two bytes following this op-code as the absolute address of the data source'.

In the second version, the data to be loaded into the accumulator is actually in the byte following the op-code, so the two machine code bytes, A9 42 and 3E 42, are decoded by the CPU to mean 'interpret the byte following this op-code as the data source'. Something in the op-code (A9 or 3E) tells the CPU to load the accumulator from the next byte. Since its program counter always contains the address of the next instruction to be executed, the CPU can calculate the address of the source byte, and then do a simple 'load the accumulator from an addressed byte' operation.

This reinforces the point that the operations of the CPU are mostly very simple, uncomplicated procedures. One whole class of its operations (about a fifth of its entire repertoire) consists of operations that involve copying data from an addressed byte into one or other of its internal registers. These 'primitive' operations all involve one task — to transfer data from memory to a CPU register — and all that distinguishes one instruction from another is the format in which the address of the source byte is presented.

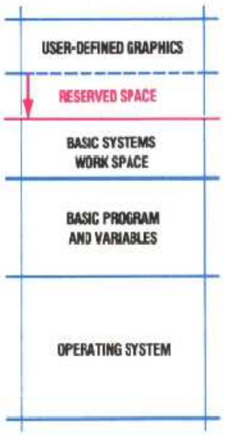
Digging this deep into CPU micro-operations is potentially confusing at first, but well worthwhile for the unifying insights it brings later. Such insights are unnecessary if all you want to do is write Assembly language programs for the sake of speed and efficiency. To do that you need only pick up the idea, learn the instruction set, get a few programming tips, and start right in. If you want to understand what you're doing, however, you'll want to do more than just add another programming language to your range, and you'll find that understanding how one processor works makes learning other Assembly languages enormously easier and more interesting.

Spectrum

In direct mode enter:

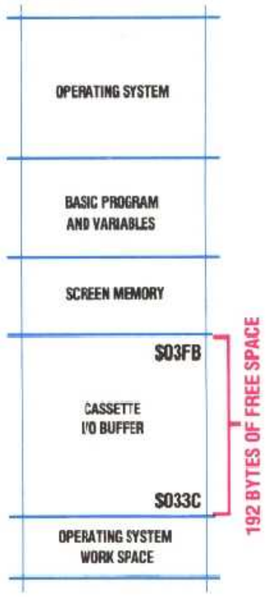
```
LET RTOP=PEEK (23730) +
256*PEEK (23731)
LET RTOP=RTOP-N
PRINT "RTOP= ";RTOP
```

where N is the number of bytes you wish to reserve for your program. Your reserved space starts at 1+RTOP



Commodore 64

Use the cassette buffer at $033C to $03FB



Warning

You must adjust these memory pointers immediately after turning on your machine when there is no BASIC program in memory