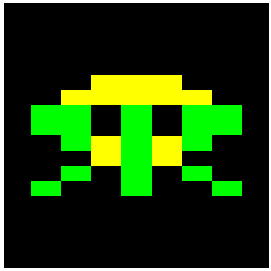


2

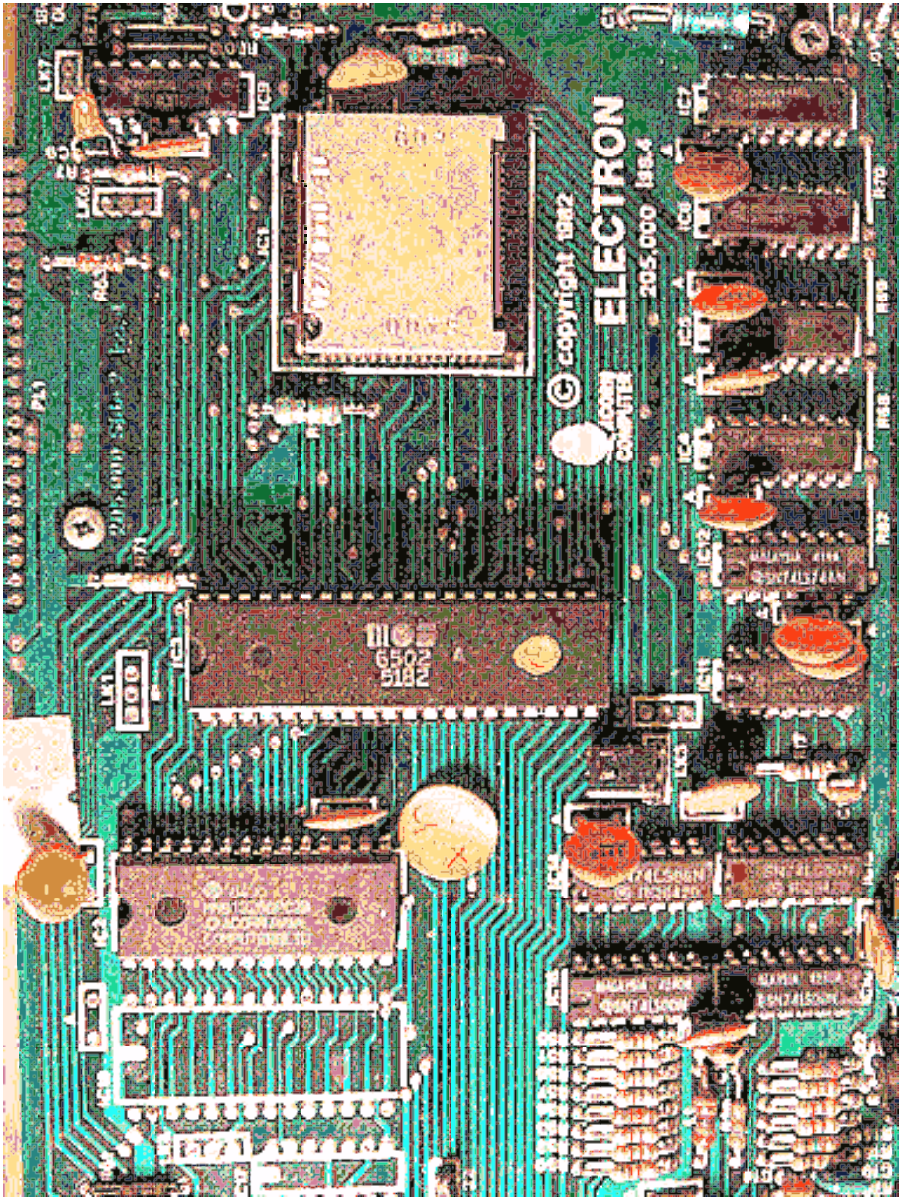
CARRYING OUT INSTRUCTIONS

The previous chapter showed how characters and numbers are represented in the computer's memory, i.e. how the computer deals with data storage. However, data on its own is useless to a computer, it also needs instructions telling it what to do with the data. This chapter looks at how the computer handles instructions and explains some of the simpler assembler instructions which it can use. The instructions listed refer only to the 6502 processor so if you have any other sort of processor connected to your machine, e.g. a Z-80 second processor, this will have to be disabled before attempting any of the routines given in this and subsequent chapters.

2.1 The CPU

The Central Processing Unit or CPU is the computer's brain. It is the most active part of the computer; although areas of memory can remain unchanged for hours on end when a computer is being used, the CPU is working all the time the machine is switched on. The CPU's job is to read a sequence of instructions from memory and carry out the operations specified by those instructions.

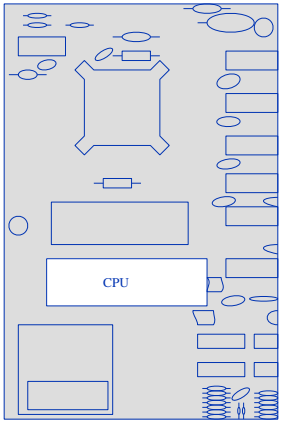
The instructions which the CPU acts on are just values stored in memory locations. The CPU takes a


ELECTRON
© copyright 1982
6502
5182

byte and interprets it as an instruction, e.g. &18 will be interpreted to mean ' clearcarry flag' ;this will be explained later in this chapter. It then performs the operation as instructed and goes on to collect the next byte.

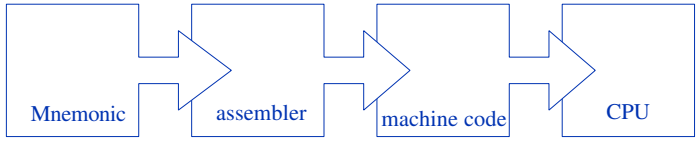
The first byte of all instructions is the operation code, or ' op-code' Some instructions, such as the example above, consist of just the op-code; other instructions require data on which they must operate. These instructions therefore consist of two or three bytes, the first one being the op-code and the other one or two consisting of data. For example the value &E6 is translated into the instruction ' increasethe contents of the memory location with the following one-byte address by one' .Hence the CPU then takes the next byte from the memory and interprets this, not as an instruction, but as the address of the location whose contents are to be incremented. It then adds one to the number stored in that location. Having executed this instruction, the CPU then goes on to the next byte which is taken to represent the next instruction it must perform.

The CPU, the computer's brain.



2.2 Machine code V assembler

The above few paragraphs should have given you the idea that everything the CPU acts upon is a number between 0 and &FF (255 decimal); each number being interpreted by the CPU as an instruction or some data which an instruction must use. The list of numbers which are being used are referred to as machine code. It is possible for us to talk to the computer in its own language, i.e. program in machine code, but this would mean that we would have to know which instructions all the op-codes stand for. Programming in assembler alleviates the need for learning all these translations. In assembler each op-code is represented by a three letter mnemonic, e.g. CLC is used instead of &18 to give the instruction ' clear carry' . The



computer then converts all the mnemonics into the corresponding op-codes.
This process is carried out by an assembler and hence is known as ' assembling' The program that the assembler takes as its input is known as the source code, and the machine code output is referred to as the object code.

2.3 The accumulator and the carry flag
The accumulator is just a temporary location inside the CPU which plays a part in many of the operations performed by the CPU. For example, to add two numbers together you have to load the first number into the accumulator from the memory, add in the second number from memory, and then store the result somewhere. To do this the following assembler instructions will be needed:

Mnemonic	Description	Symbol
LDA	load accumulator from memory	A=M
STA	store accumulator in memory	M=A
ADC	add memory to accumulator with carry	A=A+M+C
CLC	clear carry	C=0

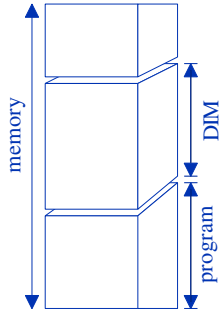
The carry is needed to allow numbers greater than one byte (255 or &FF) to be generated. When an eight-bit value is added to another eight-bit value the result could be too great to be represented by eight bits, e.g. 140 +160 = 300 (>255).

In order to allow for this, the CPU will use the carry as the ninth bit of the accumulator, and thus the carry will contain the extra bit. In the above example, when the numbers 140 and 160 are added together and the result stored in a memorv location, this location will contain the value 44 (300 MOD 256). By using the carry flag you will have a record of whether the result of the addition was actually the value 44 or if it was 300. Hence, to avoid confusion, clear the carry before performing any additions.

2.4 Writing an assembler program
Enter the following assembler program:

```
10 DIM P% 100
```

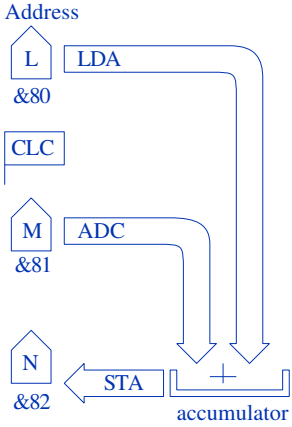
```
20[
30 LDA &80
40 CLC
50 ADC &81
60 STA &82
70 RTS
80]
90 END
```



The meaning of each line in this assembler program is as follows:

10 The DIM statement is not an assembler mnemonic; it is a BASIC instruction to tell the assembler where to put the assembled machine code by DIMensioning off an area of memory for it. The DIM statement is followed by a number (not in brackets) and the statement reserves this number of bytes for the machine code which will be generated. As a rough guide to the amount of room needed count the number of assembler instructions used, treble it and reserve at least this number of bytes.

The BASIC variable P% is used by the assembler as a location counter to specify the next free address. Hence the statement sets P0/o to the lowest address of the reserved block of memory and then as each byte of machine code is generated, P0/o increases by one byte so that it always points to the next free location.



20 The ' ['symbol is an ' assemblerdelimiter' which has to be used immediately before the first assembler statement to tell the BASIC interpreter that the following statements will be in assembler rather than BASIC.

30 Load the accumulator with the contents of the memory location whose address is &80. (The contents of the memory location are not changed.)

40 Clear the carry flag.

50 Add the contents of location &81 to the accumulator with the carry. (Location &81 is not changed by this operation.)

60 Store the contents of the accumulator to location &82. (The accumulator is not changed by this operation.)

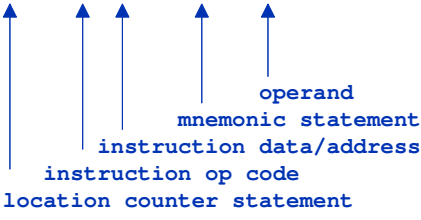
70 The RTS instruction will usually be the last instruction of any program; it causes a return to BASIC from the machine-code program. The mnenomic stands for ' return from subroutine' .

80 The ']'symbol is an assembler delimiter which has to be used after the last assembler instruction to tell the interpreter that the following statements will be in BASIC.

90 The END statement is not an assembler mnemonic; it just denotes the end of the program.

Now type RUN and the assembler program will be assembled; the assembled code being inserted directly in memory at the address specified by P%. An ' assembler listing' will be printed out to show the machine code the assembler has generated to the left of the corresponding assembler mnemonics:

```
>RUN
OE5D
OE5D A5 80    LDA &80
OE5F 18       CLC
0E60 65 81    ADC &81
0E62 85 82    STA &82
0E64 60       RTS
```



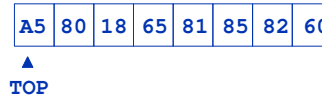
The program has been assembled in memory starting at &0E5D, immediately after the program text. This address may be different when you enter the example program if you have inserted extra spaces into the program or if you have filing systems other than cassette in your machine, but that will not affect any other part of the listing. All the numbers in

the listing are in hexadecimal; thus &18 is the op-code for the CLC instruction, and &A5 is the op-code for LDA when the number being loaded is not given directly but is obtained by looking in the memory location whose one-byte address is given. Hence this LDA instruction consists of two bytes; the first byte is the op-code, and the second byte is the address; &80 in this case.

Another method of finding out where the machine code is, is to find out where ' TOP'is by typing

```
PRINT ~TOP
```

This value gives the address of the memory location immediately after the program text. Since the machine code follows on straight after the text this address is the one corresponding to the first instruction, &A5. Thus the machine code is stored in memory as follows:



When ' RUN' was typed this assembled the assembler program and put the machine code produced into the computer' smemory, however it did not execute the program. The method for doing this is described below.

2.5 Executing a machine-code program

To execute the machine-code program at TOP, type

```
CALL TOP
```

Nothing obvious will happen except for the ' >' prompt being printed again on the screen. This indicates that the computer has finished executing the program and hence the contents of locations &80 and &81 will have been added together and the results placed in &82.

You can verify this by setting the contents of &80 and &81 to certain values by typing, for example

```
?&80=7 : ?&81=9
```

If you wish you can also set the contents of &82 to 0. Now type

```
CALL TOP
```

and then look at the contents of &82 by typing

```
PRINT ?&82
```

The result is 16 (in decimal); the computer has just added 7 and 9 and obtained 16!

2.6 Adding two-byte numbers

Try executing the program for different numbers in &80 and &81. You might like to try the following:

```
?&80=140 : ?&81=160
CALL TOP
```

We saw earlier in this chapter that if an addition generates a number greater than 255 then the result stored in the memory location specified will be that number modulo 256. Hence the result in this case will be 44 rather than 300. Here is the calculation in hexadecimal:

```
 160        &A0
 140        &8C
-----      -----
 300       &12C
-----      -----
```

Only two hex digits can fit in one byte, so the '1' of &12C is lost, and only the &2C is retained. Luckily the '1' carry is retained for us in the carry flag as was mentioned earlier, though we didn't see then how to use this. The example below shows how the two numbers can be treated as being two-byte numbers and added together using the carry to produce a two-byte number which is the complete answer. This method can be extended to any number of bytes since the carry flag makes it a simple matter to add together two numbers as large as we please. Modify the program already in memory by retyping lines 50

to 120, if you wish (leaving out the comments to the right of the assembler text). Here is the modified program:

```
 10 DIM P% 100
 20[
 30   LDA &80  Low byte of one number
 40   CLC           Clear carry flag
 50   ADC &82  Low byte of other number
 60   STA &84  low byte of result
 70   LDA &81  High byte of one number
 80   ADC &83  High byte of other number
 90   STA &85  High byte of result
100   RTS
110]
120 END
```

Assemble the program:

```
>RUN
0E6E
0E6E AS 80   LDA &80
0E70 18      CLC
0E71 65 82   ADC &82
0E73 85 84   STA &84
0E75 AS 81   LDA &81
0E77 65 83   ADC &83
0E79 85 85   STA &85
OE7B 60      RTS
```

Now set up the two numbers as follows:

```
?&81=&8C : ?&81=&00
?&82=&A0 : ?&83=&00
```

Finally, execute the program by typing

```
CALL TOP
```

and look at the result, printing it in hexadecimal this time for convenience:

```
PRINT ~?&84, ~?&85
```

The low byte of the result is &2C, as was obtained before using the one-byte addition program, but this

time the high byte of the result, &1, has been correctly obtained. The carry generated by the first addition was added into the second addition, giving

```
0 + 0 + carry = 1
```

Try some other two-byte additions using the new program.

2.7 Subtraction

The subtract instruction is just like the add instruction, except that there is a ' borrowif the carry flag is zero. Therefore to perform a single-byte subtraction the carry flag should first be set with the SEC instruction.

Mnemonic	Description	Symbol
SEC	set carry flag	C=1
SBC	subtract memory from A with carry	A=A-M-(1-C)

Example

```
      10 DIM P% 100
      20[
      30 LDA &80          Low byte of first number
      40 SEC                      Initialise carry flag
      50 SBC &82          Low byte of other number
      60 STA &84          Low byte of result
      70 LDA &81          Now do high bytes
      80 SBC &83
      90 STA &85
100   RTS                 Return
110]
120 END
```

Note that the above program is very similar in structure to the addition example in section 2.6.

2.8 Comments

There are two methods of putting comments in assembler programs. The first of these, which is used in previous examples, is to put the comment after an assembler instruction, separated from it by one or more spaces, e.g.

```
60    STA &84   Low byte of result
```

Alternatively a statement may start with a backslash (\), in which case the remainder of that statement is ignored, e.g.

```
65  \Now for the high bytes
```

Note that a colon (:) will end the comment and start a new assembler statement, for example line 60 could be replaced by

```
60  \Low byte of result : STA &84
```

2.9 Printing a character

The computer contains routines for the basic operations of printing a character to the VDU, and reading a character from the keyboard, and these routines can be called from assembler programs.

Name	Address	Function
OSWRCH	&FFEE	Puts character in accumulator to output (VDU)
OSRDCH	&FFE0	Reads from input (keyboard) into accumulator

In each case all the other registers are preserved. The names of these routines are acronyms for ' operating system write character' and ' operating system read character' respectively. These routines are executed with the instruction JSR (jump to subroutine).

A detailed description of how the JSR instruction works will be left until the following chapter.

The following program outputs the contents of memory location &80 as a character to the VDU, using a call to the subroutine OSWRCH:

```
10 DIM P% 100
20 oswrch=&FFEE
30[
40   LDA &80
50   JSR oswrch
60   RTS
70]
80 END
```

The variable ' oswrch'is used for the address of the OSWRCH routine. Assemble the program, and then set the contents of &80 to &21 by typing

```
?&80=&21
```

Then execute the program using

```
CALL TOP
```

and an exclamation mark will be printed out before returning to the computer' s prompt character, because &21 is the code for an exclamation mark. An alternative method of setting the contents of location &80 to &21 is therefore

```
?&80=ASC"!"
```

Try executing the program with different values in &80, with values chosen from the table of ASCII values in Appendix A.

2.10 Immdiate addressing

In the previous example the instruction

```
LDA &80
```

loaded the accumulator from the location whose address is &80, this is known as ' absolute' addressing. The location was then set to contain &21, the code for an exclamation mark. If at the time the program was written it was known that an exclamation mark was to be printed in would be more convenient to specify this in the program as the actual data to be loaded into the accumulator. Fortunately an ' immediate' addressing mode is provided which achieves just this. Change the instruction to

```
LDA #&21
```

where the ' #'(hash) symbol specifies to the assembler that immediate addressing is required. Assemble the program again, and note that the instruction op-code for ' LDA#&21' is &A9, not &A5 as it was previously for the absolute addressing. The op-code of the instruction specifies to the CPU whether the following byte is the actual data loaded, or the address of the location containing the data.

2.11 Using addresses

So far when a value has been saved, a numerical address has been used to define where it is to be stored, e.g.

```
STA &80
```

A better method of giving an address is to use a variable name, e.g.

```
STA addr
```

In this case ' addr' must be specified at the beginning of the program, e.g.

```
addr = &80
```

This method is better than the previous one since it makes the program easier to understand, i.e. an address can be given a relevant name, e.g. ' xlowbyte' or ' yhighbyte'In addition, changing the location of a value becomes easier, since only the initial specification need be altered rather than every occurence of that value throughout the program.

The locations used must be chosen carefully to avoid corrupting operating system or BASIC workspace. The memory map in Appendix A should help to show which locations can be used in different circumstances. Also there are some locations which are always free when using BASIC; these are &70 to &8F.