MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

Steve Kramer

MICRO PRESS

Machine Code for Beginners on the Amstrad

Machine Code for Beginners on the Amstrad

Steve Kramer

MICRO PRESS

First published in 1984 in the United Kingdom by
Micro Press
Castle House
27 London Road
Tunbridge Wells, Kent

Reprinted 1985 (twice)

ISBN 0 7447 0025 6

Printed and bound in Great Britain by
MACKAYS OF CHATHAM LTD

Contents

Acknowledgements

With thanks to Bob and Carol for their pertinent questions, and to Marcia for the cup of coffee.

Steve Kramer, 1984

Since this book was written before the disk drive and its associated interface were available, for some of the listings to operate correctly, it may be necessary for the disk interface to be either disconnected or the disk drive not to have been turned on.

Chapter One

Introduction

The Amstrad CPC 464 is perhaps the most exciting new computer to appear since the Sinclair Spectrum. It offers many advanced features from Basic which could previously only be accomplished by vastly more expensive machines, as well as a capacity for expansion at reasonable cost equal to that of almost any other home computer.

The real breakthrough for the programmer, however, is Amstrad's decision to document and release details of the operating system. This is a hitherto unprecedented addition to the documentation available for a home computer from the manufacturer, and it offers a real opportunity for the user to learn machine code programming the easy way, and get results almost immediately, by the use of calls to the operating system.

No longer is there the chicken-and-egg situation, that if you do not understand machine code you cannot use it, and if you cannot use it you cannot find out how to understand it on your computer, because you don't know how to make the computer respond.

This book is intended for the beginner wishing to learn how to use Machine Code on the Amstrad CPC 464. It will progress from the concepts of programming in machine code, explaining the instructions that the Z80 Central Processing Unit – or CPU for short (the silicone chip that does all the work in the computer) – understands, and how to use them, as well as introducing some of the routines in the operating system at various stages through the text.

Two complete machine code novices have assisted in the writing of this book, and their questions and problems form the basis for its structure. They have also helped to ensure that no information or detail which is so obvious to those in the know that it is almost second nature, and fundamental to being able to perform some function in programming, has been omitted. This is so often the failing which can leave a novice stranded, like directing someone to a place by telling them that it is on the corner of the High Street by Woolworth's. If they don't know which High Street or where the High Street is, it doesn't help much.

Short listings are given to help in entering machine code programs and to inspect and alter or move the contents of part of the memory. It is strongly suggested however, that you buy the Amstrad Assembler/Disassembler program. This will allow you to enter the code as mnemonics (a sort of shorthand for the names of the instructions the CPU understands) instead of by numbers. It will also allow editing and is much nearer to BASIC in the way that programs are entered.

Whilst it is obviously possible to sit down and read this book from cover to cover, machine code is such a potentially confusing subject, and so many new concepts are likely to be introduced, that it is suggested you sit down in front of your computer and enter and execute (preferably not by hanging) programs when they appear in a chapter. Only once you are sure that you follow what is happening should you progress.

Extensive use is made of the machine operating system, in order to allow results from programs to be seen immediately. The ability to do this is thanks to the Amstrad Firmware Specification (Soft 158) which, although it would probably be almost totally meaningless at this stage, will be a valuable addition to your library once you have finished this book and understand the concepts explained.

The Z80 CPU is one of the most widely used CPUs in the home computer market and, until very recently, was often the main CPU in many business machines. It offers access to the widest variety of software in the world through the medium of CP/M (short for Control Program for Microcomputers), and Amstrad are providing CP/M with their disk drives. The Z80 is also starting to appear as a second processor in business machines, as well as being available as an add-on for the BBC model B, the Commodore 64 and the Apple and its look-alikes. The skills that you will learn from this book are therefore likely to be of use if you have occasion to program other computers in machine code.

Chapter Two

What is Machine Code and Why Use it?

The CPU in your Amstrad computer is basically a very stupid creature, it runs all your BASIC programs very well and does its job excellently but it is stupid none the less. What makes it seem so clever is the firmware, the programs that are running all the time the computer is switched on. In its unmodified version the Amstrad is running an Operating System and the BASIC interpreter.

The operating system deals with such tasks as looking to see if any keys are being pressed, loading from the cassette or putting a character on the screen. You could imagine it as being in charge of all communications, and if it were not present you would have no way of knowing whether your computer was dead or alive, because you could not give it any information and it could not tell you anything.

The BASIC interpreter is literally that, it translates BASIC into the language the CPU understands. Imagine for a moment that someone tells you to turn to page 35. No problem, you just turn to page 35, but what if you were told 稯. Now you are in trouble, not only do you not know what to do, you probably don't even recognise the form of the instruction.

This is like putting yourself into the position of the CPU and giving it a BASIC command to execute. The CPU has no knowledge at all of BASIC, but it goes even further than that. The Chinese above uses one symbol to represent something that when transferred into English takes several symbols, it says Tsung. Still not much help is it even though you can read it? Translated it means 'to sow seed without first ploughing the ground'. This is similar to the difficulty the CPU would experience if asked to deal directly with BASIC. One BASIC instruction often represents many machine code instructions and worse, the characters used by BASIC cannot be understood by the CPU, which only recognises two states, 'on' and 'off'.

Fortunately the 'on's and 'off's are grouped into sets of eight, which gives 256 different combinations. It is these combinations that are used in machine code. You could think of them as the Chinese characters shown earlier.

The problems do not end there however; since one character represents a complete word and there are only 256 possible combinations, the CPU would seem to be limited to a vocabulary of only 256 words. This is nearly correct but, as in English, some words are made up of more than one smaller word.

Key board, for example, would be likely to conjure up a picture of a board in an hotel for hanging keys on, whereas a keyboard is what you will find on your computer. Here the two words have completely different meanings when together and when separate. Some words however can have their meaning subtly altered by the addition of a prefix, able, enable, and unable, or justice and injustice, for example. In each case the latter part has the same meaning but its direction is altered by the first part. The Z80 CPU has some word structures which use these types of construction. The problem of a limited vocabulary however remains.

This limitation does not constrain the concepts that it is possible to convey, but just means that more words are needed to say what you want to say in some cases. The start of this paragraph, for example, could have been written: This not having many words does not put an end to being able to put over all the ideas that can be put over when there are many words . . . same meaning but more words, rather repetitive and not very good English.

Machine code therefore tends to require a lot of simple words, or instructions to do the equivalent of one BASIC keyword, but there is no limit to the ways in which machine code instructions can be put together, and sometimes this means that machine code requires less instructions than BASIC.

With BASIC every time you run a program the interpreter checks each command and makes sure that it is valid, then it translates it into a series of machine code instructions which the CPU then executes, any results are then checked to make sure they are what was expected and saved for further use. All this takes time.

With machine code however there is no error checking, no time taken translating, and nothing is saved unless you tell the CPU to save it.

To demonstrate the time saved, enter the BASIC program below. First though, if your computer is already switched on, turn it off and back on again to make sure everything is 'virgin'.

```
10 MM = 43903

20 MEMORY 43799

30 FOR N = 43800 TO 43809 : READ D : POKE N,D : A = A + D: NEXT

40 IF A <> 1338 THEN CLS : PEN 3 : ? "DATA ERROR" : PEN 1 : EDIT
   90

50 INPUT "PRESS ENTER TO START";A : B = 255

60 ? "A";: B = B - 1 : IF B <> 0 THEN 60

70 ?

80 CALL 43800

90 DATA 6,255,62,65,205,90,187,16,251,201

100 END
```

Note that ? is used instead of PRINT to save time.

Once you have entered the program type RUN and press the enter key. If all is well you will be asked to "PRESS ENTER TO START", otherwise you will be presented with line 90 in 'edit' mode, because you have made a mistake in typing in the DATA.

When the enter key is pressed 255 'A's will be printed by the BASIC in line 60, hotly pursued by a further 255 'A's printed by the machine code routine you have 'POKED' into memory with line 30, and CALLed with line 80.

Whilst this is not a very exciting program it does show the speed of machine code.

If you count the number of characters used by the machine code routine (remembering that each item in the data statement is one machine code character) you will find that there are ten, of which the last, the 201, is only there to tell the program to return to BASIC. The BASIC program however uses thirty-seven characters if you include the spaces and not the line number. Even if it had been written without any unnecessary spaces it would have taken the equivalent of twenty-five machine code characters' worth of space.

You can check this for yourself if you wish by adding the following lines to your program:

```
110 PEN 3: FOR N = 520 TO 630 : A = PEEK (N)

120 ? A;: IF A = 32 THEN 150

130 IF A > 32 AND A < 129 AND B <> 1 THEN PEN 2: ?: ?N

140 IF A > 32 AND A < 129THEN PEN 1: ? CHR$ (A);: PEN 3: B = 0

150 NEXT

160 PEN 1

170 END
```

When run this will display the values held in the memory locations where the BASIC program is held in red. If there is a valid character represented by the number it will be displayed in yellow. You will be able to identify the start of line 60 by looking for the "PRESS ENTER TO START"; in line 50, following on until you come to 0 60 in red < in yellow and 0 in red. The 60 0 are the line number and the number before the first 0 is the number of characters in the line. The number in light blue at the start of each line on the screen is the number of the first memory location in the screen line.

The first thing you will notice is that the only characters that have been stored in the same way as you put them in are "A"; all the remainder have been reduced to a sort of code that the interpreter finds easier to handle. Every time you type LIST it is the interpreter that translates these numbers back into what you entered.

The upshot of all this is that the machine code program was not only quicker, but also more economical in terms of memory used, and these are the two main advantages of writing programs in machine code. In fact a program in BASIC can run up to about a hundred times slower than its machine code equivalent.

The main disadvantages however are that programs are almost totally incomprehensible and therefore difficult to debug, and that they tend to be long in terms of the number of instructions required, relative to BASIC or another high-level language.

The comprehensibility of machine code is greatly aided by the use of assembler and disassembler programs, and these are discussed in the next chapter, and while normally there is no way to overcome the problem of the large number of instructions required, with the Amstrad CPC 464, use can be made of subroutines in the operating system. Thanks to Amstrad's forethought in making the details available, you have already been able to do this if you entered the program earlier, and the major proportion of the instructions for most programs have already been written for you, by Locomotive Software, when they wrote the operating system.

Chapter Three

First Concepts

Before embarking on the world of machine code, there are some concepts that may be new to you and, as it is important that you have at least a rudimentary understanding of these, they will be briefly explained here.

Hex and Binary

Hex and Binary are different forms of counting, binary to base 2 and Hex to base 16. You have probably come across binary before at school and no doubt thought it was a pretty dumb way to count. For the computer however it is the only way, as you have no doubt realised by now. Due to the CPU only recognising the two states OFF and ON the only way it *can* count is with binary, ON corresponding to 1 and OFF to 0.

Each Binary Digit, or bit for short, has a fixed value according to its position. The decimal system uses the same convention. The right-most digit is the number of units, the next to the left, the number of tens, the next is hundreds and so on. In binary, since there can only be one or none, the values for the positions have to be tailored, so any number can be given. If the same position values were used as with decimal you would count one, ten, eleven, one hundred, one hundred and one and so on.

Your Amstrad computer stores its information in sets of 8 bits, called a byte, and it can also manipulate pairs of bytes (16 bits, known also as a Word) as if they were representing a single number, so the values assigned to each of 16 bit positions are shown below.

BIT NUMBER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

VALUE

With this combination it is possible to represent any number from 0 to 65535 by a combination of 0s and 1s. Note that the least significant bit is known as bit 0.

Sometimes it is wished to represent a negative value, and a convention has been assigned to this. If you start with 0, which is the same in any number base, and take 1 away you will get −1. Do this with a binary number and you will change every bit, for as far as you can go, to a 1, look at the example below which uses a 4 bit number.

```
0000

   -1   0 - 1 = 1 borrow 1

  -1    0 - 1 = 1 borrow 1

 -1     0 - 1 = 1 borrow 1

-1      0 - 1 = 1 borrow 1
------
1111
```

And the answer, because the subtraction is limited to 4 bits, is 1111 binary, or 15 decimal. The same will happen if you do the subtraction with 8 bits, or 16 bits, the decimal answer will be 255 and 32767 respectively.

If you take another number away instead of 1 the same borrow is made, and the left (most significant) bit will always be set (1) whenever the result is a negative number. It is this last fact which gives the clue as to how to represent negative numbers.

If negative numbers are to be used, or may result from a subtraction, it is the convention to use the most significant bit to indicate the sign of a number. Set (1) when the number is negative, and reset (0) for positive. This changes the range of numbers which can be represented by a given number of bits. 16 bits can now show −32768 to +32767, and 8 bits from −128 to +127. This technique for showing signed numbers in binary is called two's complement, if you complement (change 1s to 0s and *vice versa*) a binary number, and add 1, you will change the sign.

It will be up to you whether you use two's complement notation for numbers in programs, or normal unsigned binary, you can even use a mix of the two. The suggested representation for use with a particular instruction, will be shown used in the text of the remainder of the book, and where either method can be employed according to what end result is required, this will be mentioned.

The GENS assembler lets you use binary numbers in a program, if you prefix the binary number with a % sign.

But what of HEX? For the computer there is no problem thinking in terms of ONs and OFFs 0s and 1s but try counting yourself in binary or writing it down and checking it. You will find it unbelievably awkward! Most of the time decimal will be the easiest numbering system to use, but occasionally there will be times when it is easier to think in terms of binary. For example, when you are wanting to put a number into one byte in a special way. If you needed to have the number 9 (decimal) in each half of a byte it would mean reverting to binary to work out how. 1001 binary is 9 decimal, $(1*8+0*4+0*2+1*1=9)$ so you will need to have 1001 1001 to make each half of the byte holding 9 decimal. The decimal value for this is:

$$1*128+0*64+0*32+1*16+1*8+0*4+0*2+1*1$$

which equals 153 decimal. Convoluted, isn't it?

It is possible to have any value from 0 to 15 held in each half of a byte, giving 16 different values for each set of four bits. So, to make it easy to operate on binary numbers in bytes a new numbering system which uses base 16 is needed. If this was available you could have just said 99 instead of going through the rigours of discovering the decimal equivalent of what you wanted. This base 16 numbering system is called HEXadecimal, but as that is such a mouthful everybody just says HEX.

The first problem you will encounter is that whilst there are already numbers to count from 0 to 9, what do you do with 10 to 15. Rather than learn new symbols for these values, the first six capital letters of the alphabet are used. 10 decimal therefore becomes A, 11 becomes B and so on to 15 which is F. The other problem is that anybody else will think that you are using the decimal system, so some way of signalling that a number is to base 16 (a HEX number), is essential.

Unfortunately there is no set convention for this, your Amstrad uses the & sign to signify that the number following is HEX, the Firmware Specification Manual uses £ and the GENS assembler uses # and many other assemblers (probably including the Picturesque offering) use either a lower or upper case h following a number to show that a number is HEX. This is all downright confusing, but suffice it to say that if a number has anything except a number in its make-up, it is likely to be HEX.

In this book all HEX numbers are suffixed by a lower case h, except in listings from the GENS assembler, where they are prefixed by #.

ASCII

ASCII is short for the American Standard Code for Information Interchange. This is really just a glorified name for numbers representing letters, and operations. Appendix III of the Amstrad User Instructions gives the full list of ASCII codes.

Address

Address is the term used to describe a memory location. Each memory location has a unique address, starting from 0, for the first location and going up to 65535 (FFFFh). It is often given as a HEX number rather than a decimal number, and most assemblers give the address of a.. instruction in the first columns of the printout, when they assemble a program.

Assembler

Reference is made above to something called an assembler, but what is an assembler?

An assembler is a program which allows you to program machine code in a more recognisable form than numbers, in mnemonics. (Take note! This is a good word for crossword addicts and Scrabble fanatics alike.) Mnemonics are a sort of shorthand way of writing the description of what a machine code operation does, mnemonic means 'an aid to memory', and this is what they are, because you wouldn't have a snowball's chance in hell of remembering all the numeric forms of instructions (unless you are an Icelander, or from northern Scandinavia, they believe Hell is freezing cold and not hot). The assembler lets you write in this shorthand description form, and then when you have finished, it will translate (assemble) this into the 0s and 1s that the computer understands.

Most assemblers also have an integral Editor, to allow the program to be written and modified easily. Without this facility, if you had written a long program and you then found that the *n*th instruction was wrong, you would have to rewrite everything from there to the end again.

The program which you write, using the assembler, is called the source code, and this can be saved to tape for later editing if required, but is not needed to run a program once it has been assembled by the assembler. The actual program which can be run, or executed as it should really be called, is the object code.

This object code can also be saved to the tape, using the O command from the GENS assembler or by saving it from BASIC. When saving from BASIC the form of the command is:

```
SAVE "filename",B,start address,length,entry point
```

The entry point is the address at which execution is to start, if the program is loaded by the RUN" command, and if not specified the Amstrad will do a system reset when the program is loaded by RUN".

An assembler will allow you to use what are known as 'labels' instead of addresses when you are writing a machine code program. This is an incredibly helpful, and almost Pascal-like facility. (Pascal is a high-level language like BASIC but designed to be assembled like assembly language, the machine code that is created will not run anywhere near as fast as that created by assembly language, and it will also take up more room, but is still a lot faster than BASIC.)

With Pascal instead of using GOSUB followed by a line number, you give a name to a subroutine, and just place the name in the program, when this name is met, the routine associated with its name is executed. The assembler allows labels (short names ended with a colon [:]) to be placed in the listing beside an instruction, and when that label is referred to the address of the instruction which it is beside, will be used instead. This is like being able to give a subroutine a name, and thereafter you would no longer need to know the line number where it started, but could just write GOSUB and the name of the routine.

The assembler also allows *Pseudo Operations*, or *Pseudo Mnemonics* as they are sometimes called. (Quite why one cannot guess, because they are real mnemonics, but just don't become machine code when assembled.) These are used to tell the assembler to do something with the number which follows, and the main ones are:

Which tells the assembler that everything following it is a comment, and should be ignored. Just like a REM in BASIC.

EQU EQUate, or EQUals. This allows you to use a label to represent any number you chose. The label to be EQUated should be put to the left of the EQU Pseudo Op, terminated with a colon as always, and the number you wish the label to equate to should be put to the right. For example, LABEL: EQU #1234 will make the label LABEL represent 1234h (4660 decimal) whenever it is used.

DEFB DEFine Byte. The byte at the address will be made to hold the value which follows. For example, DEFB #20 will make the byte at the address of the DEFB mnemonic hold 20h (HEX) when the program is assembled.

DEFW DEFine Word. This is the same as the mnemonic above, but it will put a 16 bit number into two memory bytes. The byte of the instruction, and the following byte.

DEFM DEFine Message. This allows letters to be placed after the mnemonic, between inverted commas, which will have their ASCII codes put into successive locations when assembled.

DEFS DEFine Space. The number of memory locations given by the number following the DEFS mnemonic will be skipped by the assembler, when assembling the program.

ORG ORiGinate. The number following the ORG mnemonic will be the address of the next instruction when the program is assembled.

ENT ENTry. The address at which execution should start, in response to the assembler's J command.

The ORG mnemonic in a listing will give the start address for a section of program, required by the HEX Loader program, and the ENT mnemonic will be followed by the address which should be CALLed from BASIC to run a machine code program.

An OPCODE is a machine instruction which tells the computer what to do and this is sometimes followed by an OPERAND which gives the information upon which the instruction is to be executed.

Assembler Listings

Listings of machine code programs normally have five columns, sometimes six if comments (preceded by a ; remember?) are used. The first column gives the address of the start of the instruction, normally in HEX.

The second gives the HEX version of the machine code instruction, and it is this that should be entered if you are using a HEX Loader, such as the one given in the Appendix, in pairs of numbers.

The third is a line number, and of no use except when writing the program.

The fourth is occupied by any labels, next to the instruction which occupies the address referred to by the label, but the colon which must terminate the label is not shown. This must be entered if you are copying a program from a listing.

The fifth is the mnemonic form of the instruction, as entered by the programmer, and what you enter if you are using an assembler, after any labels in column four that are on the same line.

The sixth column may be taken up by a comment.

Armed with this information you should be ready to proceed!

Chapter Four

Flow Charts

A flow chart is often used in the program design and development stage; this is simply a symbolic representation of the flow of the program under development. There is a standard set of symbols used for drawing flow charts and those that you are most likely to employ are shown in Fig. 4.1 with their uses.

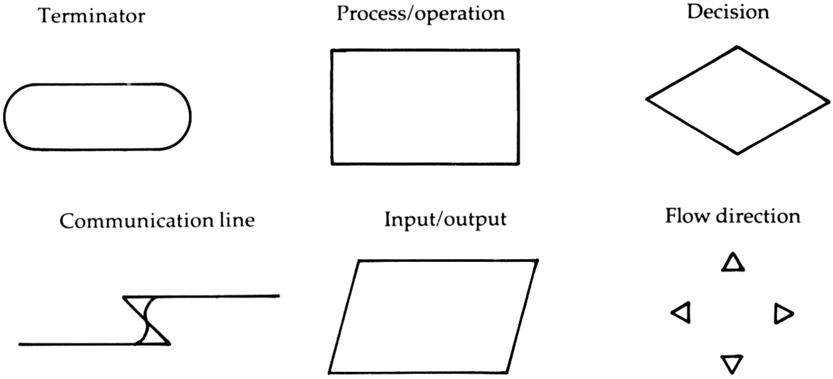


Figure 4.1

There are a number of further symbols, but they are not often used. The purpose of a flow chart is to make clear the processes that are being carried out by a program. Consider a very simple example, a flow chart to load a program from tape into your Amstrad computer.

This covers the essential operations, without going into too much detail, as a flow chart should. There are many applications where the flow chart is almost essential for analysing the actions carried out, or necessary to be carried out, by a program. Often they will enable you to find faults before they occur, as a glance at a flow chart will let you see the overall principles of the program under investigation.

Look at the example in Fig. 4.3, which shows the difference between a BASIC WHILE loop and a BASIC FOR NEXT loop. It should be immediately apparent what the main difference is.

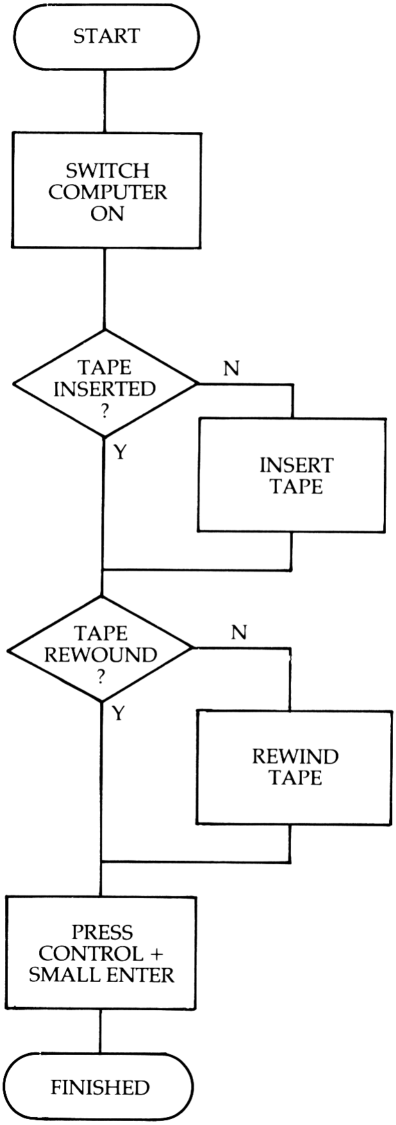
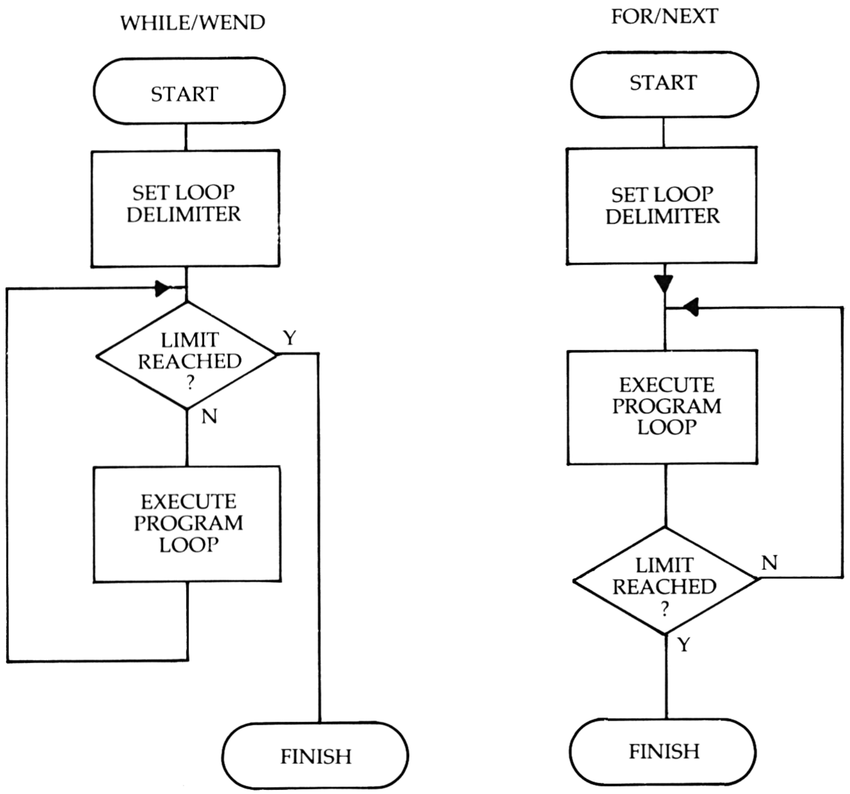


Figure 4.2


WHILE/WEND

FOR/NEXT

Figure 4.3

Chapter Five

Simple Machine Code Instructions

LD CALL RET JP JR

The CPU has fourteen registers, each of which can be thought of as being similar to a BASIC integer variable. They are shown below with their functions. Don't worry if this makes no sense yet, all is about to be revealed.

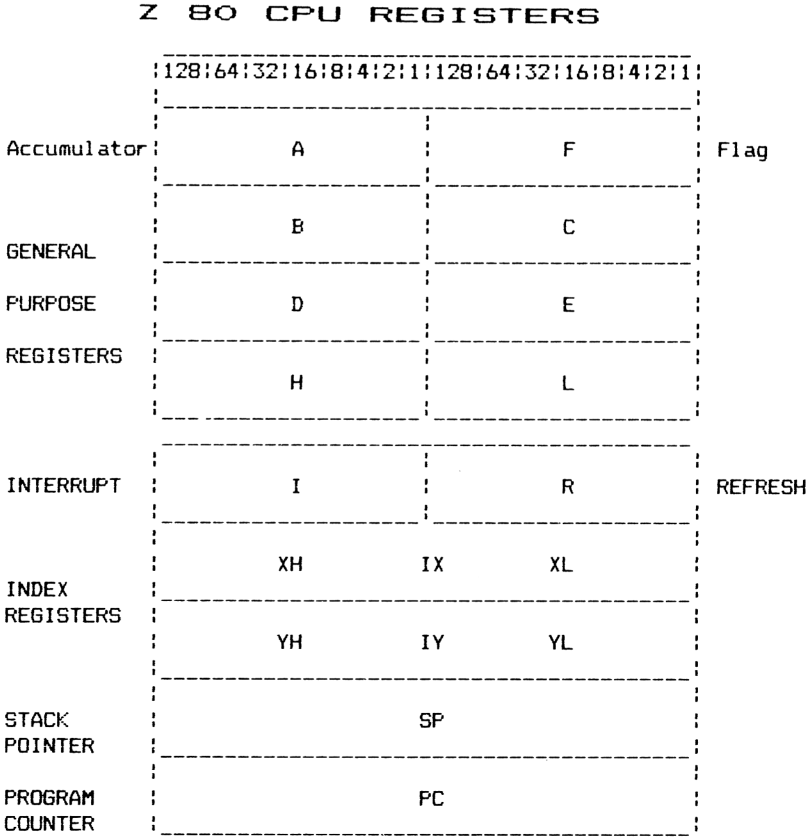
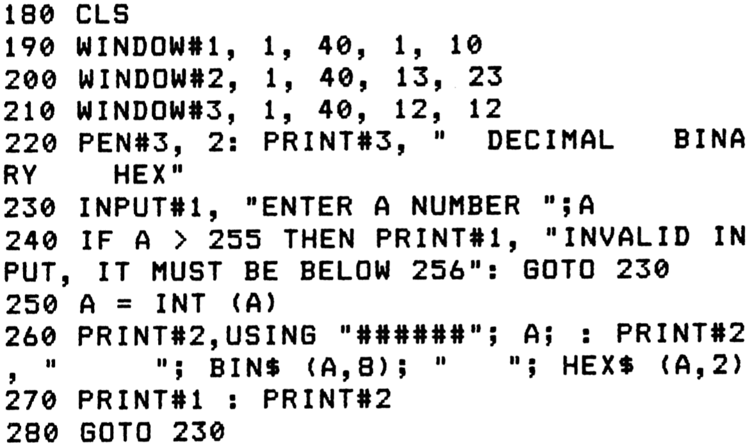


Figure 5.1

There are six general-purpose registers, and it is these that will be considered in this chapter, along with the special-purpose 'A' or accumulator register and the PC or program counter register.

Each of the general purpose registers B, C, D, E, H and L has the capability of holding a number between 0 and 255, each being made up of eight bits, and they can each be loaded in three basic ways. To continue the analogy to a BASIC variable and help explain each of the ways that a register can be loaded enter the short BASIC program shown in Fig. 5.2. There is no need to delete the first program if it is still there.



```
180 CLS
190 WINDOW#1, 1, 40, 1, 10
200 WINDOW#2, 1, 40, 13, 23
210 WINDOW#3, 1, 40, 12, 12
220 PEN#3, 2: PRINT#3, "  DECIMAL   BINA
RY    HEX"
230 INPUT#1, "ENTER A NUMBER ";A
240 IF A > 255 THEN PRINT#1, "INVALID IN
PUT, IT MUST BE BELOW 256": GOTO 230
250 A = INT (A)
260 PRINT#2,USING "######"; A; : PRINT#2
, "      "; BIN$ (A,8); "    "; HEX$ (A,2)
270 PRINT#1 : PRINT#2
280 GOTO 230
```

Figure 5.2

When you have typed the program in run it by typing RUN 180, and you will be asked to enter a number. The basic variable 'A' in the program represents the 'A' register in the CPU. When you enter a number if it is between 0 and 255 it will be printed in Decimal, as you entered it; Binary as the computer handles it, and in Hex.

Each number you enter is loaded into BASIC variable A, and this is then used to provide the number for other duties in the program. If you were to type in 77 you would load A,77 when you pressed enter.

Surprisingly the machine code instruction for this would be spoken of as Load A,77. Easy, isn't it!

Unfortunately, however, the CPU does not understand this, what it requires is 00111110 followed by 01001101. Or 3Eh followed by 4Dh, 62 and 77 decimal. Now things are getting difficult again and this is where the assembler program comes in. You can tell the assembler that the instruction is LD A,77. When the assembler assembles the instruction it will do the translation for you. Note that load has been shortened to LD, which saves your fingers when typing machine code programs and is the standard Z80 assembler convention.

If you now turn back to the program in Chapter 2 and look at line 90, you will see that the third DATA item was 62. This will give you a clue about one of the instructions in the program, and if you were to look up the ASCII code for a capital A (of which an awful lot were printed by the program) you would find that it was 65. Things are probably beginning to dawn about now, and when you find out that the code for instruction to load the B register with a number is 00000110 in binary or 6 in decimal (and Hex for that matter), and you remember that 255 As were printed you will very likely be deafened by the morning chorus.

Who said machine code was difficult? The first two instructions of the program, in assembler, are therefore:

```
LD B,255
LD A,65
```

Knowing this you can now change the number of times the character is printed, and which character is printed. Turn first to Appendix III in the Amstrad User Instruction book that came with your computer. Page 1 will show you the codes for each of the ASCII characters and the following pages the characters created by all the codes from 32 up to 255.

Now remove line 40, which checked that you had entered the DATA correctly when you wrote the program. The checksum will be wrong since you are going to change items in the DATA line.

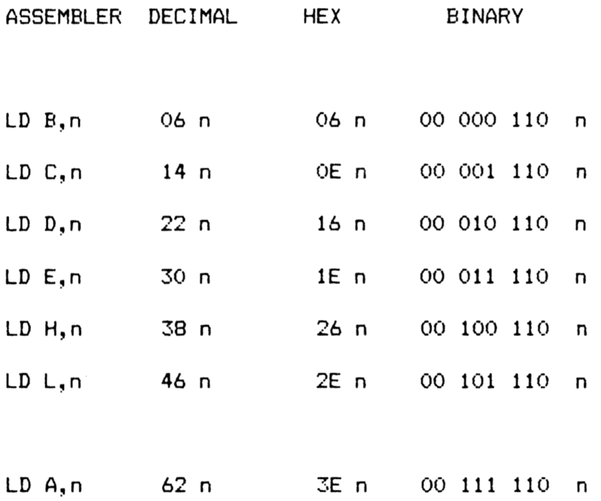
All you need to do now is change the 255 for the number of times the character is to be printed, the 65 for the code of the character you want printed, and then RUN the program from the start to see the result. Don't use any values below 32 for the code, or you may get peculiar results.

If you replace the 255 with 0 you will find that the character in the A register is printed 256 times. Remembering that each register can only hold 8 bits and looking at the BASIC in line 40, can you work out why this should be?

Think of the sequence of operations, A = 0 or 00000000b. 0 − 1 = −1 but 00000000b − 00000001b = 11111111b which is 255 in binary.

You can check this by asking your computer. Type in ? BIN$ (−1) and you will get the answer 1111111111111111. As the left-most eight digits are all that a single register or memory location can hold, this means that −1 decimal is equal to 255 when transferred through an 8 bit register. Confused? Then turn back to Chapter 3 in this book and Appendix II page 2 on, in the Amstrad Computer User's Instructions.

Each of the general purpose registers can be loaded in the same way as the A and B registers. The code for each is as follows:



ASSEMBLER	DECIMAL	HEX	BINARY
LD B,n	06 n	06 n	00 000 110 n
LD C,n	14 n	0E n	00 001 110 n
LD D,n	22 n	16 n	00 010 110 n
LD E,n	30 n	1E n	00 011 110 n
LD H,n	38 n	26 n	00 100 110 n
LD L,n	46 n	2E n	00 101 110 n
LD A,n	62 n	3E n	00 111 110 n

Figure 5.3

In each case the n represents any number, between 0 and 255 decimal (FFh or 11111111b) which is to be loaded into the chosen register.

If you look closely at the binary codes for each instruction there are two things that you may notice.

First, the two ends of the instruction are the same in all cases. It is these two sections that tell the CPU that it is a load instruction involving a number being put into a register.

Second, the register is determined by bits 5, 4 and 3, and one possible combination is missing. The three bits which decide which register is to be used are always the same for each register. Whenever an instruction can be performed on any of the general purpose registers three bits are used to tell the CPU which register is to be used.

B is always 000
C is always 001
D is always 010
E is always 011
H is always 100
L is always 101
A is always 111

Figure 5.4

The missing combination, 110, is used for a special purpose and this will be explained later in this chapter.

As well as being able to load a register direct with a number from the following location in memory it is possible to load a register with the contents of another register or from memory.

Think of the BASIC statement A = B. What this is doing is telling the computer that you want the variable 'A' to be equal to the variable 'B'.

If you enter the following lines into your computer and execute them by typing RUN 300 you will find that after line 320 B was loaded with the same value as A, but A was not changed.

```
300 B = 10

310 ? "BEFORE :A=";A;" B=";B

320 A = B

330 ? "AFTER  :A=";A;" B=";B
```

Figure 5.5

Having learnt that the machine code mnemonic equivalent to line 300 is LD B,10 what do you think is the equivalent to line 320?

It's pretty obvious isn't it? LD A,B and the same applies to all the other registers.

The actual instructions are made up in the same way as that for loading a register with a number except that bits 7 and 6 are changed from 00 to 01, and the bottom three bits, instead of being 110 are used to identify the register from which the value is to be taken.

```
The instruction is therefore;

ASSEMBLER  DECIMAL   HEX     BINARY

LD A,B       120      78   01 111 000
```

If you remember the three bits for each of the general purpose registers or if you look back at them you should now be able to give the binary instruction for loading any register from any register.

Bits 7 and 6 will always be 01 bits 5, 4 and 3 will be the register to be loaded and bits 0, 1 and 2 the register to be loaded from.

LD H,A would therefore be 01 100 111. What would LD A,H be? Or LD B,D?

Now you have two ways in which you can put numbers into registers and you are no doubt realising that the instructions are really quite logical in the way that they are made up, and hence not too difficult to follow.

All of the general purpose registers have special things that they can do and these will be introduced at various stages in this book. Unfortunately though, unlike BASIC variables, their limitations as to use are fixed, not decided by the user, and are much more subtle, or rather devastating in their effect.

Don't worry, the resemblance between the registers in the CPU, and BASIC variables which has been emphasised, still holds true, but whereas on the Amstrad, when you turn the computer on, any variable can be used for any purpose, be it string, integer or real, and each numeric variable could be used in place of any other numeric variable, in Z80 machine code there are things which can only be done with particular registers.

This is rather like the effect of adding the following line to the program in Chapter 2: 21 DEFSTR A. Now when you RUN the program you will get a type mismatch error when line 30 tries to add numeric information to a variable that can only be a string. It has already been explained that a general purpose register can only hold an 8 bit value but, apart from this limitation, any can be used to represent a number, as the B register in the program in Chapter 2, or a letter, as the A register in the same program.

Were you to make the statement 8 = 9 it would be utter balderdash, we all know that 8 = 8 and 9 = 9. If you type into your computer 8 = 9 followed by enter you do not get an error message, but only because the computer has interpreted the first 8 as a line number, and if you list the program you will find line number 8. If you change this to 8 8 = 9 and try to execute it, the computer will point it out as a syntax error, the same applies if you try to make a number equal to a variable, for instance by the statement 8 = HL.

No matter what you do (short of redefining the key), typing ? 8 will always give the answer 8. But change this to ? PEEK (8) and you will get the answer 195. This is because by adding the PEEK function you are now asking the computer 'what is the contents of the memory location at address 8?' instead of just 'what is 8?'.

When writing machine code there is no need for the PEEK function, or the corresponding POKE command since you are already at machine level. But it is still necessary to be able to access memory locations.

The A register is explained in the next chapter in its role as the accumulator, but has some other special instructions which are relevant to this chapter. This is due to it being the only 8 bit register that can be loaded direct from a memory location, the equivalent to the BASIC:

```
A = PEEK (nn)
```

where nn is any 16 bit number.

At first thought the machine code instruction to load the 'A' register with the contents of the memory location at address 8 might be LD A,8, but you will immediately see that this would put the value 8 into A, so a way of differentiating is required.

Each memory location can be thought of as a box, divided into eight smaller boxes, and the PEEK function in BASIC goes some way towards reinforcing this visualisation, by requiring brackets – which look a little like a box – round the number of the box which is to be inspected. (All right, very little like a box!)

You have probably already worked it out, and don't need to be told, but just to confirm how clever you are; the machine code mnemonic for 'with the contents of' requires brackets round whatever is representing the address of the memory location. Hence the instruction to load register 'A' with the contents of memory at address 8 is LD A,(8) and to load into memory at address 40000 the contents of the 'A' register the instruction is LD (40000),A.

If you don't have an assembler things become a little more complicated, but not much.

ASSEMBLER	DECIMAL	HEX	BINARY
LD A,(nn)	58 n n	3A n n	00 111 010 n n
LD (nn),A	50 n n	32 n n	00 110 010 n n

It is vital to remember that the 2 'n's in each of the above instructions occupy a memory location and they are calculated by the formula:

n1 = number MOD(256) and n2 = INT (number/256)

This is because of the internal operation of the CPU and there is no alternative but to have the two bytes with the low byte before the high byte. The opposite way to which you would expect. All 16 bit numbers are stored in memory in this manner, be they as part of an instruction or just data in memory put there by the CPU.

It should have been possible to use the above equation directly on your computer to save having to work n1 and n2 for each new number, but due to considerable inconsistencies in Amstrad's BASIC, which sometimes uses 2s complement notation and sometimes normal integer representation, the MOD function is useless with values over 32767.

The following BASIC line however will do the job for you, and you can add it to the program already in memory.

```
1010 N2= INT (NUMBER/256): N1 = NUMBER - N2*256: ? "N1  =";N1;"
   N2 =";N2
```

If you now type

NUMBER = 40000: GOTO 1010 followed by [ENTER]

you should get the answer N1 = 64 N2 = 156. So the full instruction for each of the latest opcodes, with the address to be loaded from and to as 40000 in each case, will be:

```
         ASSEMBLER          DECIMAL     HEX

        LD A,(40000)     58 64 156   3A 40 9C

        LD (40000),A     50 64 156   32 40 9C


                         BINARY

LD A,(40000)     00 111 010   0100 0000   1001 1100

LD (40000),A     00 110 010   0100 0000   1001 1100
```

```
or for address 8;
```

```
         ASSEMBLER      DECIMAL       HEX

         LD A,(8)       58 8 0       3A 08 00

         LD (8),A       50 8 0       32 08 00


                         BINARY

   LD A,(8)      00 111 010   0000 1000   0000 0000

   LD (8),A      00 110 010   0000 1000   0000 0000
```

You can test this out for yourself if you wish by changing the program which printed the 'A's.

Line 60 has to be changed to:

```
60 ? CHR$ (PEEK (8));: A = A - 1: IF A <> 0 THEN 60
```

And the second machine code instruction in the DATA statement in line 90 must be changed from LD A,65 to LD A,(8).

This will make it read:

```
90 DATA 6,255,58,8,0,205,90,187,16,251,201
```

The checksum in line 40, if it is still there, must be changed to 1277.

Most important line 30 must be changed to:

```
40 FOR N = 43880 TO 43890: READ D: POKE N,D: A = A + D: NEXT
```

This is because there is now one extra byte of code that needs to be POKEd into memory.

If you added line 21 earlier (DEFSTR A) remember to remove it!

Now when you RUN the program, instead of getting 'A's you should get\ s (backslashes).

Perhaps the next most useful thing about the general purpose registers, is that each pair can be used together, that is as BC, DE and HL. When used this way they can be treated as 16 bit registers.

Whereas with a single register you are limited to numbers that can be represented in eight bits, in other words between 0 and 255, with a register pair you can use any whole number from 0 to 65535, because you now have sixteen bits to play with. There are however penalties to be paid for the ability to use registers in pairs as if they were just one 16 bit register.

You have seen that, when used on their own, ALL general purpose as well as the 'A' or accumulator registers can:

1) be loaded from any other general purpose register;
2) have a number loaded directly into them;

but that the 'A' register alone can be loaded directly from or to, a numbered memory location, as in the LD A,(nn) or the LD (nn),A instructions.

With register pairs however, there is no machine code instruction for LD rr,rr' (load one register pair with the contents of another register pair). You can however load a number directly into any register pair.

For those of you with an assembler, nothing could be simpler. You probably don't even need to be told what the instruction is! It's LD rr,nn; rr is any of the register pairs, BC, DE or HL and nn is any 16 bit integer.

The instruction to put the value 40000 into the BC register pair would therefore be:

LD BC,40000

or to put the value 8 into the HL register pair:

LD HL,8

If you remember the construction of the binary instructions to load a number into a single register you almost certainly can guess the first two bits of the binary instruction to load a register pair. If you don't remember then turn back and have a look.

Right first time (I hope)! The first two bits are 00.

The remainder of the instruction is made up in much the same way. After the initial 00 the next two bits are used to determine which register pair is to be loaded.

00 for the BC register pair.
01 for the DE register pair.
10 for the HL register pair.

(These two bit codes are always the same for each register pair, and are used whenever an instruction can be performed on any of the register pairs.)

The next bit is 0 and the last three bits are 001.

The full instruction for each register pair is therefore:

ASSEMBLER	DECIMAL	HEX	BINARY
LD BC,nn	1 n n	01 n n	00 000 001 n n
LD DE,nn	17 n n	11 n n	00 010 001 n n
LD HL,nn	33 n n	21 n n	00 100 001 n n

n1 and n2 are calculated in the same way as for the LD A,(nn) instructions. Therefore:

ASSEMBLER	DECIMAL	HEX	BINARY
LD BC,40000	1 64 156	01 40 9C	00 000 001 0100 0000 1001 1100
LD HL,8	33 8 0	21 08 00	00 100 001 0000 1000 0000 0000

Knowing how to load a register pair with a sixteen bit number is a waste of time if there is no use for the register pair once loaded. One of the most common uses of a register pair is as a variable to point to a memory location.

In the BASIC earlier PEEK (8) was used to find the contents of memory location 8, and the machine code equivalent using the 'A' register was explained. This type of instruction is very limiting, especially if a series of locations need to be read from or written to. In BASIC the way to handle the problem would be to use a variable. For example if the variable HL was made equal to 8 it would have been possible to use PEEK (HL) instead.

In machine code the same applies, but this is where the idiosyncrasies of the Z80 CPU start to gain prevalence.

With a general purpose register, not only is it impossible to use the LD r,(nn) or the LD (nn),r constructions, but only the HL register pair can be used as a pointer. You have seen the elusive 110b code, missing from the series of three bit values used to represent the general purpose registers, used at the end of a LD r,n instruction code starting with 00b. Here it indicates that the next byte (the one after the instruction) is to be used as a number.

Used in the middle of the LD r,n instruction or in the LD r,r′ instructions, beginning with 01b, it has a different interpretation. It would be impossible for the 110b to have the same meaning since, as has already been demonstrated earlier, a number always has the same value. All that happened when it was tried to change the value from BASIC was a syntax error. But there is a way in which a number can be changed, and that is when it is used as an address.

Wherever the 110b code is used in a load instruction in place of a code representing a register, it is taken by the CPU to refer to the memory location whose address is held in the HL register pair.

Therefore to load the D register with whatever is in memory location 40000 you would write:

```
ASSEMBLY      DECIMAL     HEX                   BINARY

LD HL,40000   33 64 156   21 40 9C   00 100 001 0100 0000 1001 1100

LD D,(HL)     86          56         01 010 110
```

or to load the contents of memory at address 8 into the B register:

```
ASSEMBLY    DECIMAL     HEX                     BINARY

LD HL,8     33 8 0    21 08 00   00 100 001   0000 1000   0000 0000

LD B,(HL)   70        46         01 000 110
```

Note the brackets round the HL in the assembly language instruction, meaning 'with the contents of the address at'.

It is perfectly in order to reverse this process and, instead of loading a register with the contents of a memory location, load a memory location with the contents of a register. The instruction then becomes:

```
LD (HL),r
```

As with the LD r,(HL) any general purpose register or the 'A' register can be used, and the binary instruction opcode is about as predictable as it could be.

To load a register with the contents of the memory location at address HL the opcode is:

[01] [the three bit code for the register] [110]

so, to load the memory location at address HL with the contents of a register it becomes:

[01] [110] [the three bit code for the register]

You didn't really need this book to tell you that, did you?

If you change the DATA statement in line 90 of your BASIC program to:

```
90 DATA 33,8,0,70,58,8,0,205,90,187,16,251,201
```

change the checksum in line 40 to 1127 and change the number after the TO in line 30 to 43892 you can now see the LD B,(HL) and the LD HL,nn in action. The BASIC equivalent would be to change the end of line 50 from B = 255 to HL = 8 : B = PEEK (HL).

When run the machine code routine will now load the HL register pair with 8 and then load B with whatever is in location HL. The start of the routine is now:

ASSEMBLY	DECIMAL
LD HL,8	33 8 0
LD B,(HL)	70
LD A,(8)	58 8 0

When the 110b code is used in the LD r,n opcode for the r part, giving the binary opcode 01 110 110, this gives the assembly language instruction: LD (HL),n which will put into memory at address HL the number in the location following the instruction.

With the 'A' register it is possible to use any register pair as a variable, or pointer. The assembly language instructions are pretty obvious. They are LD A,(rr) or LD (rr),A where rr is any register pair. For example using the 'A' register it is quite acceptable to write:

```
LD DE,8
LD A,(DE)
```

The opcode for LD A,(HL) has already been explained and you have no doubt noticed that all possible combinations of instruction starting with 01 are allocated. It is necessary to use a different construction for the instructions LD A,(BC) LD A,(DE) LD (BC),A and LD (DE),A.

One clue to how these instructions are made up is to be found in the LD A,(nn) and LD (nn),A opcodes, and another is in the codes for the register pairs.

Have you noticed that the 3 bit codes for the general purpose registers share their high two bits with the 2 bit codes for the register pairs that use them. B has the code 000 and C 001, BC the code 00. D has the code 010 and E 011, DE is 01, H is 100 and L is 101, making HL 10.

The opcode for LD A,(nn) in binary is 00 111 010 and it has been pointed out that, in the instructions to load a number into a register pair, bits 5 and 4 tell the CPU which register pair to use. The only opcode missing from the set is 11. Lo and behold! What do we have in bits 5 and 4 of the LD A,(nn)? And what *do* you suppose the instruction LD A,(BC) will be, in binary?

Well done, but it didn't really require you to be a genius, did it? The binary opcode for LD A,(BC) is 00 001 010, for LD A,(DE) it would be 00 011 010.

The opcode for LD A,(HL) is *not* 00 101 010, it is 01 111 110 and was explained earlier, so what do you suppose happens in response to 00 101 010? Stay tuned and all will be revealed. First though there is the question of the LD (rr),A opcodes to be resolved, and be warned! These are almost as complicated as the LD A,(rr) opcodes.

LD (nn),A in binary is 00 110 010, again the missing 11b from the set of codes for register pairs is present in bits 5 and 4. Here, as in the previous instructions, to change (nn) to (BC) or (DE) all you do is alter the 11b to 00b or 01b. The opcode for LD (BC),A is therefore 00 000 010 and for LD (DE),A 00 010 010. But as with the previous instructions the HL code 10b used to make the opcode 00 100 010 doesn't mean LD (HL),A.

All the above opcodes, which load the 'A' register from or to a memory location pointed at by a register pair, are single byte instructions. The codes 00 100 010 and 00 101 010, which use the HL code 10, are made up of three bytes. The first is, of course, the opcode. Either 00 100 010 or 00 101 010, and the next two are the operand. (Remember, the operand is information needed for the opcode to be able to perform its task.) These two opcodes are used to load the HL register pair either to, or from, the memory location addressed in the next two bytes. They work in the same manner as the LD (nn),A which loads the A register to, and LD A,(nn) instruction which loads the A register from, a memory location addressed by the (nn).

Here are the assembly language and binary instructions in full.

```
ASSEMBLER         BINARY

LD HL,(nn)     00 101 010 n n

LD (nn),HL     00 100 010 n n
```

Assume that the address (nn) is 8, as has been used in the previous load instructions. This will make n1 0000 1000 and n2 0000 0000 in both the binary examples above and the assembler will become:

LD HL,(8) and LD (8),HL

In the first case the HL register pair will be loaded with a sixteen bit number from memory at the specified address, and in the second the sixteen bit number in HL will be loaded into memory at the specified address.

One problem though, only one memory address has been defined by the operand, and a single memory location is only eight bits, so how can a 16 bit number be condensed into eight bits? The short answer is, it can't. The way the CPU copes with

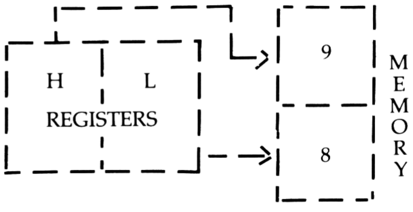


Figure 5.6

this is to use the memory location specified, for the low byte, and the memory location one above, for the high byte. Being logical, because logic is what computers are all about, the high byte comes from the H register and the low byte from the L register, and the low byte is in the lower memory location.

The CPU always starts at the bottom and works up and this is why, when you are loading machine code into memory, all 16 bit numbers are reversed. If you think of HL as high low, and remember to start from the bottom and work up it may help you to avoid any mistakes when writing code. When using an assembler it does all the reversals for you so, if you are using one, there is no need to change numbers from their normal form.

The last load instructions to be explained in this chapter are those which work in the same manner as above but for the BC and DE register pairs. These are less used than the instructions using the HL register pair because they use two bytes of memory to hold the instruction. The assembly language for these instructions is just what you would expect:

LD BC,(nn) LD DE,(nn) LD (nn),BC and LD (nn),DE

If you cast your mind back to when you were reading Chapter 2, you may recall that, in the analogy to English and Chinese language words, it was shown how the addition of an extra word couľd change the meaning of a word. Well, the opcodes for the four instructions above all have the prefix (in Hex) ED. (1110 1101b or 237 decimal but ED is much easier to remember.) All the comments regarding the instructions just explained, using the HL register pair, also apply to these instructions. It is considered good practice to use instructions which use the HL register pair, where possible, as these take half the amount of memory for the opcode. The actual opcodes are made up as follows:

ASSEMBLER	DECIMAL	HEX	BINARY
LD BC,(nn)	237 75 n n	ED 4B n n	1110 1101 01 001 011 n n
LD DE,(nn)	237 91 n n	ED 5B n n	1110 1101 01 011 011 n n
LD (nn),BC	237 67 n n	ED 43 n n	1110 1101 01 000 011 n n
LD (nn),DE	237 83 n n	ED 53 n n	1110 1101 01 010 011 n n

Figure 5.7

There is a summary of all the instructions starting with LD at the end of this chapter, and a graphic representation is given in Zilog's summary in the appendix.

The Program Counter

All the time your computer is turned on, unless someone has stopped the CPU for some reason, the program counter register is chuntering away quite happily, with only one aim in life, which is to get to the top and start again. Its purpose is to keep track of the program being run, and it will always hold the address in memory of the program instruction which is currently being executed. When you first turn on the computer the PC is forced to hold address 0, so that the first instruction is fetched from here. This means that a program to set the computer into a known state can be automatically executed at turn-on. This is known as a Cold Start or Early Morning or Wake Up, and on the unexpanded Amstrad this puts the computer into the BASIC programming mode and displays the Amstrad and Locomotive copyright.

As already mentioned, the computer is always running a program whenever it is switched on but, since the computer does not know what it is going to be asked to do next, it is essential to have some control over the program counter. Imagine what it would be like if the computer's memory were a piano keyboard, and all you could do was play each note in turn, from the bottom of the keyboard to the top, and then start over again. Some interest could be added by tuning the strings in the piano to play a short tune, but this would soon become boring, and all the strings would need to be retuned every time a new melody was required. This is similar to what would happen if the computer's program counter could not be altered. The way the strings were tuned would be the program, and each key a memory location.

Fortunately it is possible to alter the PC, and in BASIC this is achieved by the GOTO and GOSUB and RETURN commands. The GOTO forces the execution of the program to jump to the line nominated, and the GOSUB calls a subroutine at the line chosen. When the subroutine has completed its task, control is returned (by the RETURN command) to the main program, which carries on at the instruction after the GOSUB.

The machine code equivalents to these BASIC commands perform exactly the same tasks but their names are slightly different, as with the assembler LD mnemonic, which describes the action performed by the equivalent BASIC command. (LD is spoken as 'load', which you undoubtedly remembered.)

Can you guess what the machine code instructions are? GOTO becomes JUMP and GOSUB becomes CALL, RETURN stays the same. The return instruction is abbreviated to RET when using an assembler but CALL is written in full. Jump is a little more complicated so this will be explained later in the chapter, after the use of CALL and RET has been defined.

The CALL and RET instructions exactly duplicate their BASIC equivalents but, since machine code does not have line numbers, the CALL is made to the address holding the start of the first instruction of the subroutine.

The CALL instruction is made up of three bytes, the first is the opcode, and the next two are the address of the subroutine to be called. The two bytes holding the address are made up in normal Z80 manner, low byte first. If you are using an assembler it will do the calculation for you, as it did for the LD instructions.

The CALL and RET opcodes are as follows:

ASSEMBLY	DECIMAL	HEX	BINARY
CALL nn	205 n n	CD n n	11 001 101 n n
RET	201	C9	11 001 001

If you look back to Chapter 2 and look at the program you typed in, or if you are in front of your computer and have entered the programs so far, list line 90. You will find that the numbers immediately after those that you have experimented with, are 205, 90, 187.

You should now know what these are telling the CPU to do. The 205 is a CALL instruction and the address CALLed can be worked out by adding the next number to 256 * the third.

187 * 256 = 47872. 47872 + 90 = 47962 or BB5Ah

So now you know that the start of the routine you have written first loads the A register with the code of the character to be printed, then loads the B register with the number of times the character is to be printed, and then CALLs a subroutine starting at address 47872 (BB5Ah). This subroutine is part of the operating system of the computer, and is probably the subroutine that you will use more than any other. Amsoft have called it TXT OUTPUT, and it will print the character whose code is held in the 'A' register to the current window, at that window's current cursor position. The subroutine will also obey control codes, and these are explained in Chapter 9 of the Amstrad User Instruction book.

To give a brief example of how a control code is responded to change line 90 to: 90 DATA 62,7,205,90,187,201 and change the number after the TO in line 30 to 43885. If line 40 is still present then the checksum must be changed to 752. In assembly language the program now reads:

```
LD A,7
CALL 47962
RET
```

When RUN all that the machine code will do is sound the bell, emitting a bleep from the computer. If you hear nothing try turning the volume up and typing CALL 43880 followed by [ENTER]. This will call the machine code directly, instead of it being called by the BASIC program.

Unlike load (LD) instructions it is not possible to nominate an address by pointing to it with a register pair, the address to be called must always be in the two bytes following the CALL opcode. The RETurn at the end of a subroutine will be to the address immediately after the three bytes of the CALL (one byte for the opcode and two for the operand).

To be able to make a RETurn the CPU has to know where the subroutine was CALLed from, and it accomplishes this by using something called the Machine Stack, or just the stack for short. A brief explanation of what this is, and how it is used when a CALL or a RETurn instruction is encountered in a program, follows. Use of the stack will be discussed in detail in Chapter 9.

The stack can be compared to a spike on the ceiling, and each byte of information being saved on it, can be thought of as a piece of paper. When the piece of paper is put on the stack, the stack grows downwards, and the only information that can be taken off the stack is the piece on the bottom. In order to get something not on the bottom location, everything below will have to be taken off first.

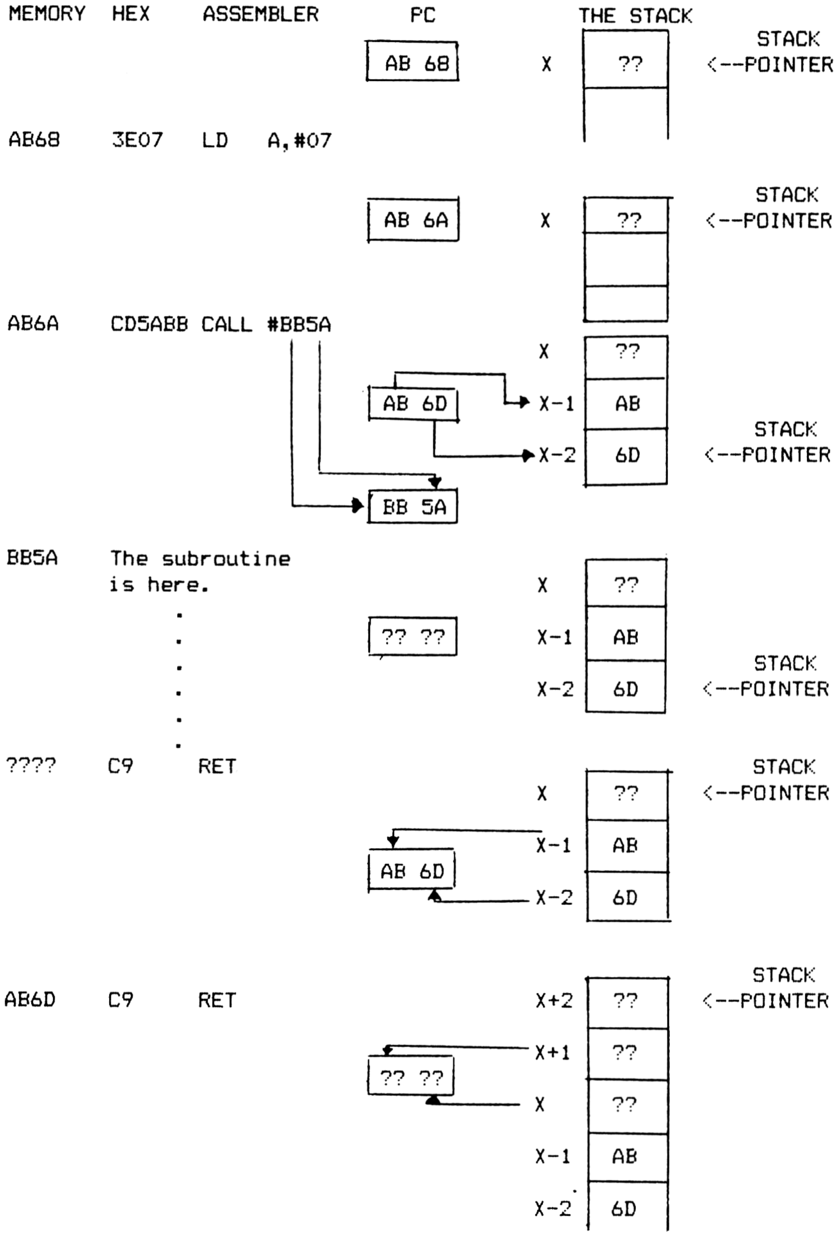
The stack occupies an area of memory, and the address of the bottom of the stack is always held in the 16 bit Stack Pointer register. The area of memory chosen for the stack must be protected from being used by the program, because if corrupted it would almost certainly cause the program to crash. Normally this is achieved by placing the stack at the top of a large area of free memory, as this allows it the maximum space to grow down without trespassing on anything else.

Whenever a CALL instruction is encountered in a program, the CPU, once it has found out the destination of the CALL, places the address currently in the PC (the address of the next instruction to be fetched) onto the stack, and replaces it in the PC with the address of the subroutine. This causes the next instruction to be fetched from the address of the CALL, and the subroutine is then executed. At the end of the subroutine, when the RET is executed, the CPU collects the return address from the stack, and puts it into the PC, thereby making program execution continue at the address after the original CALL.

The diagrams overleaf shows the sequence of events when a subroutine is CALLed and when the RETurn is executed. The assembler listing for the program is given in Fig. 5.8. The first column is the memory address of the start of the instruction, in Hex, the second column is the Hex code of the instruction and the third column is the actual assembler listing, again Hex numbers have been used. The reason for the use of Hex numbers is that they demonstrate much more clearly what is happening, since each byte of memory or single register can hold a two digit Hex number.

The example used is the program given above to sound the bell.

Every time anything is put onto the stack it grows down by two bytes, and each time something is taken off the stack it shrinks by two bytes. It is therefore very important to make sure that a program does not put more onto the stack than it takes off. This could make the stack either grow down so much that it starts to use memory used to hold a program, or have the wrong information on the bottom when something is to be taken off. Between information being put onto the stack and its being required again, there *must* be an equal number of things put onto and taken off the stack. This is absolutely vital and cannot be stressed too much, especially since there are instructions explained later, other than CALLs and RETurns which use the stack for storage of information. An imbalance of the stack is the single most



```
MEMORY   HEX      ASSEMBLER

AB68     3E07     LD    A,#07

AB6A     CD5ABB CALL  #BB5A

BB5A     The subroutine
         is here.
                .
                .
                .
                .
                .
                .
????     C9       RET

AB6D     C9       RET
```

Figure 5.8

common cause for a program crashing. Unlike with BASIC, it is often the case that all one can do when a program crashes, is switch off and start again.

Jumps

There are two types of jump instruction, the first one to be explained imitates the BASIC GOTO almost exactly. With this instruction the absolute destination is given after the command. Consider for example the BASIC command GOTO 100. There must be a line in the program with the number '100' and when the GOTO 100 is executed control is transferred to line 100. In machine code, as you know, there are no line numbers, so instead of transferring control to a line number control is transferred to an address.

The mnemonic for this instruction is JP, short for JUMP, and it is normally followed by two bytes giving the address to jump to. The actual jump is made in exactly the same way as the CALL instruction transfers control to a subroutine, except that, as there is to be no RETurn, the stack is not used. The mnemonic for this type of jump is JP nn, and it allows a jump to any location in the memory currently paged in. The instruction comprises three bytes, constructed as the three bytes for a CALL instruction, but the first byte, instead of being 11 001 101 for CALL, becomes 11 000 011 for JP. If the CALL in the last program is changed to a JP it will become:

ASSEMBLER	DECIMAL	HEX	BINARY
JP 47962	195 90 187	C3 5A BB	11 000 011 0101 1010 1011 1011

The JP instruction can also be used with the HL register pair containing the address to be jumped to. In this case the instruction only takes one byte and the jump is made to the address contained in the HL register pair. If you recall how assembly language represents 'contained in' you should already know the mnemonic. The full opcode is given below:

ASSEMBLER	DECIMAL	HEX	BINARY
JP (HL)	233	E9	11 101 001

This is one of the most useful instructions in the Z80 CPU and is often used in conjunction with a CALL to achieve an equivalent to an ON GOSUB command in BASIC. This is explained in the next chapter.

Most jumps are made to an instruction very close to the address from which the jump is being made, and the Z80 CPU has an instruction that allows a jump to be made to a location relative to the address in the program counter. Surprisingly this instruction is called a Jump Relative! In assembly language this is shortened to JR.

The JR instruction consists of just two bytes, the first is the opcode and the second, the distance of the jump from the address the program counter expects the next instruction to come from, in twos complement notation, as explained in Chapter 3. This allows a jump to be made +127 to −128 relative to the PC. The opcode for JR is:

ASSEMBLER	DECIMAL	HEX	BINARY
JR n	24 n	18 n	00 011 000 n

Normally when using an assembler there is no need to calculate the length of a JR. Instead a LABEL is used, either by making the LABEL EQUal to the address to be jumped to in the program or, more usually, by defining the LABEL within the program (EQU is a pseudo operation and was explained in Chapter 3).

To help explain this consider the example below:

ADDRESS	LABEL	ASSEMBLER	DECIMAL	HEX
43880 {AB68h}		LD A,7	62 7	3E 07
43882 {AB6Ah}	PRINT:	CALL 47962	205 90 187	CD 5A BB
43885 {AB6Dh}		LD A,65	62 65	3E 41
43887 {AB6Fh}		JR PRINT	24 249	18 F9

The 249 after the 24 opcode of the JR instruction tells the CPU to transfer execution to an address −7 from the current PC. The PC is already pointing to the next instruction since the read of the JR n has been completed, and therefore contains the address 43889. 43889 − 7 = 43882. The jump will therefore be made to the start of the CALL instruction.

If the program had been:

```
ADDRESS              LABEL   ASSEMBLER       DECIMAL         HEX

43880 (AB68h)                JR    GO        24 5            18 05

43882 (AB6Ah)                LD    A,7       62 7            3E 07

43884 (AB6Ch)        PRINT:  CALL  47962     205 90 187      CD 5A BB

43887 (AB6Fh)        GO:     LD    A,65      62 65           3E 41

43889 (AB71h)                JR    PRINT     24 249          18 F9
```

Then instead of the bell being sounded followed by 'A's until you reset the computer, the JR GO would make the PC jump the LD A,7 and the CALL 47962 and the first thing to be printed would be an 'A'. Note that the jump is only 5 which skips the five bytes *after* the JR GO.

It is possible to use calculated relative addressing with an assembler, but this is unnecessary except with very long programs where it is important to save space by not using labels. The Highsoft Devpac slightly complicates matters, in the way the GENSA3 assembler operates. Jump distances are calculated relative to the assembler's location counter and *not* the PC. This is explained on page 2.6 of the manual which comes with the Devpac cassette. In essence the location counter will be at the start of the JR instruction, when the calculation is made, and therefore you will need to add 2 to the jump distance worked out the proper way, to make the correct jump. The location counter is addressed by use of the $ symbol. The JR PRINT instruction in the routine above would be rewritten:

JR $ − 5

and not the more logical JR −7.

The Picturesque assembler, which is to be available shortly, will employ the standard Z80 method of calculation, and the instruction using this, therefore becomes the expected JR − 7.

If, as will most often be the case, except when space is at a premium, you are using labels none of this will affect you, and there is no need to worry about which assembler you are using. Any relative jump to an offset indicated by a label, will always be made to the instruction immediately following the label.

There is one last instruction to be described in this chapter, and it is one of the simplest but most useful instructions available. It enables the contents of the DE register pair to be exchanged with the contents of the HL register pair. This is extremely useful if the HL register pair is being used to point to an address in memory, but is then needed for another purpose. You will find out why this situation is likely to arise in the next chapter. As you might expect the mnemonic for the instruction is: EX DE,HL the EX being the abbreviation for EXchange. The full opcodes are given below:

ASSEMBLER	DECIMAL	HEX	BINARY
EX DE,HL	235	EB	11 101 011

If DE held 10 and HL 37 before the EXchange DE will hold 37 and HL 10 afterwards.

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number 0 to 255
nn = a 16 bit number 0 to 65535
() round a number or register pair = the address at
PC = Program Counter
SP = Stack Pointer.

LD means LOAD

Any r can be loaded with any n. The instruction has the form LD r,n.

Any r can be loaded from any other r. The instruction has the form LD r,r′.

The A register can be loaded from an address in memory. The instruction has the form LD A,(nn).

An address in memory can be loaded from the A register. The instruction has the form LD (nn),A.

Both the above can use the contents of HL register pair instead of nn. The instructions are LD A,(HL) and LD (HL),A.

Any rr can be loaded with any nn. The instruction has the form LD rr,nn.

Any rr can be loaded from any memory location and the location 1 higher. The instruction has the form LD rr,(nn).

Any memory location and memory location +1 can be loaded from any register pair. The instruction has the form LD (nn),rr.

It takes one less byte to do either of the above using the HL rr.

A subroutine is accessed by a CALL nn.

A CALL can be to any address in memory.

A subroutine is ended by a RET.

Both the above use the stack.

A jump can be made to any location in memory using the instruction JP nn.

The HL register pair can be used to contain the nn, this reduces the instruction from three bytes to one byte. The instruction then takes the form JP (HL).

A relative jump can be made to a location within the range +127 to −128 of the end of the jump instruction, which takes the form JR n.

Sixteen bit numbers are held in memory in reverse order. If the number is made into a 4 digit Hex number the two most significant digits are stored in the memory location after the two least significant digits. In a register pair they are stored in normal order High Low.

All opcodes are listed in the Appendix with a symbolic operation.

Chapter Six

Simple Maths

In the previous chapter you learnt about the instructions to load a single register with an eight bit number or with the number held in another single register, the LD r,n and LD r,r opcodes. There is also a complete set of instructions for adding and subtracting, and the opcodes for these follow very closely the LD r,n and LD r,r′ instruction forms.

You will recall that the 'A' register is also known as the accumulator, and that there is a number of 'load' operations that can only be performed using the accumulator. The 'A' register really comes into its own in 8 bit maths functions, as it is the only register which can be used to contain the result of an 8 bit maths operation.

Before going on to examine the true maths operations there are two instructions which, although they are not strictly mathematical, can be carried out using any of the general purpose registers. These are an increase by 1, or a decrease by 1. Imagine a general purpose register holding, for example, the value 99: An increase by 1 would leave it holding 100, and a decrease by 1 would leave it holding 98. The assembly language for these instructions is about as imaginative as a red tomato. To increase a register by 1 has the instruction INC r with r being any general purpose register; and to decrease by 1? You've guessed it! DEC r.

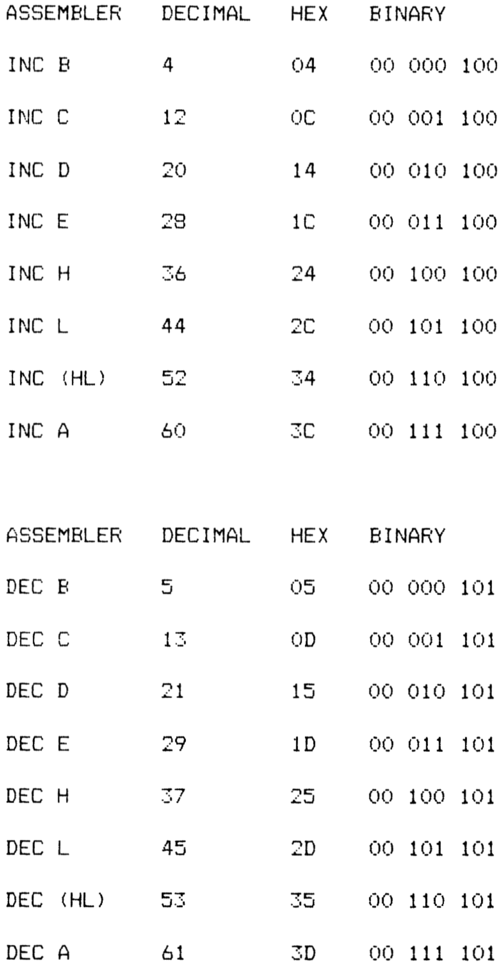
There is one case where an INC does not leave a register holding a value one greater than before, and one where a DEC does not leave it holding one less than before. As you know a single register can only hold a value between 0 and 255, so what do you think will happen if the register holds 255 when the INC instruction is encountered?

Whenever the value that can be held in eight bits is exceeded everything starts again from 0. This is rather like having a clock with only one hand, numbered from 1 to 256 clockwise round its face. If it is showing 255 o'clock (can you imagine a 256-hour day?

We'd all get old before our time!) after adding 1 hour it will be 256 o'clock but, as with the normal 24-hour clock like the digital clock you may have on your video recorder or by your bed, it does not show 24:00 at midnight but shows 00:00 instead. This is because it is not 24 hours into the old day, it is 0 hours into the new.

If you still don't know the answer to what comes after 255 try thinking of the binary, which is 1111 1111. Increase this by 0000 0001 and the result is 1 0000 0000. The register can only hold an 8 bit number, so what is in the register?

The other exception, as you have very likely realised, is when a



ASSEMBLER	DECIMAL	HEX	BINARY
INC B	4	04	00 000 100
INC C	12	0C	00 001 100
INC D	20	14	00 010 100
INC E	28	1C	00 011 100
INC H	36	24	00 100 100
INC L	44	2C	00 101 100
INC (HL)	52	34	00 110 100
INC A	60	3C	00 111 100

ASSEMBLER	DECIMAL	HEX	BINARY
DEC B	5	05	00 000 101
DEC C	13	0D	00 001 101
DEC D	21	15	00 010 101
DEC E	29	1D	00 011 101
DEC H	37	25	00 100 101
DEC L	45	2D	00 101 101
DEC (HL)	53	35	00 110 101
DEC A	61	3D	00 111 101

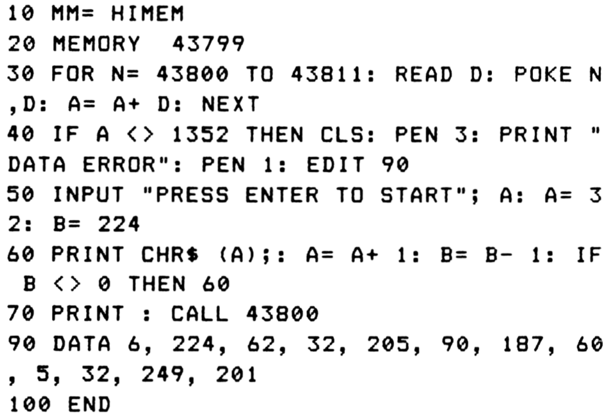
Figure 6.1

DEC instruction is enacted on a register containing 0. Using the same system as above if you find it helps, what will the register hold after the DEC is executed? In binary the register holds 0000 0000 before the DEC instruction. If you are at all confused by this, turn back to Chapter 3, and re-read the section on twos complement binary numbers.

The construction of the INC and DEC instructions is quite straightforward.

As you can see in Fig. 6.1 bits 5, 4 and 3 nominate the register to be used in the same way as they did in the load (LD) instructions.

To demonstrate the INC and DEC instructions enter the following short program:



```
10 MM= HIMEM
20 MEMORY  43799
30 FOR N= 43800 TO 43811: READ D: POKE N
,D: A= A+ D: NEXT
40 IF A <> 1352 THEN CLS: PEN 3: PRINT "
DATA ERROR": PEN 1: EDIT 90
50 INPUT "PRESS ENTER TO START"; A: A= 3
2: B= 224
60 PRINT CHR$ (A);: A= A+ 1: B= B- 1: IF
 B <> 0 THEN 60
70 PRINT : CALL 43800
90 DATA 6, 224, 62, 32, 205, 90, 187, 60
, 5, 32, 249, 201
100 END
```

Figure 6.2

The Hex and Assembly language versions of the machine code, POKEd into memory by line 30 from the DATA in line 90, are shown in Fig. 6.3.

Note that a new instruction has been introduced before the RET at the end of the routine. This will be explained properly in the next chapter, but you will probably be able to discern its meaning if you examine line 60 of the BASIC. Briefly the NZ means Not Zero, and it relates to the answer to the last maths operation. The instruction therefore means: if the result of the last maths operation was not 0 then Jump Relative to the label 'PRINT'.

```
HEX              ASSEMBLER

06 E9                  LD   B,224

3E 20                  LD   A,32

CD 5A BB    PRINT: CALL 47962

3C                     INC  A

05                     DEC  B

20 F9                  JR   NZ,PRINT

C9                     RET
```

Figure 6.3

When RUN this will print the character set of the Amstrad CPC 464 onto the screen, starting with a 'space' and going right through all the characters shown in Appendix III of the Amstrad User Instructions from page 2.

The instructions to add to the 'A' register or to subtract from the 'A' register are almost equally straightforward, in their simplest form. The mnemonic for a simple ADD is ADD, and for a simple SUBtract it is SUB. As only the 'A' register can be used for 8 bit maths it would seem unnecessary to specify which register is to be used, and in the case of the SUB mnemonic this is indeed the case. However the ADD mnemonic is also used to perform a 16 bit ADD using the HL register pair, as will be explained later. With the ADD it is normal practice to state which register is to be used, even though some assemblers do not require it. The Devpac assembler will not accept an ADD without the register being named, but it is likely the Picturesque assembler will accept an ADD on its own.

If it is required to add a number to the A register the full instruction is ADD A,n and to take a number away from the A register the instruction is just SUB n. The other forms of the instructions are:

ASSEMBLER	DECIMAL	HEX	BINARY
ADD A,n	198 n	C6 n	11 000 110 n
SUB n	214 n	D6 n	11 010 110 n

You can see the ADD A,n instruction in use if you change the BASIC program above as follows:

```
30 FOR N= 43800 TO 43812: READ D: POKE N,D: A= A+ D: NEXT
```

The 1352 in line 40 becomes 1491, and line 90 becomes

90 DATA 6,224,62,32,205,90,187,198,1,5,32,248,201

This has changed the INC A in the assembly listing to ADD A,1 and the length of the JR has also been changed otherwise the extra byte would have made the jump go to the wrong destination. When RUN there will be no difference in the result, but the code takes one extra byte of memory.

To see the SUB instruction you can alter these changes to the BASIC program:

Change the 1491 in line 40 to 1730
line 55 becomes 55 A = 255: B = 224
In line 60 change A = A + 1 to A = A − 1

and line 90 is changed to

90 DATA 6,224,62,255,205,90,187,214,1,5,32,248,201

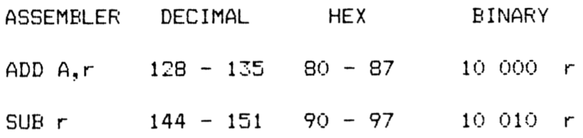
The assembly language is now:

```
HEX              ASSEMBLER

06 E9                   LD   B,224
3E FF                   LD   A,255

CD 5A BB    PRINT:  CALL 47962

D6 01                   SUB  1

05                      DEC  B

20 F8                   JR   NZ,PRINT

C9                      RET
```

Figure 6.4

As with the INC and DEC instructions when the result of an ADD or SUB is 0 the Zero flag is set, otherwise it is reset, showing that the result is Not Zero. If the execution of an ADD requires a carry over to a ninth bit, because the answer is greater than 255 or, in the case of a SUB instruction, when a borrow is needed from a ninth bit, due to the result being less than 0, there is a flag which will tell you if the operation has had to make this carry over, and it is called – predictably – the carry flag. The carry flag is always set if a carry or borrow was needed by any maths instruction, and reset if not. Note that the INC and DEC instructions do *not* affect the carry flag.

As well as being able to ADD or SUB a number and the A register it is also possible to perform these operations with another register, giving the mnemonics ADD A,r and SUB r. The other versions are:



ASSEMBLER	DECIMAL	HEX	BINARY
ADD A,r	128 - 135	80 - 87	10 000 r
SUB r	144 - 151	90 - 97	10 010 r

The r represents any general purpose register or the A register, and the codes are the same as those shown in Chapter 5.

B = 000 H = 100
C = 001 L = 101
D = 010
E = 011 A = 111

The code 110 is again used to signify (HL). That is, with the contents of the memory location whose address is held in the HL register pair.

The SUB r and ADD A,r work in exactly the same fashion as the ADD A,n and the SUB n, and affect the Zero and Carry flags in the same manner. The difference is, of course, that the r is used as a BASIC variable might be used in place of a predefined number.

Knowing the above and remembering what you learnt in the last chapter, it should be possible for you to write a short routine in assembly language, which adds the contents of memory at address 43894 to the contents of memory at address 43896 and places the result in address 43898. When you have done this, if you have an assembler you can type your routine in. If not you will have to code it manually, using the information already given, or by looking at the appendix giving the Z80 instruction set. You can then use the loader program to enter it; the start address should be 43850.

There are several ways in which this routine could be written, and two are given at the back of this book in case you get stuck. The acid test will be to see if the routine you write works, and a subroutine which will allow you to do this is given below. It uses some instructions that you do not know yet, and these will be explained shortly, but you will probably understand one of them immediately. The end of your program must be:

```
ASSEMBLER              DECIMAL           HEX

CALL 43800             205 24 171        CD 18 AB
RET                    201               C9
```

This is the subroutine to print out the result, which is CALLed from your routine (Fig. 6.5).

```
ASSEMBLER                        DECIMAL          HEX

        ORG 43800                              SET MEMORY TO AB17
        ENT  43800                             START ADDRESS AB18
        LD   A,(43898)           58 122 171      3A 7A AB   CHECK
        LD   L,A                 111             6F
        LD   H,0                 38 0            26 00
        LD   DE,-100             17 156 255      11 9C FF
        CALL REDN                205 44 171      CD             046D
                                                    2C AB
        LD   E,-10               30 246          1E F6
        CALL REDN                205 44 171      CD 2C AB
        LD   A,L                 125             7D
        JR   PRIN                24 9            18 09          042D
REDN:   LD   A,0                 62 0            3E 00
FNUM:   INC  A                   60              3C
        ADD  HL,DE               25              19
        JR   C,FNUM              56 252          38 FC
        SBC  HL,DE               237 82          ED 52
        DEC  A                   61              3D
PRIN:   ADD  A,#30               198 48          C6             0409
                                                    30
        CALL 47962               205 90 187      CD 5A BB
        RET                      201             C9             END 02DB

CHECKSUM                         3966
```

Figure 6.5

This routine can be entered from the assembly language listing if you have an assembler, or by using the Hex loader program given at the end of the book, which you should have on tape ready to load. If you feel brave, you can change the BASIC program you entered earlier in this chapter. You should be able to do this for yourself but, as a small help, there are 35 bytes in the listing, so the start of line 30 will be:

30 FOR N = 43800 TO 43834

Remember that your code will start at 43850, so when the program loads your part N should start as 43850.

If you do change the BASIC, be sure to save your amended version before you RUN it, just in case you have made an error which causes the program to crash.

The first problem you will have once you have got your machine code loaded into memory is how to get numbers into the routine, for it to add. This is where we have to resort to BASIC once more – until you get a bit further in the book!

```
400 INPUT "FIRST NUMBER";A: INPUT "SECOND NUMBER";B
410 ? A;"+";B;"=";
420 POKE 43894,A: POKE 43896,B
430 CALL 43850
440 GOTO 400
```

Figure 6.6

Now type GOTO 400 and the BASIC will ask you for the two numbers, and then POKE them into memory ready for the machine code. Your routine will then be CALLed by line 430 to do the addition, and the result will be printed by the subroutine given above. Press ESCAPE twice when you want to get out of the loop which asks for numbers and ADDs them.

If you entered any pair of numbers that added up to more than 255, you would have found that the answer was given incorrectly. You will remember the reason for this from earlier, and will hopefully also recall that the carry flag is always set when this happens. What is needed is a way to cope with numbers that will not fit into one byte, and use the carry flag to indicate when a carry needs to be dealt with.

If this is starting to confuse you, think for a moment what happens when you add 9 + 6 + 8.

9 + 6 = 5 carry 1; so you increase the tens by one
5 (from above) + 8 = 3 carry 1; again you increase the tens by one. This gives the result: two tens and three units or 23, which is correct.

Now think what happens in your program when it does the following binary addition:

```
  1010 0101 (165)
+ 1011 0000 (176)
          1     Each column is added to the column above
         +0     in exactly the  same  way  as  when  the
         =1     decimal sum above was carried  out.  But
                this time, since the sum  is  in  binary
         01     and goes on quite long enough anyway,the
        +0      carry is shown added to  the two numbers
        =01     in the next column, when it occurs.

        101     The only real carry, since up to 255 can
       +0       be  held  in  a  register  without   an
       =101     overflow, is at the end of the sum. Each
                bit falling over the end here  is  worth
       0101     256 times the least significant bit.
      +0

      =0101     If  a  register  is  used  to  store  the
                carrys when they occur it is counting in
     0 0101     units of 256.
    +1
    =1 0101     What  is needed is  a  series  of  bytes,
                from which any overflow is automatically
    11 0101     added to the next byte.
   +1
  =101 0101     This would act in exactly the  same  way
                as normal addition in  decimal,  but  to
   101 0101     base 256 instead of base ten.
  +0
  +0            In fact this is very easy to achieve and
  =101 0101     does not require  any  thought  on  your
                part. There are commands built into  the
  1101 0101     Z80  CPU  which  automatically  add   or
 +1             subtract with carry.
= 0101 0101 (85) carry 1 (worth 256)
```

These are called (surprise, surprise!) add with carry, and subtract with carry. The mnemonics for the assembler, instead of being ADDC and SUBC, are shortened to save you typing, becoming ADC and SBC. When used in place of the ADD and SUB instructions the carry flag is included in the operation. For example, consider the program in Fig. 6.7.

Imagine that 43894 holds 1010 0101 (165), 43896 holds 1011 0000 (176) and the remaining addresses all hold 0000 0000 before the program is run. The first part of the program will add 165 to 176, there will be a carry and the A register will be left holding 85. This will be stored in address 43898. The rest of the program will add 0

```
LD  HL,43896       This will load the value from address 43894
LD  A,(43894)      into the A  register  and  ADD  it  to  the
ADD A,(HL)         contents of the address pointed  to  by  HL
LD  (43898),A      (43896) and put  the  result  into  address
                   43898.
LD  A,(43895)      The process is then repeated with A holding
INC HL             the contents of 43895 and  HL  pointing  to
ADD A,(HL)         43897, the result going to address 43899.
LD  (43899),A
```

Figure 6.7

to 0 and store the result in 43899. What a waste of energy! But change the second ADD to ADC and what happens?

The first part of the program works the same as before, and when the second section is executed the carry flag is set. When this ADD with carry (ADC) is executed the calculation becomes 0 + 0 + carry, instead of just 0 + 0. The carry flag is set, so the answer this time is 1, and it is 1 that is stored in address 43899.

By adding the instruction LD HL,(43898) to the end of the program, the H register can be made to hold the high byte of the answer, and L the low byte. So after executing the program HL will hold 0000 0001 0101 0101b or 01 55 Hex. Look familiar? It should, you will find it is the correct answer to the sum attempted earlier.

Before going on to describe how to use the SBC instruction, and the ADC for numbers greater than those that can be held in two bytes, here are the numeric instruction codes:

ASSEMBLER	DECIMAL	HEX	BINARY
ADC A,n	206 n	CE n	11 001 110 n
SBC A,n	222 n	DE n	11 011 110 n
ADC A,r	136 - 143	88 - 8F	10 001 r
SBC A,r	152 - 159	98 - 9F	10 011 r

As usual r can be any general purpose register, the A register or (HL), the codes are the same as always. Note that the SBC needs the register A defined. This is because, unlike the SUB instruction, the SBC can be used in another way, which will be explained later.

You can now enter the program in Fig. 6.9 which will allow you to experiment with the ADC and SBC instructions. If you have an assembler you will be able to modify the program you wrote to ADD the contents of two memory locations, otherwise you will have to use the HEX LOADER program. The decimal listings are no longer of any relevance since the programs are now starting to become too long for entry by the 'DATA' method.

If you have used the assembler you can type R followed by [ENTER] to execute it, otherwise you will have to type CALL 43850 followed by [ENTER]. The program will then add the ASCII code of the next key you press to the contents of address 43896, and save the answer in address 43898. The contents of address 43895 is then added with carry to the contents of address 43897, and the result stored in address 43899.

A routine in the operating system is used to read the keyboard, and this is called by the CALL 47896. This CALL waits for a key to be pressed, and returns with the ASCII code for the key in the A register.

Unless you have put something into the addresses added together by the program the answer will always be the code for the key you have pressed. You can check this by looking at the codes in Appendix 3 of the Amstrad manual.

If you use the assembler type B followed by [ENTER]. This will return you to BASIC. You can now put values into the addresses that are being added, and then CALL 43850 to see the result. Remember that you will have to press a key to give part of the sum.

To add, for example, 220 and 89:

```
Type POKE 43896,220 [ENTER] CALL 43850 [ENTER]
```

Then press SHIFT and Y (capital Y has the ASCII code 89). The answer displayed will be 309.

Or to add 23260 to 345 you would type:

```
POKE 43896,220 [ENTER] POKE 43897,90 [ENTER]
POKE 43895,1 [ENTER] CALL 43850 [ENTER]
```

Then press SHIFT and Y. You should get the answer 23605, but you don't! Why not?

23260 is 5ADC in Hex and 345 is 159 in Hex. Remembering how the Z80 stores numbers in reverse, and looking at the program above, you will find that address 43896 and the keyboard provide

```
        ASSEMBLER                       HEX

                                    SET MEMORY TO AB17
        ORG   43850                 START ADDRESS AB4A
        ENT   43850                               CHECK
        LD    HL,43896                 21 78 AB
        CALL  47896                    CD 18 BB
        ADD   A,(HL)                   86
        LD    (43898),A                32 7A AB   04C1
        LD    A,(43895)                3A 77 AB
        INC   HL                       23
        ADC   A,(HL)                   8E
        LD    (43899),A                32 7B AB
        CALL  43800                    CD 18      044A
                                             AB
        RET                            C9         END 0174

                                    MORE? Y/N     Y
        ORG   43800                 START ADDRESS AB18
                                                  CHECK
        LD    HL,(43898)               2A 7A AB
        NOP                            00
        NOP                            00
        NOP                            00
        NOP                            00
        NOP                            00
        NOP                            00
        LD    DE,-1000                 11         0160
                                          18 FC
        CALL  REDN                     CD 35 AB
        LD    DE,-100                  11 9C FF
        CALL  REDN                     CD 35      056F
                                             AB
        LD    E,-10                    1E F6
        CALL  REDN                     CD 35 AB
        LD    A,L                      7D
        JR    PRIN                     18 09
REDN:   LD    A,0                      3E         0448
                                          00
FNUM:   INC   A                        3C
        ADD   HL,DE                    19
        JR    C,FNUM                   38 FC
        SBC   HL,DE                    ED 52
        DEC   A                        3D
PRIN:   ADD   A,#30                    C6 30      03F6
        CALL  47962                    CD 5A BB
        RET                            C9         END 02AB
                                    MORE? Y/N     N
```

Figure 6.8

the low bytes of the sum (the DCh and the 59h) and addresses 43895 and 43897 provide the high bytes (5Ah and 01h).

DCh is 220 and 59h is 89 so the 220 is poked into 43896 and the capital Y provides the 89 for the first part of the sum, these are then added and the result placed in 43898. This should be 35h as 59h + DCh = 135h. The 1 is a carry and what is left is the 35h, or 53. If you look into address 43898 by typing:

```
? PEEK (43898) [ENTER]
```

you will see this is indeed the case. The 1h and the 5Ah (the high bytes of the sum) are 1 and 90 respectively, these are the numbers that were poked into addresses 43895 and 43897 to be added with carry in the second section of the program. 5Ah + 01h + carry (which is set and therefore equal to 1) = 5Ch or 92. This should be found in address 43899. Check it and see if it is correct.

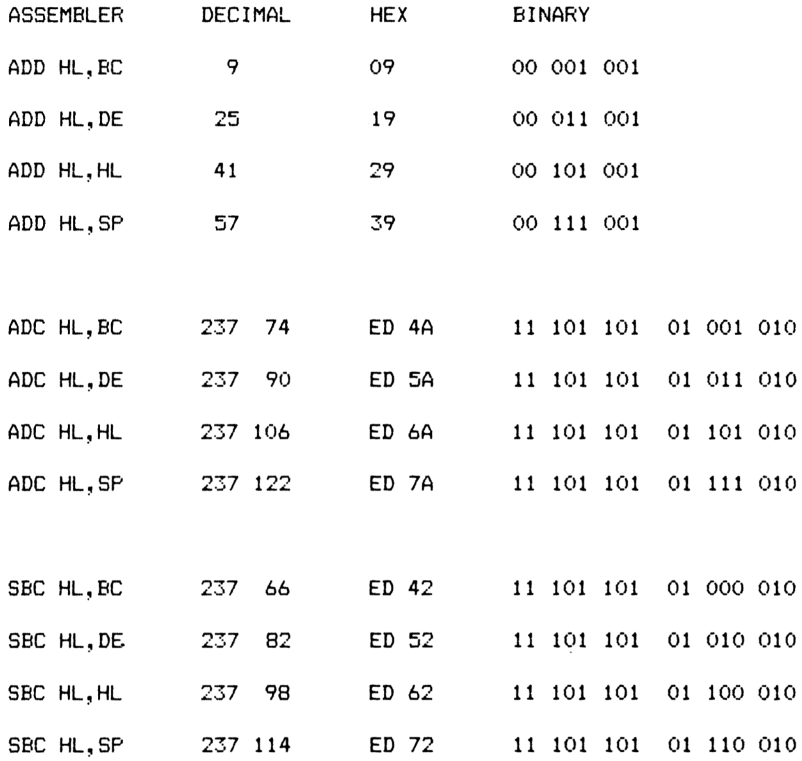
The answer to the sum is 5C35h, 5Ch is 92, it is the high byte so worth 256 * its face value which is 23552. The low byte is 35h, 53. This is then added to the value of the high byte; 23552 + 53 = 23605 or 5C35h. So the sum is correct. Why then is the answer being displayed incorrectly?

The answer to this will be found by an analysis of the second section of the program above. This is the bit that does the printing of the answer to the screen. (There is a monumental clue in the series of NOPs, no one in his right mind would waste space if there was nothing to fill it later!)

There are two instructions (ADD HL,DE and SBC HL,DE) that you are not familiar with, so these will be explained briefly first. A fuller explanation follows later in this chapter, and the ways the instructions can be employed are examined in more detail in later chapters.

As mentioned earlier, the only addition or subtraction instruction unique to the A register is the SUB instruction. The other way in which these maths operations can be used, is with the HL register pair being used as a 16 bit Accumulator. In all cases the HL register pair will hold the answer after the operation, in the same way as the A register does after an 8 bit addition or subtraction.

Unlike with the 8 bit instructions using the A register the operand, in this case the number to be added to or taken away from the HL register pair, can only be supplied from one of the general purpose register pairs, or the SP (Stack Pointer) special purpose register. It is not possible to use a numeric operand (for example, ADD HL,23456), a memory location (for example, ADC HL,(23456)) or even a memory location addressed by a register pair (for example, SBC HL,(DE)). The Stack Pointer will be dealt with in detail in a later chapter. The available instructions are:



ASSEMBLER	DECIMAL	HEX	BINARY
ADD HL,BC	9	09	00 001 001
ADD HL,DE	25	19	00 011 001
ADD HL,HL	41	29	00 101 001
ADD HL,SP	57	39	00 111 001
ADC HL,BC	237 74	ED 4A	11 101 101 01 001 010
ADC HL,DE	237 90	ED 5A	11 101 101 01 011 010
ADC HL,HL	237 106	ED 6A	11 101 101 01 101 010
ADC HL,SP	237 122	ED 7A	11 101 101 01 111 010
SBC HL,BC	237 66	ED 42	11 101 101 01 000 010
SBC HL,DE	237 82	ED 52	11 101 101 01 010 010
SBC HL,HL	237 98	ED 62	11 101 101 01 100 010
SBC HL,SP	237 114	ED 72	11 101 101 01 110 010

Figure 6.9

In operation they are exactly the same as their equivalents for 8 bit values, ADD A,B ADC A,B and SBC A,B, etc., except that they operate on 16 bit values.

For example to ADD 55536 and 2000 (decimal), the Assembly language program might look like this:

```
LD DE,55536
LD HL,2000
ADD HL,DE
```

After the program was executed the HL register pair would hold the answer, 57536 (E0C0h E0h in H C0h in L) and the DE register pair would still hold 55536 (D8F0h D8h in D F0h in E) and the carry flag would be reset. If the sum had been 55536 + 23605 then the answer, instead of being 79141 (13525h) which cannot fit into sixteen bits, would be 13605 (3525h), and the carry flag would be set. The bit carried over from the sum has a value 65536 * the value of the least significant bit of the register pair. This is 2^16, whereas when a carry occurs after an 8 bit operation it is worth 256 * the least significant bit, which is 2^8.

Had the ADD been replaced by ADC the carry would have been added in as well, again identical to the operation with the 8 bit instructions.

The SBC instruction, when used with the HL register pair, can also be made analogous to the 8 bit SBC with the A register. Due to the absence of a SUB instruction for 16 bit operations, if the carry flag is not required to be subtracted from the HL register pair, it must be reset if it is set, otherwise the answer may be one less than it should be.

There are instructions to Set the Carry Flag, and to Complement the Carry Flag (SCF and CCF) but there is no instruction to reset the carry flag. It is possible to achieve the resetting of the carry flag by first setting it and then complementing it (SCF followed by CCF) but this is long-winded and the instruction AND A does the same job with one less instruction. This is one of the logical operations which is explained in Chapter 8.

Returning at last to the problem of why the answer to the sum is incorrect, and the analysis of the second half of the program, where it has been established the problem resides.

The first section has placed the high byte of the answer in address 43899 and the low byte in address 43898.

The first instruction of the second section is

```
LD HL,(43898)
```

this loads the contents of the address named in the instruction into the L register, and the H register is loaded from the next address.

After the instruction is executed L will therefore hold 35h and H will hold 5Ch, making HL = 5C35h or 23605. This is correct so the problem does not lie here.

Next there are six NOPs, these do No OPeration so the problem is not here.

Now the DE register pair is loaded with −1000 which is FC18h. This looks peculiar but may be correct; what happens next?

A subroutine named REDN (short for REDuce Number) is CALLed. This will be examined as a unit, broken into stages.

1) LD A,0 — A = 0

2) INC A — A = A + 1

3) ADD HL,DE — HL = 5C35h and DE = FC18h the first time the subroutine is called. FC18h is 64536 in decimal or −1000 if it is in 2s complement. 23605 + 64536 = 88141. The largest number that can fit into sixteen bits is 65535 so a carry occurs at 65536. 88141 − 65536 = 22605. 1000 has therefore been taken away from the HL register pair.

4) JR C,FNUM — FNUM (short for Find NUMber) is stage 2. So each time there is a carry as a result of the ADD HL,DE the A register is increased by 1. In other words the A register counts the number of times DE is ADDed to HL.

5) SBC HL,DE — To get here there must have been no carry from the ADD HL,DE at stage 3. It was not therefore possible to subtract DE from what was left in HL, and the answer is wrong. By using the SBC instruction with a negative value the end result is an addition. (Did your teacher at school try to explain that a minus and a minus are a plus? This proves it.) The carry flag is known to be reset so it will not affect the calculation.

6) DEC A — Since DE has been added back by the previous instruction, the count (in A) must be reduced. A has now been increased the number of times DE was taken from what was in HL at the start of the subroutine. HL now holds what HL held at the start of the subroutine – (A * DE). In the case of this example where DE was −1000 HL = 605 and A = 23.

7) ADD A,#30 — When using the Highsoft assembler the # sign signifies that the next digits are to be

interpreted as a Hex number. A was 23 (17h) so after adding 30h (48) it becomes 71 (47h).

8) CALL 47962 This is the tried and tested 'print the character whose code is in the A register' ROM routine.

9) RET The end of the subroutine.

The reason for the error in the answer to the test example should now be apparent. If the result is greater than 9000 DE (which is −1000) will be taken away from HL more times than there are numbers, that is more than 9 times. The A register will therefore hold a number in excess of 30h + 9 = 39h (57) when the print routine is called. Numbers from 30h to 39h inclusive are the ASCII codes for the numbers 0 to 9, which is how the answer is printed, but codes above 39h are used for punctuation and letters. The character with the ASCII code 71 is G, hence G was printed in place of the numbers 2 and 3.

You can probably see what needs to be done to make the routine work for any number that can be represented in sixteen bits. The following instructions can be inserted in place of the NOPs.

If you are using the assembler type

CALL 30004 [ENTER]

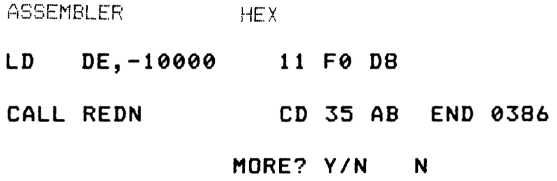
Then press L and [ENTER] to list the program. Next enter the following instructions in place of the first two NOPs and delete the remaining four NOPs, then press A [ENTER] [ENTER] [ENTER] to reassemble the code.

For those of you using the HEX LOADER type

RUN and [ENTER]
SET MEMORY TO AB17

and

START ADDRESS AB1B



```
ASSEMBLER            HEX

LD    DE,-10000      11 F0 D8

CALL  REDN           CD 35 AB   END 0386

                  MORE? Y/N    N
```

Now execute the program as before, and you should find that it works for any two numbers whose sum can be held in sixteen bits (<65535).

You can now change the first section of the program, the part which adds the numbers starting at address 43850, to experiment with the other 8 bit add and subtract instructions. As long as you always use the (HL) to point to the address where the numbers are held, and you do not use the instructions which have n or (nn) as part of the assembler representation, all that will need changing is the byte containing the instruction. Remember that machine code is not like BASIC, you cannot insert instructions!

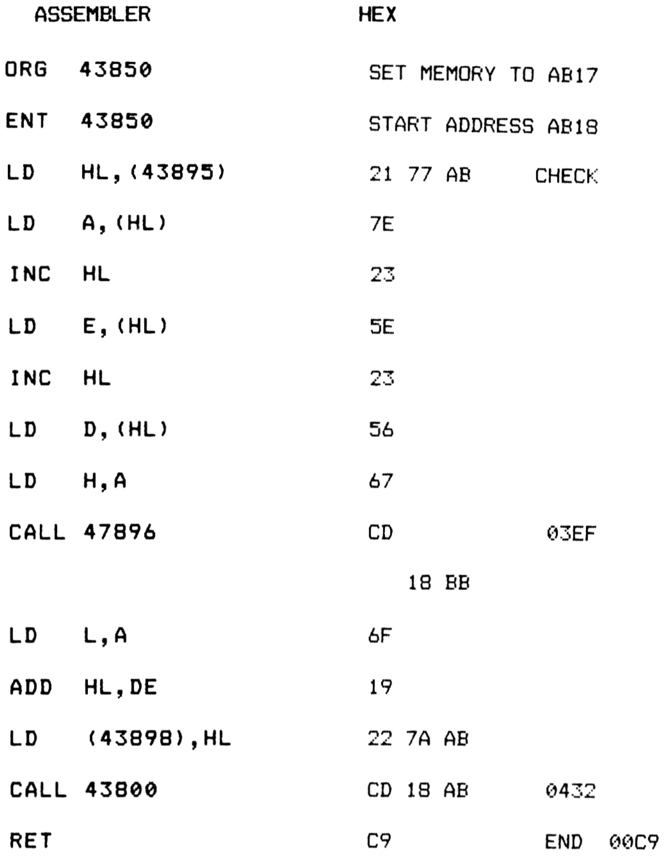
When you are happy that you understand what is happening with each of the instructions read on.

You should now be reasonably conversant with 8 bit maths, and if required it ought not to be an impossible task for you to write a program which will add any two numbers or take any number away from any other number, using solely 8 bit operations. How to output the result may still be a problem. If the screen is to be used a modification of the program you are using at the moment will give the desired result.

These sorts of tasks are where 16 bit maths really starts coming into its own. To be able to add successfully any two 16 bit numbers it is necessary to deal with 17 bit results, but to be able to multiply any two 16 bit numbers a 32 bit result must be catered for. The availability of 16 bit maths operators makes it worth while allowing for 32 bit results, since no more instructions are used than for twenty-four bits using 16 and 8 bit maths. This gives the possibility of using numbers up to 4294967295 (2^32), which is likely to be able to cope with anything short of the national debt.

The limitation imposed by not being able to use 16 bit maths on numeric operands is easily overcome. The first section of the program in Fig. 6.8 used 8 bit maths, but it could have been written as shown in Fig. 6.10.

This uses 21 bytes, which represents a saving of 1 byte over the original. Whilst the result will almost certainly need to be stored in memory for use at a later stage, it is available for immediate use in the HL register pair, whereas with the 8 bit routine the result was never available except in memory. This necessitated the LD HL,(43898) instruction in the 'print number' routine, which can now be omitted, thereby saving a further three bytes.



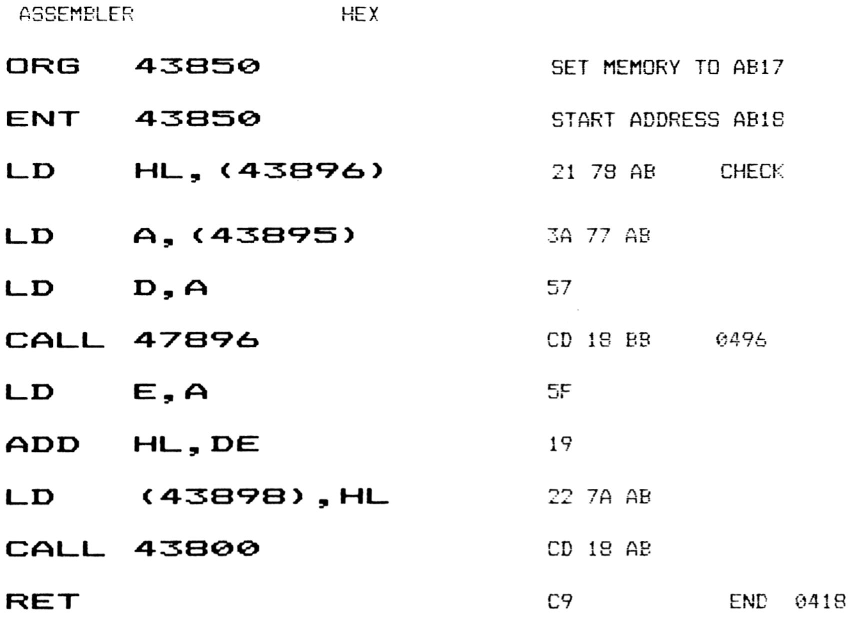
```
  ASSEMBLER                   HEX

ORG   43850                  SET MEMORY TO AB17

ENT   43850                  START ADDRESS AB18

LD    HL,(43895)             21 77 AB      CHECK

LD    A,(HL)                 7E

INC   HL                     23

LD    E,(HL)                 5E

INC   HL                     23

LD    D,(HL)                 56

LD    H,A                    67

CALL 47896                   CD                03EF

                                18 BB

LD    L,A                    6F

ADD   HL,DE                  19

LD    (43898),HL             22 7A AB

CALL 43800                   CD 18 AB          0432

RET                          C9                END  00C9
```

Figure 6.10

A further saving in memory usage can be made by using 16 bit loads in place of the 8 bit loads where possible. This makes the program as shown in Fig. 6.11.

The byte count is now reduced to only 19.

This sort of routine could be used to keep the score in an arcade game, or for a multitude of other uses, but as it stands it is limited by the fact that it cannot be called as a subroutine. This is because set locations are used for the values to be added and the result to be saved. If the routine was to be used for scorekeeping in a game of Space Invaders, every different scoring invader would need a separate routine. So if there were invaders that scored 10, 20, 50, and 100 and a Mothership which scored 400, the routine would



```
ASSEMBLER                   HEX

ORG     43850               SET MEMORY TO AB17

ENT     43850               START ADDRESS AB18

LD      HL, (43896)         21 78 AB      CHECK

LD      A, (43895)          3A 77 AB

LD      D,A                 57

CALL    47896               CD 18 BB      0496

LD      E,A                 5F

ADD     HL,DE               19

LD       (43898),HL         22 7A AB

CALL    43800               CD 18 AB

RET                         C9             END  0418
```

Figure 6.11

have to be written five times. A further problem would be that the answer from one CALL would need to be made one of the parts of the addition for the next CALL.

What is needed is a subroutine which will add two numbers given by the program which calls it, and then returns the answer, ready for saving by the main program. This can be written using registers to carry information. Using the scenario above this could be achieved by using HL to hold the score and DE to hold the value of the raider hit. Then the subroutine to add the two numbers together and print the score onto the screen would be called, and on return the result, in the HL register pair, would be saved as the new score.

The routines in Figs 6.10 and 6.11 can also be used to subtract two numbers, but the carry flag must be reset before the subtraction is made. If this is not done an incorrect result may be given. The AND A instruction mentioned earlier is used to reset the carry flag, in the example given in Fig. 6.12, which is a simple rewrite of Fig. 6.11.

```
 ASSEMBLER              HEX

ORG 43850               SET MEMORY TO AB17

ENT    43850            START ADDRESS AB18

LD     HL,(43896)       21 78 AB      CHECK

LD     A,(43895)        3A 77 AB

LD     D,A              57

CALL 47896              CD 18 BB      0496

LD     E,A              5F

AND    A                A7

SBC    HL,DE            ED 52

LD     (43898),HL       22 7A AB

CALL 43800              CD 18 AB      051C

RET                     C9            END  00C9
```

Figure 6.12

This will take DE away from HL and leave the result in HL.

It was mentioned earlier that, by using 16 bit instructions, it was worth while allowing for 32 bit results. Can you write a program which adds any two 16 bit numbers and stores the correct result in memory as a 32 bit number? The answer should be stored in successive memory locations, with the least significant byte stored at address 43896 and the most significant at address 43899.

Before starting to work out your program enter the following routine which will allow the result to be displayed. It will also give you some clues as to how to write your part.

Some things to remember, that may help, are:

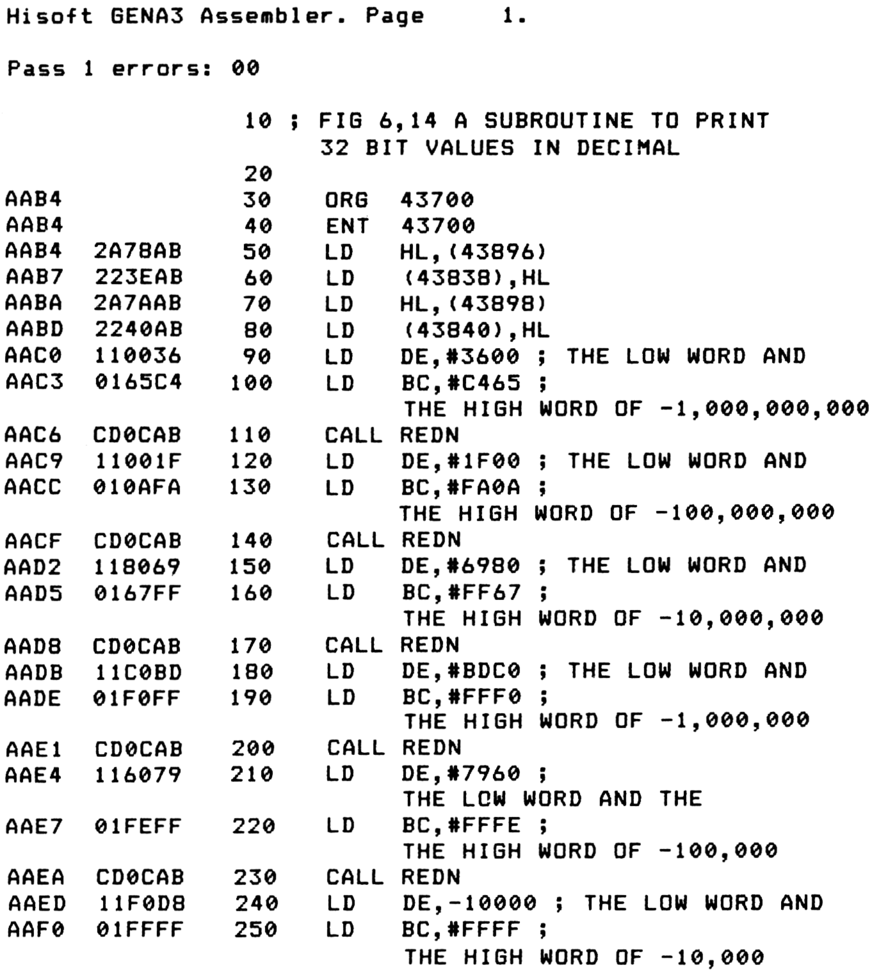
1) The answer may be greater than that which can be held in a register pair. Each part will therefore have to be stored in memory when not being used.
2) The program is really only a rewrite of the first part of Fig. 6.8, using 16 bit maths in the place of 8 bit, and not relying on an input from the keyboard.

3) The program to print the number does a 32 bit subtract, and a 32 bit ADD will be very similar.
4) The program to print the number is a 32 bit version of the second part of the program in Fig. 6.8.

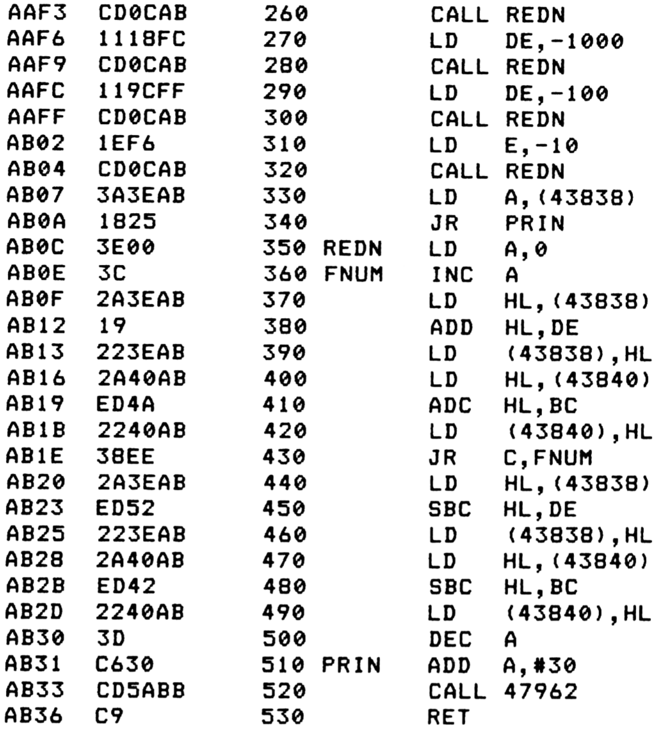
When you enter your addition program into the computer it should start at address 43840 (AB40h) and finish with

CALL 43700 RET

The program for the assembler is listed in Fig. 6.13, taken directly from the assembler to ensure accuracy. All that you have to enter is the column to the right of the line numbers, but remember to put colons after the labels.

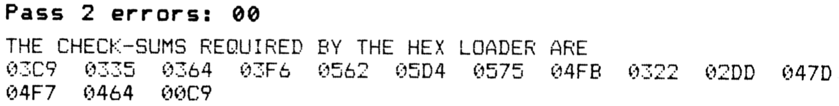


```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                    10 ; FIG 6,14 A SUBROUTINE TO PRINT
                         32 BIT VALUES IN DECIMAL
                    20
AAB4                30    ORG  43700
AAB4                40    ENT  43700
AAB4   2A78AB       50    LD   HL,(43896)
AAB7   223EAB       60    LD   (43838),HL
AABA   2A7AAB       70    LD   HL,(43898)
AABD   2240AB       80    LD   (43840),HL
AAC0   110036       90    LD   DE,#3600 ; THE LOW WORD AND
AAC3   0165C4      100    LD   BC,#C465 ;
                               THE HIGH WORD OF -1,000,000,000
AAC6   CD0CAB      110    CALL REDN
AAC9   11001F      120    LD   DE,#1F00 ; THE LOW WORD AND
AACC   010AFA      130    LD   BC,#FA0A ;
                               THE HIGH WORD OF -100,000,000
AACF   CD0CAB      140    CALL REDN
AAD2   118069      150    LD   DE,#6980 ; THE LOW WORD AND
AAD5   0167FF      160    LD   BC,#FF67 ;
                               THE HIGH WORD OF -10,000,000
AAD8   CD0CAB      170    CALL REDN
AADB   11C0BD      180    LD   DE,#BDC0 ; THE LOW WORD AND
AADE   01F0FF      190    LD   BC,#FFF0 ;
                               THE HIGH WORD OF -1,000,000
AAE1   CD0CAB      200    CALL REDN
AAE4   116079      210    LD   DE,#7960 ;
                               THE LCW WORD AND THE
AAE7   01FEFF      220    LD   BC,#FFFE ;
                               THE HIGH WORD OF -100,000
AAEA   CD0CAB      230    CALL REDN
AAED   11F0D8      240    LD   DE,-10000 ; THE LOW WORD AND
AAF0   01FFFF      250    LD   BC,#FFFF ;
                               THE HIGH WORD OF -10,000
```



```
AAF3  CD0CAB      260            CALL REDN
AAF6  1118FC      270            LD   DE,-1000
AAF9  CD0CAB      280            CALL REDN
AAFC  119CFF      290            LD   DE,-100
AAFF  CD0CAB      300            CALL REDN
AB02  1EF6        310            LD   E,-10
AB04  CD0CAB      320            CALL REDN
AB07  3A3EAB      330            LD   A,(43838)
AB0A  1825        340            JR   PRIN
AB0C  3E00        350 REDN       LD   A,0
AB0E  3C          360 FNUM       INC  A
AB0F  2A3EAB      370            LD   HL,(43838)
AB12  19          380            ADD  HL,DE
AB13  223EAB      390            LD   (43838),HL
AB16  2A40AB      400            LD   HL,(43840)
AB19  ED4A        410            ADC  HL,BC
AB1B  2240AB      420            LD   (43840),HL
AB1E  38EE        430            JR   C,FNUM
AB20  2A3EAB      440            LD   HL,(43838)
AB23  ED52        450            SBC  HL,DE
AB25  223EAB      460            LD   (43838),HL
AB28  2A40AB      470            LD   HL,(43840)
AB2B  ED42        480            SBC  HL,BC
AB2D  2240AB      490            LD   (43840),HL
AB30  3D          500            DEC  A
AB31  C630        510 PRIN       ADD  A,#30
AB33  CD5ABB      520            CALL 47962
AB36  C9          530            RET
```

Figure 6.13



```
Pass 2 errors: 00
THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
03C9  0335  0364  03F6  0562  05D4  0575  04FB  0322  02DD  047D
04F7  0464  00C9
```

If you do get stuck writing the 32 bit addition program, there is one possible solution elsewhere in the book. Try to follow what it does and then rewrite it another way. It is also suggested that you re-read the latter part of this chapter if you still have problems.

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number

nn = a 16 bit number
() round a number or register pair = the address at
PC = Program Counter
SP = Stack Pointer.

The INC r or DEC r adds 1 to or takes 1 from r. Both affect the zero flag according to the result. If the result is 0 then the flag is set, otherwise it is reset.

INC rr and DEC rr operate as above, but they work on a register pair as if they were one 16 bit register. These instructions do not affect any flags.

The A or accumulator register is the only register that can contain the result of an 8 bit maths operation.

Eight bit maths operations are:

SUB r SUB n SUB (nn) SUB (HL) which subtract from the A register.

ADD A,r ADD A,n ADD A,(nn) ADD A,(HL) which add to the A register.

SBC A,r SBC A,n SBC A,(nn) SBC A,(HL) which subtract with carry from the A register.

ADC A,r ADC A,n ADC A,(nn) ADC A,(HL) which add with carry to the A register.

The HL register pair must be used to contain the result of a 16 bit maths operation.

The 16 bit maths operations are:

ADD HL,rr which adds the contents of rr to the HL register pair.

ADC HL,rr which adds with carry to the HL register pair.

SBC HL,rr which subtracts with carry from the HL register pair.

If there is a borrow needed or a carry over from any maths operation the carry flag will be set, otherwise it is reset. If the result of a maths operation other than a 16 bit addition is 0 the zero flag is set, otherwise it is reset.

The instruction AND A is used to ensure the carry flag is reset regardless of its previous condition, when the SBC instruction does not require the carry in a 16 bit operation.

Chapter Seven

Flags, Conditions and Decision Making

In the last chapter the use of the carry flag in mathematical operations was examined in some detail, and it was shown how the result of a maths calculation affected this flag. The zero flag was also briefly introduced.

Both these flags are single bits in a special register known, predictably, as the Flag Register. As you all know there are eight bits in a register, so what are the rest used for? Correct! There are more flags to indicate other things. The flag register is made up as follows:

S	Z		H		P/V	N	C
SIGN	ZERO	NOT USED	HALF CARRY	NOT USED	PARITY/ OVERFLOW	ADD/ SUBTRACT	CARRY
M/P	Z/NZ				PE/PO		C/NC

The letter at the top of each flag bit is the abbreviation used by Zilog to identify the flag; it will be found in the Appendix of instruction codes, as well as in most literature about the Z80. Next is the full name of the flag, and last the programmer's means for testing the condition of the flag. You will note, however, that only four of the flags are accessible to the programmer; the remainder are used internally by the CPU.

Imagine the CPU in your computer as being a railway engine, and the program as being the rails that the train is running along. The train can be made to change the track it is running on by switching points, and somewhere along the line there will be a man in a signal box who will pull a lever to change the points. He will know in advance where the train is going and which points to set in which direction. This is the sort of action taken by jumps and calls in a program. But what happens if the train is to go different ways according to the amount of freight and to which stations it is consigned?

The only way the signalman can know which way to switch the points is if the train-driver lets him know. Problem! How to let the signalman know. The train must not stop otherwise it will be late, not to mention the fuel that will be wasted. So a system of flags is devised, but the driver can only signal one thing at a time, because he can only hold one flag, and he can only signal yes or no, because he is going too fast to be able to use semaphore.

With machine code it is the same, and it is the flags that form the basis for all decision making, and only yes or no to a specific question can be signalled by a flag. Since there are only four flags that can be employed, some thought has to be applied in order to arrive at the decision required; it's a little like twenty questions.

It has already been shown that arithmetic operations affect flags, but it is often the case that, when a test is required, it is unacceptable to alter whatever is being tested. The score example in the last chapter is a good instance of this. Consider the situation where a table of highest scores is being maintained, and the score from the last game is being investigated as to eligibility. It would be a bit daft if the only way to find out if the new score is higher than that at the top of the table, is to subtract the old high score from the one under investigation, and then test the carry flag. If it is not set, that is no carry (NC), it is then known that the new score was higher than the previous best.

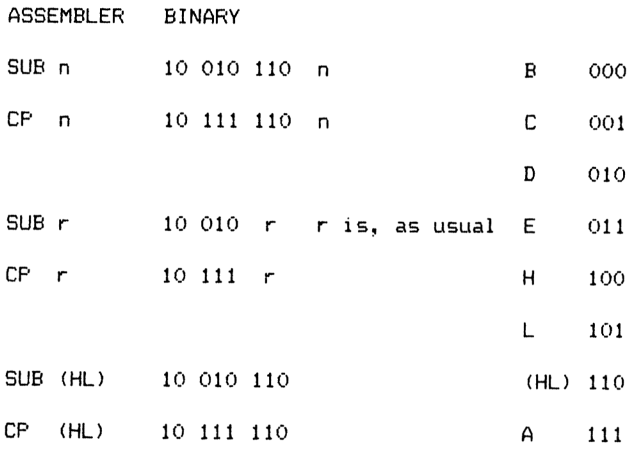
Before criticising the change in tense, consider what will happen when the new score is put at the top of the high score table. Assume that the previous best is 15575 and the latest was 21024, when the test was made. The score that will be put at the top of the table is 5449. Daft!

It will nearly always be possible in this situation to reclaim the value previous to the test, but why should one have to? It would be much nicer if there was a sort of 'dummy' subtraction to make the test. An instruction to compare and set the flags, without altering anything else. There is, and what's more it is called compare! Fiendishly cunning, these Zilogians!?!

As usual the mnemonic is an abbreviated version of the English, and compare is shortened to CP. It acts exactly like the SUB instruction, but does not change the contents of any register except the flag register. You will recall that the SUB instruction only operates with the A register, the CP also only operates on the A register.

All maths operations except the 16 bit ADD influence all the flags; the only usable flag affected by the 16 bit ADD is the carry flag. It is therefore normally the case that a compare will not be necessary after a maths operation, upon whose result a decision depends.

The compare instruction is, as you would expect, constructed in the same way as 8 bit maths instructions, the only change is to bits 5, 4 and 3. These bits become 111 in place of what they would have been for any other operation. For example:



ASSEMBLER	BINARY			
SUB n	10 010 110 n		B	000
CP n	10 111 110 n		C	001
			D	010
SUB r	10 010 r	r is, as usual	E	011
CP r	10 111 r		H	100
			L	101
SUB (HL)	10 010 110		(HL)	110
CP (HL)	10 111 110		A	111

In the previous chapter two flags were introduced, the carry flag and the zero flag. These were utilised by the program, to enable it to make decisions dependent upon their condition. The term used to describe instructions which act according to the condition of a flag is, as always, boringly predictable. They are known as 'conditional' instructions. In BASIC the "IF such-and-such THEN so-and-so" statement is conditional, and the THEN is so often followed by a GOTO that many BASICs, including your Amstrad's, even allow the GOTO to be omitted.

In machine code things are much the same. The analogy between the GOTO in BASIC and jump instructions in machine code has already been emphasised, but this similarity goes even further than has so far been pointed out. In the program in Fig. 6.3 a jump (JR) was made according to the condition of the zero flag, and the carry flag was employed in the same way, by the program in Fig. 6.5. This is almost identical to the IF . . . THEN structure of BASIC.

The apparent lack of things that can be tested for is in fact no real limitation, as a short analysis of the flags and the things that they indicate will soon show. The carry flag will be examined first, because you already have some experience of it.

With the exception of INC and DEC instructions, any operation which causes an overflow from the register or registers being operated on sets the carry flag, and conversely any operation which could cause the carry flag to be set, will reset it if there is no overflow.

To make this clearer a few examples are given below:

```
LD   A,0

DEC  A    Will re-set the carry flag, but  LD   A,0

                                           SUB  1   will set it.

LD   B,156

LD   A,100                                    LD   BC,65000

ADD  A,B  Will set the carry flag, as will    LD   HL,5536

                                              ADC  HL,BC     or

LD  BC,65000

LD  HL,5536      LD  BC,5536

SBC HL,BC   but  LD  HL,65000       LD   A,225

                 SBC HL,BC      or  ADD  A,25

                                will leave the flag reset.
```

In brief, any 8 bit addition which gives an answer over 255, or any 16 bit addition which comes to over 65535 will set the carry flag, as will any subtraction whose answer is less than 0. It is similar to a BASIC > (greater than) or < (less than).

Most of the instructions that affect the carry flag also set the zero flag if the result is 0, or reset the zero flag if the answer is not 0. The only exception to this rule, with the instructions introduced so far, is the 16 bit ADD HL. This opcode leaves the zero flag in the same condition as it was before the ADD HL was executed. The zero flag can be thought of as being equivalent to the BASIC = (equals).

This similarity is most profound when the CP instruction is employed to set the flags. The zero flag will always be set if there is no difference between the contents of the A register and what the A register is being compared with, and reset otherwise.

There are also a great many instructions that modify the zero flag and *not* the carry flag, but so far, as was pointed out at the time, the 8 bit INC and DEC are the only two cases. From now on as new instructions are broached, the way they alter the flags which are accessible to you, the programmer, will be detailed.

By testing just the carry and zero flags after a well-thought-out comparison (CP) it is possible to answer almost any questions which can be answered yes or no. At first you will find that you often get unexpected results but, once you learn to think like a microprocessor, it will become second nature to ask the right question and look at the correct flag.

Beware of the snap decision, or using two tests where different flags may indicate both answers in response to one well-thought-out test! Look at the following program which jumps to various labels according to the value in the A register. What is being sought is the code for the letter "A", but more information is also required.

1) Does the A register contain an ASCII code?
2) If it does THEN is it a code for a letter?
3) IF it is THEN is it the code for the first letter of the alphabet?

```
CP 128          All valid ASCII codes are below 128
JR NC,NOTASC    If there is a carry the A register must hold a
                value below 128. No carry and it must be 128
                or over hence A does not hold an ASCII
                code. So jump to the label NOTASC if NC
                (no carry)
CP 32           All ASCII letter codes are over 32. If there is
                a carry A must hold a value below 32
JR C,NOTLET     Jump on carry to NOTLET (NOT LETter)
CP 65           This could have been written CP "A"
JR Z,ISA        IF there is no difference THEN the zero flag
                will be set so, jump on zero.
```

At this point you might think that it is safe to assume the A register holds an ASCII letter code and it is not A, but you would be wrong. Look at Appendix III in your Amstrad manual and you will find that whilst codes over 31 are all 'printable' letters do not start until code 65. So if you now change the instruction CP 32 to CP 65, and delete the original CP 65, you have not only saved an instruction but also avoided a misleading result. There are still a number of things wrong. ASCII codes for letters do not continue to code 127, all codes over 122 are punctuation, as are codes 91 to 96 inclusive. Can you add the necessary instructions to jump to the label NOTLET with these codes? Try to work out a solution before reading on. One clue, you will need an extra label ISLET (IS LETter).

You should have added the following instructions:

```
CP 123          ; The first code that is not a letter
JR NC, NOTLET   ; No Carry means A must hold over 123 (the
                  last letter)
CP 91           ; The first non letter value after the
                  CAPITALS
JR C,ISLET      ; If there is a carry then A must hold a value
                  below 91. Therefore as all values below 65
                  have already been excluded, A must hold
                  the code for a CAPITAL LETTER
CP 97           ; The code for the first lower case letter
JR C,NOTLET     ; Bearing in mind the previous instructions,
                  this decision can be made. JR NC,ISLET
                  would also have worked, but!
```

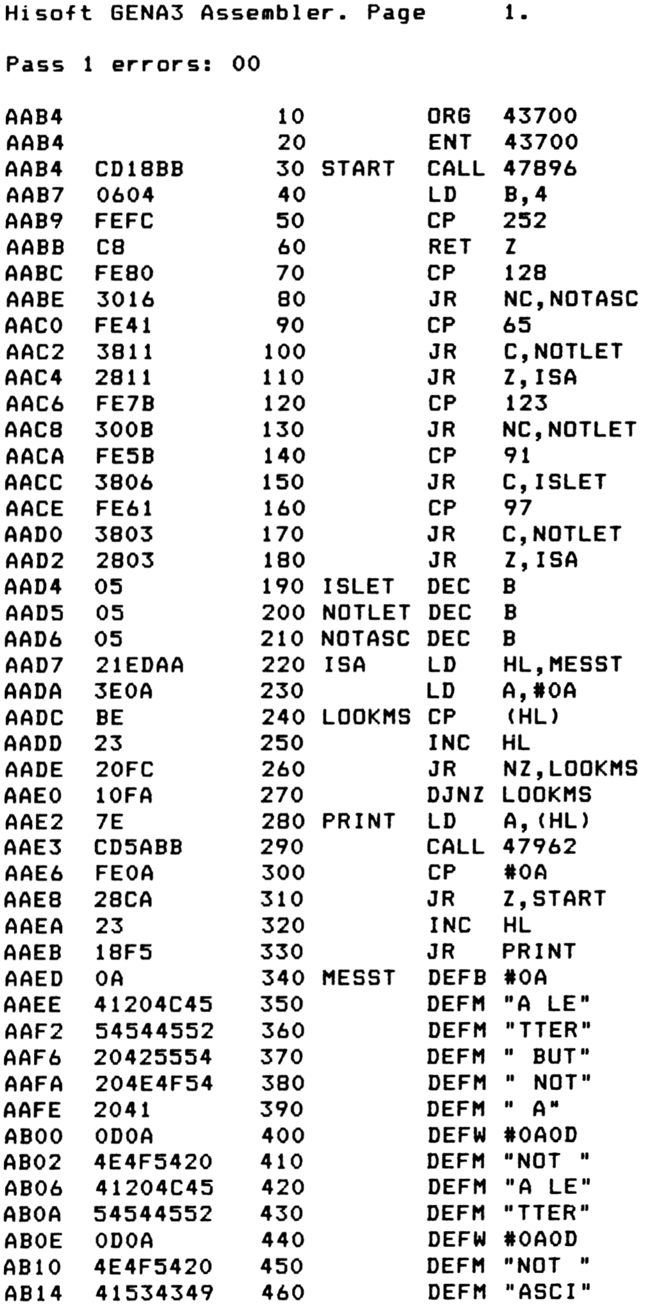
If you were testing for someone pressing "A" from the keyboard, what happens if they have pressed a lower case "a"? Nothing! This is the sort of thing that you must be very careful of, whenever you are looking for an input from the keyboard, or searching through text. The addition of one further instruction will correct the program.

```
JR Z,ISA        The zero flag will be set if A contains 97 (the
                code for "a")
```

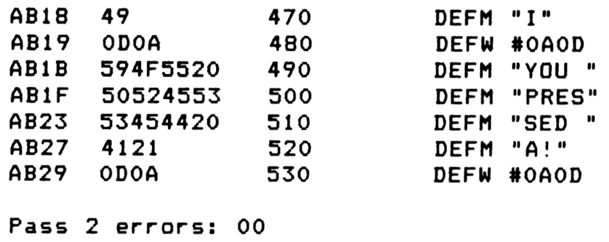
ISLET (the label for the address the program transfers control to if the A register holds the ASCII code for a letter which is not "A" or "a") should be placed at the end of the program. This avoids the addition of another jump, because the program's flow will arrive at this label naturally, if no jumps at all have been made, and the A register will be holding an ASCII letter code, not "A" or "a".

Enter this program into your computer and experiment with it until you are happy that you understand how it works. Then change it to look for another letter. For those of you using an assembler it will be easy to make the modifications once you have worked out your new coding. Those using the HEX LOADER will have to completely rewrite the program, work out all the jumps, and re-enter it in its entirety.

This first section will allow the program to take input from the keyboard, and the results of the program to be output to the screen. It uses the same two routines in the operating system as have been used before. Note the way the program prints out messages, and chooses which message to print. This is analysed later.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

AAB4                 10         ORG  43700
AAB4                 20         ENT  43700
AAB4   CD18BB        30 START   CALL 47896
AAB7   0604          40         LD   B,4
AAB9   FEFC          50         CP   252
AABB   C8            60         RET  Z
AABC   FE80          70         CP   128
AABE   3016          80         JR   NC,NOTASC
AAC0   FE41          90         CP   65
AAC2   3811         100         JR   C,NOTLET
AAC4   2811         110         JR   Z,ISA
AAC6   FE7B         120         CP   123
AAC8   300B         130         JR   NC,NOTLET
AACA   FE5B         140         CP   91
AACC   3806         150         JR   C,ISLET
AACE   FE61         160         CP   97
AAD0   3803         170         JR   C,NOTLET
AAD2   2803         180         JR   Z,ISA
AAD4   05           190 ISLET   DEC  B
AAD5   05           200 NOTLET  DEC  B
AAD6   05           210 NOTASC  DEC  B
AAD7   21EDAA       220 ISA     LD   HL,MESST
AADA   3E0A         230         LD   A,#0A
AADC   BE           240 LOOKMS  CP   (HL)
AADD   23           250         INC  HL
AADE   20FC         260         JR   NZ,LOOKMS
AAE0   10FA         270         DJNZ LOOKMS
AAE2   7E           280 PRINT   LD   A,(HL)
AAE3   CD5ABB       290         CALL 47962
AAE6   FE0A         300         CP   #0A
AAE8   28CA         310         JR   Z,START
AAEA   23           320         INC  HL
AAEB   18F5         330         JR   PRINT
AAED   0A           340 MESST   DEFB #0A
AAEE   41204C45     350         DEFM "A LE"
AAF2   54544552     360         DEFM "TTER"
AAF6   20425554     370         DEFM " BUT"
AAFA   204E4F54     380         DEFM " NOT"
AAFE   2041         390         DEFM " A"
AB00   0D0A         400         DEFW #0A0D
AB02   4E4F5420     410         DEFM "NOT "
AB06   41204C45     420         DEFM "A LE"
AB0A   54544552     430         DEFM "TTER"
AB0E   0D0A         440         DEFW #0A0D
AB10   4E4F5420     450         DEFM "NOT "
AB14   41534349     460         DEFM "ASCI"
```



```
AB18  49          470         DEFM "I"
AB19  0D0A        480         DEFW #0A0D
AB1B  594F5520    490         DEFM "YOU "
AB1F  50524553    500         DEFM "PRES"
AB23  53454420    510         DEFM "SED "
AB27  4121        520         DEFM "A!"
AB29  0D0A        530         DEFW #0A0D

Pass 2 errors: 00
```

Figure 7.1

You will see that there is no separate Hex listing given for the program in Fig. 7.1 and none of the programs in the remainder of this book will have them. By this stage you should be familiar enough with the HEX LOADER to be able to use the Hex listing from the assembler. This is the column one from the left, starting – in this case – CD18BB.

As usual, if you are using the HEX LOADER you enter the program in pairs of Hex numbers. The addresses for SET MEMORY, the START ADDRESS and the checksums are given below.

SET MEMORY TO AAB3
START ADDRESS AAB4
Checksums; 05EA, 0380, 036C, 023A, 0567, 0395, 02DB, 0226, 02A5, 0248, 0264, 01C8

If you are using the assembler, there is no need to split the messages into small blocks. This has been done because the assembler only gives a Hex listing of the first four bytes on each line. Line 350 could therefore be

DEFM "A LETTER BUT NOT A"

and lines 360 to 390 inclusive would not be needed. The same applies to the other messages.

There are a number of interesting points in this program which are worth explanation.

With almost any machine code program, once it is running there is no way of stopping the program unless you have given an escape route. The program above goes round in a loop. When called from BASIC by a 'CALL 43700' command, or from the assembler by R, the program would continue giving its assessment of keys pressed on the keyboard until Doomsday, a breakdown, or you reset or switched off the computer.

A means of escape has therefore been provided. The first check the program makes on the code returned in the A register, from the WAIT KEY routine at 47896 is to see whether it holds 252, the code returned by the red ESC key, if so a RETurn to the CALLing program is made.

The next section of the program has already been explained, and the four labels ISLET, NOTLET, NOTASC and ISA have now been included. There are four messages that are printed according to the assessment of the code in the A register, and each message has been given a number, 1 to 4. The B register is loaded with 4 at the start of each loop round the program, and is decremented according to which label the program jumps to. Should it jump directly to the label ISA then the B register will be left holding four, but when the jump is to the label ISLET, the B register will be decremented three times, and hold 1 when the program arrives at the label ISA. This is used to decide which of the messages is to be printed.

The message table starting at the label MESST is then scanned, and the B register decremented each time a byte holding 0Ah is found until B holds 0. Rather than use two instructions, the semi-automated DJNZ instruction is employed.

This instruction is constructed in exactly the same manner as a JR NZ instruction, but it is the first of many Z80 instructions yet to be introduced, that do the combined jobs of more than one normal instruction.

The DJNZ instruction performs the equivalent of a DEC B followed by a JR NZ instruction, but with a saving of 1 byte, and without affecting any flags.

ASSEMBLER	DECIMAL	HEX	BINARY
DJNZ n	16 n	10 n	00 001 010 n

As usual with a relative jump n is the jump distance, in 2s complement, from the address of the next instruction.

Once the B register is decremented to 0, the program continues and the message is printed from the byte following that holding the 0Ah which caused B to reach 0, up to and including the end of the message, marked by the next 0Ah. You will notice that each end marker (0Ah) is preceded by a byte holding 0Dh. The combination of these perform a carriage return and a line feed, positioning the cursor ready for the next message. Control is then returned to the start of the program and the process is repeated for the next key pressed.

The next flag to be examined is the sign flag, which indicates whether the sign of the result of a maths operation is Plus or Minus. This flag, and the Parity/Overflow flag, whose function will be explained next, cannot be used in connection with a relative jump (JR) instruction. It is time therefore to detail all the instructions which can be made conditional upon the setting of a flag.

So far, with the exception of one instruction in the program in Fig. 7.1, the only conditional branches have been made by relative jumps. The full opcodes for both unconditional and conditional relative jumps are shown in Fig. 7.2.

ASSEMBLER	DECIMAL	HEX	BINARY
DJNZ n	16 n	10 n	00 010 000 n
JR n	24 n	18 n	00 011 000 n
JR NZ,n	32 n	20 n	00 100 000 n
JR Z,n	40 n	28 n	00 101 000 n
JR NC,n	48 n	30 n	00 110 000 n
JR C,n	56 n	38 n	00 111 000 n

Figure 7.2

As you might expect an absolute jump JP can also be made conditional on the setting of flags, as can the CALL and RET instructions. These instructions are not limited like the relative jumps, to using only the Carry and Zero flags. They can use any of the user accessible flags.

You will remember that each of the general purpose registers has a three bit code, used to identify it in all the instructions which can use any general purpose register, by changing three bytes in the instruction, according to which register is to be used. The same system is employed by the Z80 to identify conditions.

NZ	(not zero)	000
Z	(zero)	001

NC	(no carry)	010
C	(carry)	011
PO	(parity odd)	100
PE	(parity even)	101
P	(sign positive)	110
M	(sign negative)	111

In each of the instructions below, cc represents the condition, selected from those above, upon which the program is to branch. The three bits denoted cc in the binary instruction, comprise three bits from above according to the condition chosen.

```
ASSEMBLER          BINARY

JP    cc,nn        11 cc 010   nn

CALL  cc,nn        11 cc 100   nn

RET   cc           11 cc 000
```

JP NC,47962 therefore becomes 11 010 010 0101 1010 1011 1011
and CALL Z,47960 will be 11 001 100 0101 1000 1011 1011.

Obviously the sign flag can only indicate the sign of a result correctly when 2s complement notation is being used. The flag is meaningless as a sign flag when computations are in unsigned binary, although it can still be useful as a test of bit 7. For example: the A register contains a positive value of 254 after a maths operation. The sign flag however will be set, erroneously indicating a negative result. This is because the sign flag simply reflects the state of bit 7 of the result. In future, instead of making references to 2s complement notation, which is rather long winded, the term 'signed' will be employed. This will reflect that the sign flag correctly indicates the sign of the result of a maths operation.

The sign flag is affected by *all* 8 bit maths operations, including CP (compare), 8 bit INC and DEC instructions, and the 16 bit ADC and SBC instructions. None of the other instructions you have learnt so far affects it in any way. As new instructions are annotated their effect on flags will be detailed where it is useful, and the effect of *all* instructions on any flags is shown in the appendix of opcodes.

The last flag that is of use to the programmer is the Parity/ Overflow flag. This flag has two distinct uses and cannot be used for both purposes at the same time. It is *either* an overflow flag *or* a parity flag, *never* both.

All instructions that affect the Zero flag also affect the P/V flag, and all the instructions explained in this book so far that affect the Zero flag use the P/V flag in its role as an overflow flag.

The Overflow flag indicates that the execution of a signed calculation has caused the result to exceed the range that can be held in signed form. Confused? Not surprising, this is possibly the most complicated concept so far, but once you get the hang of signed overflows you will wonder what all the trouble was about. Consider the following short program:

```
LD      A,−80
ADD     A,−80
```

On completion the A register will hold 0110 0000 in binary, which is 96 or 60h. This is a positive number (bit 7 reset) and not the correct answer. In this example the carry flag will be set, allowing you to pick up the fact that the result caused an overflow. But what about the sum:

```
LD      A,80
ADD     A,80
```

This time the A register will hold the answer 1010 0000 in binary, which is −96. Again not the correct answer but this time the carry flag will not be set. Ostensibly, without knowing instinctively, there is no indication of the sum going wrong. This is where the Overflow flag comes in. Any maths operation that causes a result outside the arithmetic range for the instruction, that is $-128 <= n <= 127$ for an 8 bit operation, or $-32768 <= nn <= 32767$ for a 16 bit operation will cause the overflow flag to be set.

To test the overflow flag the mnemonics are PE and PO, a jump will be made by PO if there has been an overflow, and by PE if there has not. In fact PE stands for Parity Even and PO stands for Parity Odd, and are used here, because no additional mnemonics have been assigned to the flag, to separate its use as an overflow flag and a parity indicator. It may help to remember which mnemonic to use if you note that the mnemonic which will cause a branch if there is an overflow is the only one to contain the letter O.

No overflow can be caused by adding two numbers with different signs, and only numbers with different signs can cause an overflow in subtraction.

Parity is determined by the number of bits in a byte that are set to 1; if there is an even number then parity is said to be even. The P/V flag, after an instruction which uses it as a parity flag, will indicate the parity of the byte tested. The flag is set if the parity is even and reset if the parity is odd. None of the instructions introduced so far uses the P/V flag in its parity role. It will be pointed out when a new instruction employs the P/V flag as a parity flag.

There are two further instructions to be considered in this chapter, SCF and CCF.

These are both completely straightforward in operation. SCF is short for Set Carry Flag, and that is exactly what it does when executed. CCF is short for Complement Carry Flag, and again that is what it does. No, *not* by telling it what a nice carry flag it is, but by changing its state. If the carry flag was set before the execution of a CCF instruction it would be reset afterwards, and the converse if the flag was reset before the CCF instruction.

The opcodes are as follows:

ASSEMBLER	DECIMAL	HEX	BINARY
CCF	63	3F	00 111 111
SCF	55	37	00 110 111

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number
nn = a 16 bit number
() round a number or register pair = contained in
cc = a condition
PC = Program Counter
SP = Stack Pointer

The usable flags are CARRY ZERO SIGN and P/V.

The P/V flag has two separate uses, Parity and Overflow.

Overflow signals that the sign has changed on a signed arithmetic operation, making the result wrong.

cc is C, NC, Z, NZ, PE, PO, M and P.

CP performs a dummy SUB on the A register, it sets flags, but does not alter anything else.

JR can only be conditional on the Carry flag or the Zero flag.

DJNZ does the equivalent of a DEC B and a JR NZ, but does not affect any flag at all.

CALL JP and RET can all be made dependent on any usable flag.

No LD CALL JP JR or RET instruction affects any flag.

Chapter Eight

Logical Operations

The Z80 CPU is endowed with a set of logical operators almost identical to that possessed by the Amstrad CPC 464's BASIC. This is not really surprising, since it is the Z80 which does all the work of running BASIC. This makes the job of explaining the machine code instructions AND OR and XOR much easier, since you are no doubt already familiar, albeit unknowingly, with the way they function, excepting their effect on flags. If you have not yet become acquainted with these operators on the Amstrad, then turn to Chapter 4 page 18 in the Amstrad User Instructions, where you can read about them. The remainder of this chapter assumes a working knowledge of Amstrad BASIC logical expressions.

The AND OR and XOR logical operators are classed as mathematical, and can only be performed on 8 bit values using the A register. If you look at the opcodes below, you will find that they are made up identically to the other 8 bit maths instructions, with bits 5, 4 and 3 changed according to the instruction. The mnemonics do not require the A register to be stipulated since, as with the SUB mnemonic, there is no other register which can employ them.

The flags are affected by all the logical operators, and are set according to the contents of the A register after execution. Obviously, since no result from an AND OR or XOR can ever cause a result outside the range of 8 bit numbers all these instructions reset the carry flag to 0. And because it would be impossible to cause an overflow the P/V flag takes on its Parity role. The Sign flag will reflect the state of bit 7 in the A register after the operation, and the Zero flag will be set if the A register has no bits set otherwise it will be reset.

Before being able to make efficient use of these logical operators you will have to start thinking in binary. Only then do the many and varied uses become apparent. At present it is most unlikely

ASSEMBLER	DECIMAL	HEX	BINARY
AND n	230	E6	11 100 110
AND r	160 - 167	A0 - A7	10 100 r
XOR n	238	EE	11 101 110
XOR r	168 - 175	A8 - AF	10 101 r
OR n	246	F6	11 110 110
OR r	176 - 183	B0 - B7	10 110 r

that you are thinking in binary, so you probably can't begin to think what purposes these may be.

Consider the program in Fig. 7.1. Here tests had to be made separately for lower and upper case letters, and for the gaps between the two types of letters. But in fact the only difference (in binary) between the two sets of letters is the condition of bit 5. All upper case letters have bit 5 reset, and all lower case letters have bit 5 set. Using the logical operator *and* it is possible to make a lower case letter upper case, and by use of the logical operator *or* an upper case letter can be made lower case. Can you work out the full form of the instruction?

With the alterations to the program in Fig. 7.1 given below, the answer to the question above can be demonstrated.

Change line 220 to:

```
ADDRESS   HEX          ASSEMBLER

AAD7      CD 2B AB     CALL EXTRA    Checksum for this is 01A3.

Add the following at the end of the program

AB2B      CD 5A BB     CALL 47962

AB2E      00           NOP

AB2F      00           NOP

AB30      CD 5A BB     CALL 47962

AB33      3E 20        LD   A,32   ; THE CODE FOR SPACE

AB35      CD 5A BB     CALL 47962

AB38      21 ED AA     LD HL,MESST

AB3B      C9           RET
```

The checksums required by the HEX LOADER for this second section are: 0422, 0463.

Now when the program is executed the character generated by the code returned by the key you press will appear twice, followed by a space, before the program gives its verdict. The two NOPs give space for you to put an AND OR or XOR instruction, and then see the effect it has.

First change the two NOPs to:

```
ADDRESS   HEX         ASSEMBLER

AB2E      F6 20       OR #20
```

The easiest way to do this if you are without an assembler is to type POKE &AB2E,&F6: POKE &AB2F,&20 as a direct command. Now execute the program again and try pressing various keys, both with and without the shift key. (Remember to make sure that the CAPS LOCK is not on . . .. Why oh why couldn't Amstrad have put a light in the caps lock key to indicate when it was in use?)

You will find that all upper case letters are changed to lower case, numbers are left alone, as are lower case letters, and other codes may or may not be changed, according to whether they had bit 5 set originally. By incorporating this OR #20 instruction in the main program, a lot of the tests using CP can be made redundant. The revised version of the program in Fig. 7.1 is given in Fig. 8.1. You will see that the sign flag has been used to indicate when the code is not ASCII (bit 7 set, that is the value in A is 128 or over). The sign flag couldn't be used before because it would not have given an indication of bit 7 of the code of the key pressed without a CP 0 instruction, and this would have added a byte to the length of the program, because the sign flag cannot be tested by a JR instruction. Now that the logical operator is being used the test instruction that was necessary in the original program can be thrown out, giving a net saving of one byte after the JR is replaced by the JP.

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                          1 ; FIG 8,1
                          2 ; AMMENDED VERSION OF PROGRAM IN FIG 7,1
AAB4                     10         ORG   43700
AAB4                     20         ENT   43700
```

```
AAB4  CD18BB       30 START  CALL 47896
AAB7  0604         40        LD   B,4
AAB9  FEFC         50        CP   252
AABB  C8           60        RET  Z
AABC  F620         90        OR   #20
AABE  FACDAA      100        JP   M,NOTASC
AAC1  FE7B        120        CP   123
AAC3  3007        130        JR   NC,NOTLET
AAC5  FE61        160        CP   97
AAC7  3803        170        JR   C,NOTLET
AAC9  2803        180        JR   Z,ISA
AACB  05          190 ISLET  DEC  B
AACC  05          200 NOTLET DEC  B
AACD  05          210 NOTASC DEC  B
AACE  21E4AA      220 ISA    LD   HL,MESST
AAD1  3E0A        230        LD   A,#0A
AAD3  BE          240 LOOKMS CP   (HL)
AAD4  23          250        INC  HL
AAD5  20FC        260        JR   NZ,LOOKMS
AAD7  10FA        270        DJNZ LOOKMS
AAD9  7E          280 PRINT  LD   A,(HL)
AADA  CD5ABB      290        CALL 47962
AADD  FE0A        300        CP   #0A
AADF  28D3        310        JR   Z,START
AAE1  23          320        INC  HL
AAE2  18F5        330        JR   PRINT
AAE4  0A          340 MESST  DEFB #0A
AAE5  41204C45    350        DEFM "A LE"
AAE9  54544552    360        DEFM "TTER"
AAED  20425554    370        DEFM " BUT"
AAF1  204E4F54    380        DEFM " NOT"
AAF5  2041        390        DEFM " A"
AAF7  0D0A        400        DEFW #0A0D
AAF9  4E4F5420    410        DEFM "NOT "
AAFD  41204C45    420        DEFM "A LE"
AB01  54544552    430        DEFM "TTER"
AB05  0D0A        440        DEFW #0A0D
AB07  4E4F5420    450        DEFM "NOT "
AB0B  41534349    460        DEFM "ASCI"
AB0F  49          470        DEFM "I"
AB10  0D0A        480        DEFW #0A0D
AB12  594F5520    490        DEFM "YOU "
AB16  50524553    500        DEFM "PRES"
AB1A  53454420    510        DEFM "SED "
AB1E  4121        520        DEFM "A!"
AB20  0D0A        530        DEFW #0A0D

Pass 2 errors: 00

Table used:   110  from   184
Executes: 43700
```

Figure 8.1

Checksums: 0582, 05B8, 0215, 04B6, 0439, 02A7, 022B, 02A2, 0251, 0268, 020D, 0608, 0278

The AND instruction could have been used in place of the OR changing lower case to upper case instead, and the appropriate modifications made to cater for this. In this case AND #DF would remove any bit 5 that was set.

If the OR in the program in Fig. 8.1 is changed for XOR and the changes given earlier to display the character represented by the code both before and after the logical operation are incorporated, you will find that upper case is changed to lower case and vice versa. Be careful not to press any non-alphabetic keys, as the XOR will make some of these into control codes.

The AND instruction is usually employed to 'mask' bits. This is the term used when a bit or bits are ignored, or made insignificant. For example, if a program required each letter of the alphabet to be represented by a number, with 'A' as 1 through to 'Z' as 26, without differentiating between upper and lower case. Here the easiest solution would be to mask the top three bits of the letter's code with an AND %00011111.

The OR instruction has the opposite effect, and could be used to reclaim bits masked out by an AND. One of the more common uses is to allow any writing to the screen to be carried out in 'over' form, that is, bits from whatever previously occupied a character square are only altered when they are overwritten. Another common use is to reclaim masked bits or modify values. For example, the programs used to print numbers held in registers or memory, which started with the program in Fig. 6.5 and developed to the program in Fig. 6.14, all used the instruction ADD A,#30 to transform a number into its ASCII code; this was in fact doing the same operation as an OR #30, which is what would have been used had you known the instruction at the time. It would not save any memory but does make what is happening much clearer.

The XOR instruction, like the OR, is often used for screen-based operations, and your Amstrad uses it for 'Transparent' mode printing (see Chapter 5, page 2 in the Amstrad User Instructions). It is also often used when a bit or bits are to be changed to the opposite of what they were.

Next in the list of logical operators is the complement instruction, which has the mnemonic CPL, and again there is a direct equivalent in Amstrad BASIC. This instruction does the same job as the BASIC NOT. As with all the instructions in this chapter it operates on the A register only, and simply changes every bit to the opposite state. In fact the result is identical to that achieved by a XOR#FF.

A graphic demonstration of the CPL in use can be given by the program shown in Fig. 8.2, which complements all the bits in the screen map (the area of memory in which all the information about what is to be displayed on the screen is held). This will invert the bits for both paper and pen and whilst in mode 2 this will cause the screen to become a negative of its former self.

In modes other than 2 things are a bit more tricky. This is because there are more than two colours available, and whilst in mode 2, paper is set to 0 and pen to 1, which is differentiated by 1 bit in a byte; and 1 byte can therefore control eight screen pixels, each bit being 1 for a pixel set to ink 1 and 0 for a pixel set to ink 0. Inverting these bits with a CPL instruction will therefore make all ink 1 ink 0 and all ink 0 ink 1.

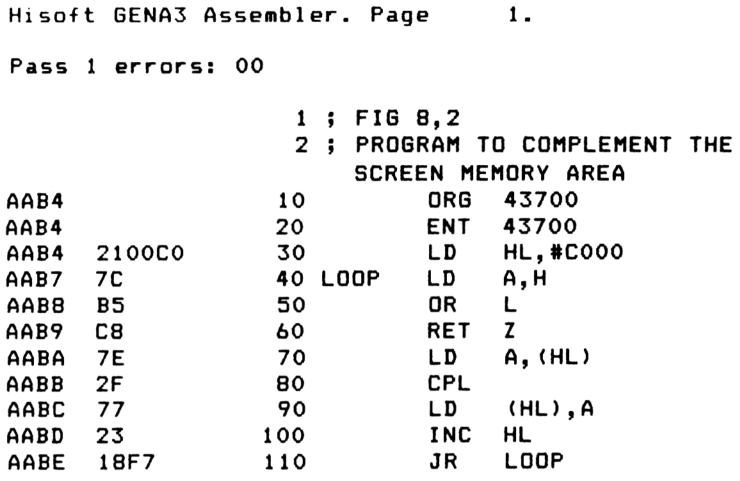
In mode 1 there are four colours to be differentiated between, and this necessitates two bits to dictate the colour for each pixel, 00 for ink 0, 01 for ink 1, 10 for ink 2 and 11 for ink 3. Each byte can therefore only control four pixels. If the PEN colour is set to 1, and the PAPER colour to 0, after complementing the bits PAPER will be 3 and PEN 2.

In mode 0 things become even worse: there are sixteen colours to cope with, and now each pixel requires four bits, so one byte can only control two pixels.

You have just found out (if you didn't already know) why the resolution goes down as the colours go up. Unfortunately the bits in a byte only correspond to the order you would expect in mode 0, in other modes they are a bit mixed up, and the pun is *most definitely* intended. In mode 1, for instance, bits 3 and 7 control the left-most pixel of the four controlled by a byte, bits 2 and 6 the next, bits 1 and 5 the one from right-most and bits 0 and 4 the right-most.

To make matters worse, even the order of the bytes which control the screen is not what you would expect (unless you have a decidedly odd mind!). Details of both the byte order and the bit order for each mode are given in the screen map in the appendix.

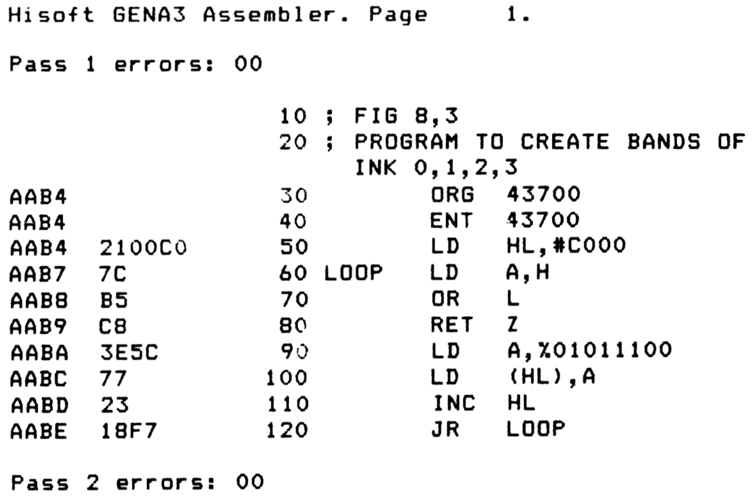
The program given in Fig. 8.3 manipulates the screen in mode 1, changing the screen to a series of one-pixel-wide columns of INK 0, INK 1, INK 2, and INK 3 when executed. Notice that the bits for the ink colours are stored with the more significant bit in the less significant bit position. Whoever thought out this screen map must be a sadist!



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                          1 ; FIG 8,2
                          2 ; PROGRAM TO COMPLEMENT THE
                              SCREEN MEMORY AREA
AAB4                     10          ORG   43700
AAB4                     20          ENT   43700
AAB4   2100C0            30          LD    HL,#C000
AAB7   7C                40 LOOP     LD    A,H
AAB8   B5                50          OR    L
AAB9   C8                60          RET   Z
AABA   7E                70          LD    A,(HL)
AABB   2F                80          CPL
AABC   77                90          LD    (HL),A
AABD   23               100          INC   HL
AABE   18F7             110          JR    LOOP

Pass 2 errors: 00
```

THE CHECKSUMS REQUIRED BY THE HEX LOADER ARE
0421, 010F

Figure 8.2



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         10 ; FIG 8,3
                         20 ; PROGRAM TO CREATE BANDS OF
                              INK 0,1,2,3
AAB4                     30          ORG   43700
AAB4                     40          ENT   43700
AAB4   2100C0            50          LD    HL,#C000
AAB7   7C                60 LOOP     LD    A,H
AAB8   B5                70          OR    L
AAB9   C8                80          RET   Z
AABA   3E5C              90          LD    A,%01011100
AABC   77               100          LD    (HL),A
AABD   23               110          INC   HL
AABE   18F7             120          JR    LOOP

Pass 2 errors: 00
```

THE CHECKSUMS REQUIRED BY THE HEX LOADER ARE
040E, 010F

Figure 8.3

The CPL instruction does not affect any of the testable flags.

The last of the logical operators is the negate opcode, NEG in assembler mnemonics, and in CB parlance as well. This instruction is the simplest of the logical operators. It takes the value in the A register and changes its sign, by taking its twos complement. In other words A becomes 0 minus A.

The NEG instruction affects the flags exactly as if a normal SUB had been executed. Carry, Zero, Sign and P/V flags are all affected, and the P/V flag is used in the Overflow mode.

The opcodes are:

ASSEMBLER	DECIMAL	HEX	BINARY
NEG	237 68	ED 44	11 101 101 01 000 100
CPL	47	2F	00 101 111

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number
nn = a 16 bit number
() round a number or register pair = the address at
PC = Program Counter
SP = Stack Pointer

All the logical instructions work on the value in the A register.

AND OR and XOR can all be used with r or n.

AND. Bits set in both the A register and the operand before execution remain set in the A register after execution, all other bits in the A register are reset.

OR. Bits set in either the A register OR the operand before execution are set in the A register after execution.

XOR. Bits that were set in either the A register OR in the operand, but not in both, before execution are set in the A register after execution.

All the above instructions reset the carry flag, and affect the remaining flags according to the result in the A register. The P/V flag is used to test parity.

CPL and NEG do not need an operand.

CPL flips the bits in the A register, and does not alter any of the testable flags.

NEG returns the twos complement of the value in the A register. Flags are affected as by a SUB instruction.

Chapter Nine

Using the Machine Stack

The machine stack has already been mentioned briefly, in Chapter 5, where it was explained how a CALL instruction placed the return address on the stack, for collection later by a RETurn instruction. The importance of keeping a balance, between the number of things pushed onto the stack and popped off again, was stressed, and it was shown that, if a RETurn was to be made to the correct address, the next value to come off the stack must be the value pushed onto it by the CALL which it is desired to RETurn from.

To complicate matters there are also instructions which allow the programmer to use the machine stack as a temporary store, in the same way as the CALL instruction temporarily stores the return address ready for collection by the associated RETurn. Though it *may* not be totally disastrous if a RETurn is made to the wrong RETurn address, a RETurn made to what the processor thinks is a return address, but which is in reality just a number which is stored on the stack, will almost certainly crash the system.

The above paragraphs may seem to be belabouring the point, but an imbalance of the stack is the single most common cause for machine crashes, and even the most experienced programmers sometimes get caught out. This is also the prime reason for making sure that you SAVE any machine code program before you try to run it. At least you won't then have to start completely from scratch if it crashes.

You have probably realised what the mnemonics are for pushing data onto the stack and popping it off again, by the slightly strange choice of verbs used. The two instructions are:

```
ASSEMBLER    BINARY

PUSH rr      11 rr0 101

POP  rr      11 rr0 001
```

As usual with instructions where rr is specified, any pair of general purpose registers can be used, and the binary codes to replace rr for each of the register pairs are, as you should know by now:

BC = 00   DE = 01   HL = 10

Additionally, with the PUSH and POP instructions you can save the A register and the flag register onto the stack, and POP them off again. The only remaining code, 11, is used to denote that AF is to be used.

When a PUSH is executed the contents of the register pair nominated are copied into the next positions on the stack, and the stack pointer is decremented by two, to point at the new bottom of the stack. A POP does the reverse, copying the contents of the top of the stack into the nominated register pair and incrementing the stack pointer twice. This was shown for the CALL and RET instructions, in Fig. 5.9. Exactly the same process occurs when the stack is used by the PUSH and POP instructions, but because instead of the program counter being POPped or PUSHed off or on the stack, normal registers are used. The program does not jump.

It is interesting to note that the binary instructions for CALL and RET are almost identical to PUSH and POP, which is what you might have expected, knowing how they use the stack.

(CALL 11 001 101   RET 11 001 001)
PUSH 11 rr0 101   POP 11 rr0 001)

A program which will show the last thing PUSHed onto the stack and the address pointed to by the stack pointer is given overleaf.

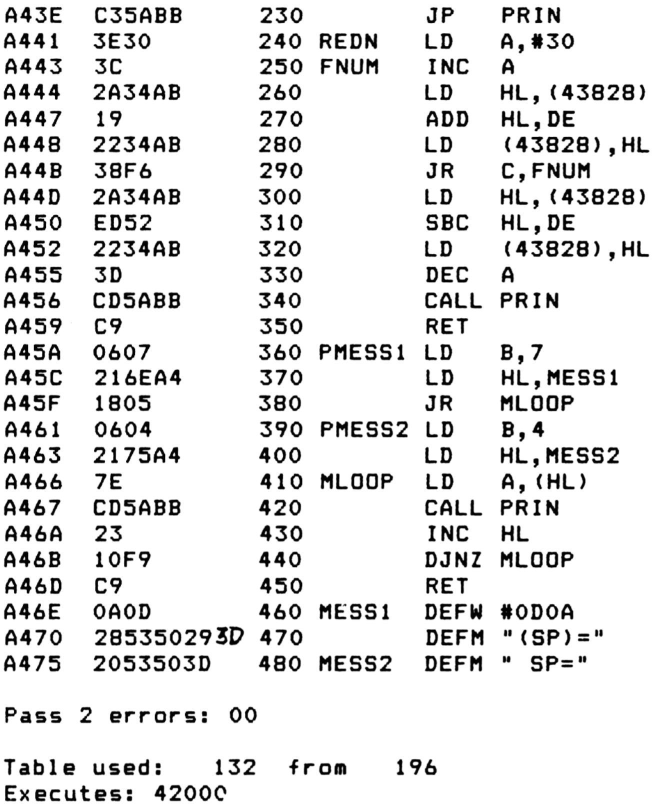
The checksums required by the HEX LOADER are as follows:

0581, 05B6, 0561, 0580, 04F9, 02C7, 047C, 0403, 03A9, 0300, 013D

The majority of this program will be familiar to you, so there is no need to explain that part, but you will see that a few changes have been made. Instead of the A register counting from zero, and having to be altered to hold the ASCII code of the number to be printed, it now starts by holding #30. The only time that it is necessary to alter it is when the remainder from the subtractions is printed, and this is now achieved by an OR.

One new instruction has been introduced, in line 110, but you will know what this instruction does from the mnemonic. One interesting point here, that you may have noticed, is that the binary instruction for LD (nn),SP is 1110 1101 01 110 011 n n, which fits neatly into the set of LD (nn),rr instructions listed in Fig. 5.8. The two bit code for the 16 bit register pair, SP is 11, and this holds true for the LD rr,(nn) instruction as well. So what, you may ask, is the last remaining two bit code, 10 used for? It would be logical if it stood for the HL register pair, as it does in all other cases, but there is already an instruction to LD HL,(nn) and to LD (nn),HL.

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        1 ; FIG 9,1
                        2 ; PROGRAM TO SHOW WHERE THE STACK
                            POINTER
                        3 ; IS POINTING AND THE VALUE THAT
                            WILL BE
                        4 ; ACCESSED BY THE NEXT RETRIEVAL
                            FROM THE STACK
A410                   10         ORG   42000
A410                   20         ENT   42000
BB5A                   30 PRIN    EQU   47962
A410  E1               60 PROG1   POP   HL
A411  E5               70         PUSH  HL
A412  2234AB           80         LD    (43828),HL
A415  CD5AA4           90         CALL  PMESS1
A418  CD22A4          100         CALL  PROG2
A41B  ED7334AB        110         LD    (43828),SP
A41F  CD61A4          120         CALL  PMESS2
A422  11F0D8          130 PROG2   LD    DE,-10000
A425  CD41A4          140         CALL  REDN
A428  1118FC          150         LD    DE,-1000
A42B  CD41A4          160         CALL  REDN
A42E  119CFF          170         LD    DE,-100
A431  CD41A4          180         CALL  REDN
A434  1EF6            190         LD    E,-10
A436  CD41A4          200         CALL  REDN
A439  3A34AB          210         LD    A,(43828)
A43C  F630            220         OR    #30
```



```
A43E  C35ABB      230          JP   PRIN
A441  3E30        240 REDN     LD   A,#30
A443  3C          250 FNUM     INC  A
A444  2A34AB      260          LD   HL,(43828)
A447  19          270          ADD  HL,DE
A448  2234AB      280          LD   (43828),HL
A44B  38F6        290          JR   C,FNUM
A44D  2A34AB      300          LD   HL,(43828)
A450  ED52        310          SBC  HL,DE
A452  2234AB      320          LD   (43828),HL
A455  3D          330          DEC  A
A456  CD5ABB      340          CALL PRIN
A459  C9          350          RET
A45A  0607        360 PMESS1   LD   B,7
A45C  216EA4      370          LD   HL,MESS1
A45F  1805        380          JR   MLOOP
A461  0604        390 PMESS2   LD   B,4
A463  2175A4      400          LD   HL,MESS2
A466  7E          410 MLOOP    LD   A,(HL)
A467  CD5ABB      420          CALL PRIN
A46A  23          430          INC  HL
A46B  10F9        440          DJNZ MLOOP
A46D  C9          450          RET
A46E  0A0D        460 MESS1    DEFW #0D0A
A470  285350293D  470          DEFM "(SP)="
A475  2053503D    480 MESS2    DEFM " SP="

Pass 2 errors: 00

Table used:    132  from    196
Executes: 42000
```

Figure 9.1

Computers are logical – it does represent the HL register pair – even though there is a shorter instruction that does an identical job! (logical?) You can test this very easily if you are using an assembler, change line 80 to read DEFB #ED and add the following lines:

81 DEFB %01100011
82 DEFB #34
83 DEFB #AB

Now re-assemble the program and execute it, you will find that it operates exactly as before.

The program operates by POPping the value off the top of the stack (yes it's called the top even though the top is at a lower address than the bottom) into the HL register pair. This moves the stack pointer to point at the previous item on the stack so, to avoid altering the stack, the HL register pair is then PUSHed back. The stack is now in exactly the same state as it was at the start of the program, but the HL register pair now holds a copy of the value on the top of the stack.

HL is then loaded into memory at address 43828 and the subroutine to print message 1 is called, then the subroutine to print a number (named PROG2 here) is called, and this will print the value copied from the top of the stack into memory. There have now been two CALLs and two RETurns, so the stack pointer is pointing to the same address as it was at the start of the program. It is this address which is now loaded into memory ready to be printed by the print number subroutine. First PMESS2 is CALLed to print message 2 and then the print number subroutine is executed again. This time the print number subroutine is not CALLed, so the RET at the end will return to BASIC or the assembler according to how PROG1 was first accessed.

A dramatic demonstration of how the stack can be manipulated to your advantage can be made by adding the following lines at the start of the program in Fig. 9.1. If you are using the HEX LOADER set memory down to 41992 and use 41993 as the start address.

```
A409                   5       ORG   41993
A409                   6       ENT   41993
A409   2110A4          7       LD    HL,PROG1
A40C   E5              8       PUSH  HL
A40D   E5              9       PUSH  HL
A40E   E5             10       PUSH  HL
A40F   E5             20       PUSH  HL
```

Re-assemble the program again if you are using the assembler, and then execute it. If you used the HEX LOADER note that the program now starts at A409 (41993) and no longer at A410 (42000).

This time the program will loop round five times; the extra four times are because of the PUSHes made onto the stack, which cause the RETurns to be made to PROG1 until the original return address surfaces.

The instructions detailed so far are the only ones that automatically update the stack pointer when the stack is used. There are however a number of instructions which allow the stack to be manipulated, and information to be passed to and from it.

The first of the remaining opcodes which employ the stack or the stack pointer to be considered are the LD instructions; this is because they are the most straightforward, not in their operation, because all the instructions on the Z80 should be fairly easy for you to understand now, but in the uses to which they may be put.

When you first turn-on your Amstrad CPC 464, part of the cold start procedure initialises the stack pointer to an address in high memory, address 49144 (BFF8h), and it is from here that the stack grows down. It is quite possible that this address will be acceptable for the bottom of the stack, and will not need to be changed. There are, none the less, circumstances when it can be beneficial, or even essential, either to change the stack pointer or save it somewhere. Instructions are therefore provided to allow this, one of which you have already used.

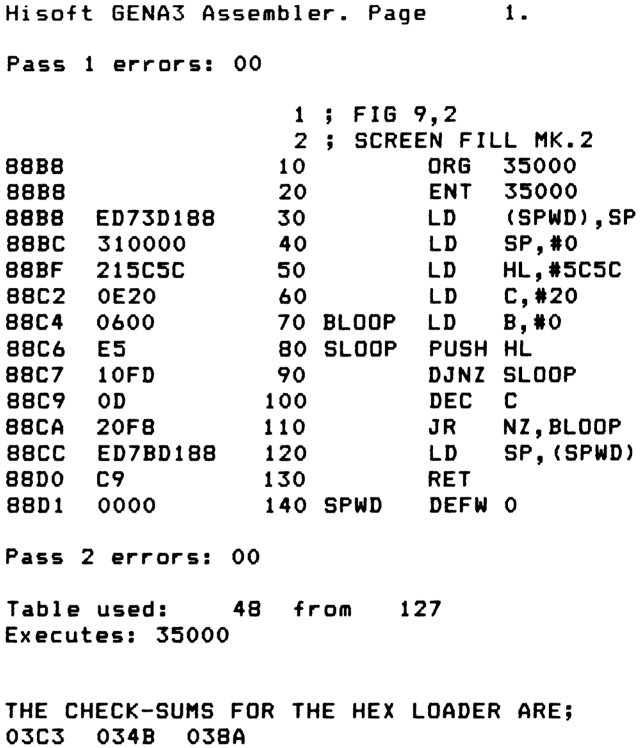
Please note that it is important to ensure that the stack pointer is always initialised to point to an even numbered address, particularly on the Amstrad, where banks of memory can be switched. If this is not done it would be possible for half a stack item to be switched out, and the other half remain. It is best to initialise it to point to an address which is a multiple of 256 as this allows the maximum downward growth before a memory page barrier is transversed.

All the 16 bit LD instructions can be used with the stack pointer by making bits 5 and 4 11. The full range of normal LD instructions is:

ASSEMBLER	HEX	BINARY
LD SP,nn	31 n n	00 110 001 n n
LD SP,(nn)	ED 76 n n	11 101 101 01 111 011 n n
LD (nn),SP	ED 73 n n	11 101 101 01 110 011 n n

Occasions where you may have to move the stack pointer could be, for example: when there is an instruction which has priority over anything else which may be going on in a program. The RESET achieved by pressing the [CONTROL] [SHIFT] and [ESC] keys on the Amstrad, is a good instance of this. When a priority instruction is executed there can be no possibility of even making sure that the stack is balanced, let alone knowing what is on the top, so the stack will have to be re-initialised to a known location, before any use of it can be made. Here LD SP,nn would be used.

Another good example is the program in Fig. 9.2. This is a rewrite of the program in Fig. 8.3 using the PUSH instruction to fill the screen memory area in a fraction of the time taken by the original program. Here the SP is saved to memory, for later restoration, and then loaded with address 0. The address below this will be the first to be filled by any PUSH instruction, and since one below 0 is −1, and the SP can only hold 16 bit numbers, this becomes FFFFh, the top of the screen memory area. HL is then loaded with 5C5Ch (the same as A in Fig. 8.3 but twice) ready to be used for the filling.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        1 ; FIG 9,2
                        2 ; SCREEN FILL MK.2
88B8                   10          ORG  35000
88B8                   20          ENT  35000
88B8  ED73D188         30          LD   (SPWD),SP
88BC  310000           40          LD   SP,#0
88BF  215C5C           50          LD   HL,#5C5C
88C2  0E20             60          LD   C,#20
88C4  0600             70 BLOOP    LD   B,#0
88C6  E5               80 SLOOP    PUSH HL
88C7  10FD             90          DJNZ SLOOP
88C9  0D              100          DEC  C
88CA  20F8            110          JR   NZ,BLOOP
88CC  ED7BD188        120          LD   SP,(SPWD)
88D0  C9              130          RET
88D1  0000            140 SPWD     DEFW 0

Pass 2 errors: 00

Table used:    48  from    127
Executes: 35000


THE CHECK-SUMS FOR THE HEX LOADER ARE;
03C3  034B  038A
```

Figure 9.2

Next a double loop is set up; this is a variation on a technique that is often used when something needs to be repeated more than the maximum number of times that can be held in registers for counting; it will be discussed fully in Chapter 16. Briefly what happens is that each pass round the big loop BLOOP makes a full complement of passes round the small loop SLOOP. (No, it wasn't a sailing-ship sinking!) In this case the small loop loops 256 times for each of the 32 loops round the big loop. The PUSH

HL is therefore executed 32*256 times which is 8192, and since each PUSH fills two memory locations a total of 16384 (4000h) bytes are filled. Finally the SP is restored and a RETurn made.

The SP can also be used in all the 16 bit maths instructions, as well as in 16 bit INC and DECs, again the instruction is made up by making bits 5 and 4 of the instruction 11. So for example:

ADD HL,DE is, in binary 00 011 001 and
ADD HL,SP therefore, is 00 111 001
DEC BC is 00 001 011 so DEC SP is 00 111 011

The next instruction to be considered allows the data on the top of the stack to be exchanged with the data in the HL register pair. As always the mnemonic is exactly what one might expect. It is an Exchange, so the first part of the mnemonic is EX, and the things being exchanged are (SP) and HL so the full opcode is:

```
ASSEMBLER          HEX              BINARY

EX   (SP),HL       E3               11 100 011
```

This is one of the most useful instructions for use on the stack; it can be used to redirect returns whilst within a subroutine or even to add extra subroutines.

Consider the situation where a program has been written in which part of a subroutine makes a number of 16 bit calculations which are required by the main program in a particular order. Naturally the results all end up in the HL register pair, but this is needed to perform the next calculation. So the result must be saved in memory by some means. An LD (nn),HL instruction would serve, but this uses three bytes for each instruction to save a result and another three bytes each time the result is retrieved. The easiest way to deal with this would be to save the results on the stack, but the return address from the subroutine is there, and you have been warned about the results of RETurning to a result! So what is the solution?

The answer is often to exchange the RETurn address on the top of the stack with the result of the calculation, and then PUSH the RETurn address back on the top. This takes only two bytes, and only one byte will be used to retrieve the result back in the main program. The RETurn will have removed its address already. The section of program to place results below the RETurn on the top of the stack would look like this (assuming the result to be in HL on entry to this section):

```
EX    (SP),HL ; HL NOW HOLDS THE RETURN ADDRESS AND THE

                RESULT IS ON THE TOP OF THE STACK

PUSH HL       ; AND NOW THE RETURN ADDRESS IS BACK ON

                THE TOP, WITH THE RESULT UNDERNEATH.
```

The last instruction involving the SP is a real oddball, from the CPU's point of view. It is the only instruction which permits a 16 bit register to register load. The instruction is:

```
ASSEMBLER           HEX         BINARY

LD   SP,HL          F9          11 111 001
```

This instruction is frequently used when the address to which the stack pointer should be pointing has been calculated, it saves having first to place the contents of the HL register pair into memory and then load the stack pointer from there.

There has been an awful lot to take in, in this chapter, so don't worry if your head is reeling a bit. Go over the example programs given, and try experimenting on your own. Remembering to save before you run! You can't do any harm to the Amstrad, no matter what your program does, so the worst that can happen is you will have to switch off and start again. Count the uses of the stack before executing a program, there should be an equal number of PUSHes and POPs in each section or subroutine, and every CALL must have a matching RETurn.

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number
nn = a 16 bit number
() round a number or register pair = the address at

PC = Program Counter
SP = Stack Pointer

The machine stack descends through memory as it grows.

The top of the stack is at the lowest memory address, and is pointed to by SP.

PUSH places the contents of the register pair named onto the top of the stack, and updates the SP to point to the new top.

POP does the reverse.

Any general purpose rr or the AF register pair can be PUSHed onto or POPped off the stack.

All 16 bit maths operations or LD instructions can use the SP register, as can INC and DEC.

The contents of the top of the stack can be EXchanged with the contents of the HL register pair.

The SP can be loaded direct from HL.

Every PUSH must have a corresponding POP.

Every CALL must have a RETurn.

The stack does not have to be POPped into the same register as it was PUSHed from.

Chapter Ten

Using Instructions that Work on a Single Bit

The Z80 CPU is something of a rarity in the 8 bit microprocessor field, because it permits a single bit, either in memory or in a register, to be set (made binary 1) or reset (binary 0), without affecting anything in the remainder of the byte containing the manipulated bit. There are also instructions to test an individual bit's status.

Why bother to have special instructions to do this? It is possible to set any bit you wish by an OR, or reset any bit by masking it with an AND. Equally tests can be made by these same instructions.

For example, to set bit 5 in the A register without affecting any of the other bits you would use:

```
OR %00100000
```

To reset bit 5 the instruction would be:

```
AND %11011111
```

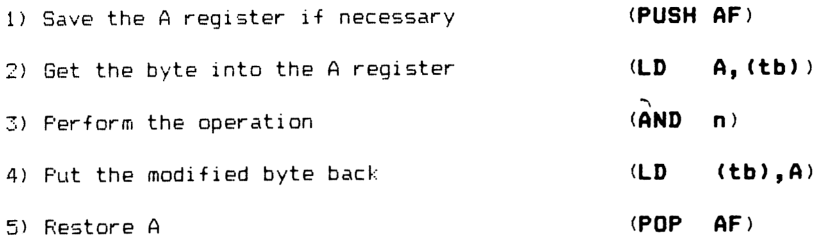
If you wanted to test bit 5 then

```
AND %00100000
```

would set the zero flag if bit 5 was zero, or reset the zero flag if bit 5 was not zero.

In all the following examples tb is the address of the byte to be tested or manipulated.

The trouble with the AND and OR instructions is that the byte to be operated on must be in the A register. This means that, if the byte was not there already, several instructions are needed in order to:



```
1) Save the A register if necessary          (PUSH AF)

2) Get the byte into the A register          (LD   A,(tb))

3) Perform the operation                     (AND  n)

4) Put the modified byte back                (LD   (tb),A)

5) Restore A                                 (POP  AF)
```

A total of 10 bytes is used for this simple task. The count could have been reduced by using the HL register pair to point to memory as follows:

```
PUSH    AF
LD      HL,tb
LD      A,(HL)
AND     n
LD      (HL),A
POP     AF
```

The byte count for the program has now dropped to 9 (big deal!). Your fingers will probably drop off too, from the amount of typing.

The program to test a bit would be similar, but the A register could not be saved by a PUSH AF because, when the A register was restored, prior to testing the zero flag to see the state of the bit tested, the flag register would be restored also, thereby destroying the flag settings from the test! The final LD (HL),A must be omitted, as the test will have modified the byte in the A register, and it must remain unaltered in memory.

An example program to test bit 5 is given below; it has a byte count of 12.

```
LD      (sb),A
LD      HL,tb
LD      A,(HL)
AND     %00100000
LD      A,(sb)
```

These programs have been given because, in spite of the fact you will never use them, they are good examples of some of the traps and pitfalls of programming, and give techniques for overcoming them. They also serve to illustrate the need for the bit test, set and reset instructions.

The bit set and reset instructions have the mnemonics SET and RES respectively. The binary opcode and assembler mnemonics are given below. In each case b should be replaced by the number of the bit to be operated on. 000 for bit 0 (the least significant) through to 111 for bit 7.

```
ASSEMBLER      BINARY

SET b,r        11 001 011   11   b   r

RES b,r        11 001 011   10   b   r
```

'r' is the usual set (000 for B 110 for (HL) 111 for A etc.). All "bit level" operations are preceded by 11 001 011 (CBh).

The full instructions to set bit 5 in the B register and to reset bit 3 in the memory location addressed by HL would therefore be:

```
ASSEMBLER      HEX        BINARY

SET 5,B        CB E8      11 001 011   11 101 000

RES 3,(HL)     CB 9E      11 001 011   10 011 110
```

None of the bit set and reset instructions affects any flag in any way.

The instruction to test a bit takes the same binary form as the SET and RESet instructions, it has the mnemonic BIT and should really be read BIT? to make sense, even though no question mark is used.

```
ASSEMBLER         BINARY

BIT b,r           11 001 011   01   b   r
```

To test bit 2 in the H register, for example, the instruction would be:

```
ASSEMBLER      HEX        BINARY

BIT 2,H        CB 54      11 001 011   01 010 100
```

A BIT instruction will show the state of the bit tested with the zero flag, this will be set if the bit is 0 or reset if it is 1. The carry flag is not changed by BIT instructions but all the other flags, apart from the zero flag, are set in an unpredictable manner.

One of the uses of the bit level instructions is to allow packing of information. This is the technique whereby one byte is used to hold details about more than one thing. An example of this would be personnel records. Consider a company's database holding the following details about personnel:

1) Male/Female
2) Married/Single
3) Children/Childless
4) Driving licence/No driving licence
5) Salaried/Hourly paid
6) Key holder/Not key holder
7) Security cleared/Not security cleared.

Each of these items could be held in a single bit, since there are only two possible answers to each question, yes/no. Yes could be represented by 1 and no by 0, and seven bits of a byte would suffice to hold all the above information. The eighth bit is often reserved to indicate that the byte is in use.

It would be possible to create records such as this by use of the logical operations AND and OR, but it would be very awkward to change a specific bit by this means once the record was set up. The program given in Fig. 10.1 demonstrates this. It is not suggested you must actually enter it, but try to follow what is happening. The program will work correctly when information is being input for the first time but for alterations the bit operations are much easier.

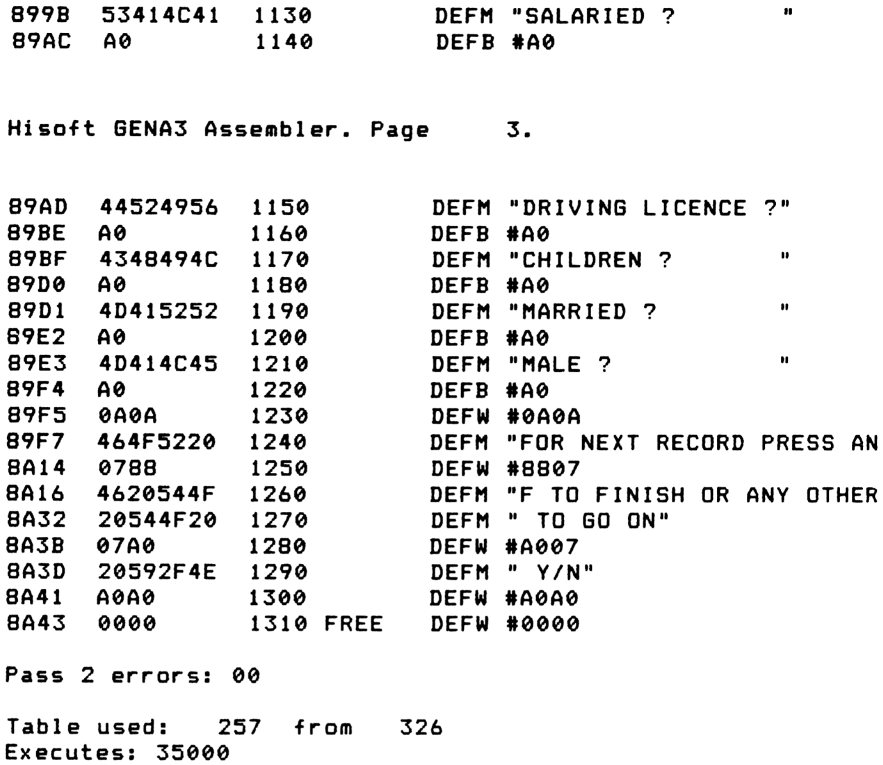
A number of 'dirty tricks' have been employed in the program, as well as many of the techniques and instructions you have learnt so far. See if you can find where the 'Y' at the end of message 8 in line 1140 goes.

The bit in the C register, which is ORed with the A register to show a yes to the question associated with the bit, is shifted left by one bit by the ADD A,A instructions at the label SLA for each question, and the same trick used to set the carry flag for each item which was answered 'Y', when records are being output.

```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                    10 ; FIG 10,1
                    20 ; PROGRAM TO DEMONSTRATE PROBLEMS
                    30 ; OF USING OR TO SET BITS.
88B8                40         ORG   35000
```

```
88B8               50        ENT   35000
BB5A               60 PRINT  EQU   47962
BB18               70 GETKEY EQU   47896
88B8  21438A       80        LD    HL,FREE
88BB  CD5689       90 NXTREC CALL  CRLF
88BE  CD5689      100        CALL  CRLF
88C1  0609        110        LD    B,9
88C3  CDFB88      120        CALL  PR_MSG
88C6  CD6389      130        CALL  KEYIN
88C9  FE66        140        CP    "f"
88CB  284E        150        JR    Z,LSTREC
88CD  3E01        160        LD    A,#1
88CF  77          170        LD    (HL),A
88D0  0607        180        LD    B,7
88D2  0E02        190        LD    C,2
88D4  CD5689      200 NXTBIT CALL  CRLF
88D7  CDFB88      210        CALL  PR_MSG
88DA  C5          220        PUSH  BC
88DB  060A        230        LD    B,10
88DD  CDFB88      240        CALL  PR_MSG
88E0  C1          250        POP   BC
88E1  CD6389      260        CALL  KEYIN
88E4  FE79        270        CP    "y"
88E6  2005        280        JR    NZ,NO
88E8  7E          290        LD    A,(HL)
88E9  B1          300        OR    C
88EA  77          310        LD    (HL),A
88EB  1806        320        JR    SLA
88ED  FE6E        330 NO     CP    "n"
88EF  2802        340        JR    Z,SLA
88F1  18E1        350        JR    NXTBIT
88F3  79          360 SLA    LD    A,C
88F4  87          370        ADD   A,A
88F5  4F          380        LD    C,A
88F6  10DC        390        DJNZ  NXTBIT
88F8  23          400        INC   HL
88F9  18C0        410        JR    NXTREC
88FB  CD6C89      420 PR_MSG CALL  SAVREG
88FE  217689      430        LD    HL,MSGTBL
8901  CB7E        440 FNDMSG BIT   7,(HL)
8903  23          450        INC   HL
8904  28FB        460        JR    Z,FNDMSG
8906  10F9        470        DJNZ  FNDMSG
8908  CD0F89      480        CALL  NXTCHR
890B  CD7189      490        CALL  RESREG
890E  C9          500        RET
890F  7E          510 NXTCHR LD    A,(HL)
8910  E67F        520        AND   %01111111
8912  CD5ABB      530        CALL  PRINT
8915  CB7E        540        BIT   7,(HL)
8917  C0          550        RET   NZ
8918  23          560        INC   HL
8919  18F4        570        JR    NXTCHR
891B  21438A      580 LSTREC LD    HL,FREE
```

```
891E  CD5689     590         CALL CRLF
8921  CD5689     600         CALL CRLF
8924  0608       610         LD   B,8
8926  CDFB88     620         CALL PR_MSG
8929  0601       630 PR_REC  LD   B,1
892B  E5         640         PUSH HL
892C  CD5689     650         CALL CRLF
892F  CD5689     660         CALL CRLF
8932  CD18BB     670         CALL GETKEY
8935  E1         680         POP  HL
8936  7E         690         LD   A,(HL)
8937  23         700         INC  HL
8938  A7         710         AND  A
8939  C8         720         RET  Z
893A  87         730 P_ITEM  ADD  A,A
893B  CDFB88     740         CALL PR_MSG
893E  F5         750         PUSH AF
893F  3007       760         JR   NC,NOT
8941  3E59       770         LD   A,"Y"
8943  CD5ABB     780         CALL PRINT
8946  1805       790         JR   NXTITM
8948  3E4E       800 NOT     LD   A,"N"
894A  CD5ABB     810         CALL PRINT
894D  04         820 NXTITM  INC  B
894E  CD5689     830         CALL CRLF
8951  F1         840         POP  AF
8952  28D5       850         JR   Z,PR_REC
8954  18E4       860         JR   P_ITEM
8956  F5         870 CRLF    PUSH AF
8957  3E0D       880         LD   A,#0D
8959  CD5ABB     890         CALL PRINT
895C  3E0A       900         LD   A,#0A
895E  CD5ABB     910         CALL PRINT
8961  F1         920         POP  AF
8962  C9         930         RET
8963  CD18BB     940 KEYIN   CALL GETKEY
8966  CD5ABB     950         CALL PRINT
8969  F620       960         OR   #20
896B  C9         970         RET
896C  E3         980 SAVREG  EX   (SP),HL
896D  C5         990         PUSH BC
896E  F5        1000         PUSH AF
896F  E5        1010         PUSH HL
8970  C9        1020         RET
8971  E1        1030 RESREG  POP  HL
8972  F1        1040         POP  AF
8973  C1        1050         POP  BC
8974  E3        1060         EX   (SP),HL
8975  C9        1070         RET
8976  A0        1080 MSGTBL  DEFB #A0
8977  53454355  1090         DEFM "SECURITY CLEARED?"
8988  A0        1100         DEFB #A0
8989  4B455920  1110         DEFM "KEY HOLDER ?      "
899A  A0        1120         DEFB #A0
```



```
899B   53414C41   1130          DEFM "SALARIED ?        "
89AC   A0         1140          DEFB #A0


Hisoft GENA3 Assembler. Page     3.


89AD   44524956   1150          DEFM "DRIVING LICENCE ?"
89BE   A0         1160          DEFB #A0
89BF   4348494C   1170          DEFM "CHILDREN ?        "
89D0   A0         1180          DEFB #A0
89D1   4D415252   1190          DEFM "MARRIED ?         "
89E2   A0         1200          DEFB #A0
89E3   4D414C45   1210          DEFM "MALE ?            "
89F4   A0         1220          DEFB #A0
89F5   0A0A       1230          DEFW #0A0A
89F7   464F5220   1240          DEFM "FOR NEXT RECORD PRESS AN
8A14   0788       1250          DEFW #8807
8A16   4620544F   1260          DEFM "F TO FINISH OR ANY OTHER
8A32   20544F20   1270          DEFM " TO GO ON"
8A3B   07A0       1280          DEFW #A007
8A3D   20592F4E   1290          DEFM " Y/N"
8A41   A0A0       1300          DEFW #A0A0
8A43   0000       1310 FREE     DEFW #0000

Pass 2 errors: 00

Table used:   257  from   326
Executes: 35000
```

Figure 10.1

Chapter Eleven

Rotates and Shifts

In the program given in Fig. 10.1 the C register was ORed with the A register to set a single bit, signifying that the question associated with the chosen bit had been answered in the affirmative. After returning the byte in the A register to memory, the C register was loaded into the A register, the A register added to itself, and the result returned to the C register. All this was just to move the one set bit in the C register left by one position, ready to set the next bit if necessary. The label SLA was chosen because the action of the section of program was a Shift Left Arithmetic.

Whenever a binary number is added to itself, doing in effect a multiply by two, the bit pattern remains the same but is moved one place to the left. The same occurs with any numbering system when a number is multiplied by the base of the system.

For example:

Binary (base 2) 1010110 * 10 = 10101100 (10b is 2 decimal)
Decimal (base 10) 1234567 * 10 is 12345670
Hex (base 16) 789ABCD * 10 = 789ABCD0 (10h is 16 decimal)

It is rather bothersome if every time it is required to shift the contents of a byte, the A register has to be used. Additionally there is the problem of what to do if a right shift is required, this would be advantageous in the program in Fig. 10.1 as it would allow the information to be displayed in the same order as it was entered. At the moment the display routine again uses the ADD A,A instruction, this time at the label P_ITEM, to shift the bits into the carry, signalling a yes or no.

There are in fact ways in which it would have been possible to output in the same order as information was input. Shifting the data byte in the A register immediately after the OR instead of the C register would have worked, but this would have caused other problems, since the shift has to be made even when, after a negative response, no OR occurs.

What is really required is a set of instructions which will allow shifting of registers, not only to allow the problems outlined above to be overcome, but also because this will allow easy division calculations.

Think back to the start of this chapter, where it was shown that multiplying a number by the base of the number system shifted it left by 1 digit; what happens in a divide? You guessed it! It shifts the number right by 1, and the right-most number falls off the end. Well, near enough if you are thinking in computer terms for bits in a byte, which has a finite size, of 8 bits. And what happens whenever a number exceeds the range that can be held in a byte? It sets the carry flag.

This is all leading to the fact that the Z80 CPU does have instructions to shift a byte left or right. They are called Shift Left and Shift Right; original, isn't it? The Shift Left instruction performs exactly the operation achieved by ADD A,A but is not limited to use on the A register alone. The full instruction is called Shift Left Arithmetic, SLA for short. The instruction is made up from two bytes, the first is a prefix, CBh, and the second is the opcode itself.

```
ASSEMBLER      HEX                   BINARY

SLA r          CB 20-27              11 001 011   00 100   r
```

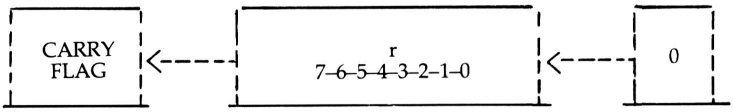


Figure 11.1

r is, as usual, any of the general purpose registers, A or (HL); and you should know the three bit codes by now.

Before going on to consider the right shifts there are a few points to watch out for when using the left shift. As you are already aware, the carry flag is set whenever the result of the shift causes the most significant bit to fall out of the register. This is all very well if the byte being shifted is not a number, and is just being used to indicate something, like in the program in Fig. 10.1. In this case the loss of the MSB does not matter, but if a multiplication was being carried out (sorry about the pun, this time it was not intended) any bit which passes into the carry flag is significant, and must be looked after.

This is normally easy to cope with, the carry must be taken into the next most significant byte. The second part of the addition program in Fig. 6.8 shows one way, by employing the instruction ADC, to collect the carry into the next byte. This is straightforward for any unsigned number. A short subroutine to multiply the value held in the A register by 2 would be:

```
MULT     SLA  A

         LD   (RESULT),A

         LD   A,(RESULT+1)

         ADC  A,A

         LD   (RESULT+1),A

         RET

RESULT   DEFW 0
```

Figure 11.2

The result will be placed in memory at address RESULT and RESULT+1 with the most significant byte in RESULT+1, ready for collection later as a 16 bit number. This subroutine can be used repeatedly to multiply by more than two if required by calling it with the A register holding (RESULT). Each successive call will raise the value to the power of 2. For example:

```
LD   A,1

CALL MULT ;      RESULT is now 2

LD   A,(RESULT)

CALL MULT ;      RESULT is now 4

LD   A,(RESULT)

CALL MULT ;      RESULT is now 8
```

Figure 11.3

And so on until the result exceeds 65535 (which it will after 16 calls in the example above); the carry flag will be set on return to the main program.

This is not a very good program, but it does however serve to illustrate how the Shift Left Arithmetic can be used to multiply. When a negative signed number is being operated on by this technique, the more significant byte of RESULT must be set to 11111111b before starting the calculation, otherwise the final result will be positive.

There is no point in trying to improve the program yet, as instructions which make things very easy are about to be explained. First, though, the right shifts.

The right shift has two forms, the arithmetic shift and the logical shift. The logical shift is exactly the same as the SLA but moves to the right. This may not seem logical, but all will be revealed, so hang on!

The binary Shift Right Logical instruction is the same as the Shift Left Arithmetic, but bits 5, 4 and 3 are changed. All the instructions in this chapter are constructed in this manner, with these three bits dictating the nature of the operation to be performed. The prefix CBh is again present for this instruction.

```
ASSEMBLER       HEX          BINARY

SRL r           CB 38-40     11 001 011   00 111   r
```

The symbolic representation is shown in Fig. 11.4.

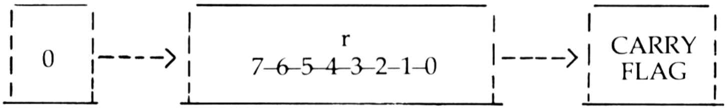


Figure 11.4

At first glance this seems to offer the opportunity to change the MULT subroutine to a subroutine which will divide by two. The SLA will need to be replaced by an SRL instruction, and the order of operations must be reversed, to start on the high byte. The first problem is that there is no way of collecting the carry, out of the bottom of the more significant byte, and using this when the less significant byte is operated on. The SUB instruction cannot be used, because this will always leave the A register holding 0. The divide will therefore have to be restricted to an 8 bit integer. This is shown in Fig. 11.5.

```
DIVD      SRL   A

          LD    (RESULT+1),A

          RET

RESULT    DEFW 00
```

Figure 11.5

Assuming the A register held 100 on entry (64h 01100100b) after execution RESULT+1 will hold 50 (00110010b) which is correct. If an odd number is divided the remainder would be present in the carry flag; remember the carry flag shows that 1 dropped off the end. So if the above subroutine was called with 101 (65h 01100101b) afterwards RESULT+1 would hold 50 and the carry flag will be set showing remainder = 1.

What happens if a negative signed number is divided? Consider the result if DIVD was called with the A register holding −26 (E6h 11100110b). After execution RESULT+1 will hold 01110011b or 73h 115 decimal, which is totally incorrect! The 0 introduced to fill the bit vacated by the shift is to blame, because bit 7 is the sign bit. The fact that this right shift cannot be used for an arithmetic shift is, obviously, the reason for the instruction being called a Shift Right (logical!).

The Shift Right Arithmetic, as you have no doubt guessed, preserves the sign bit. When the subroutine above is rewritten using the SRA instruction in place of the SRL the result after the operation is 11110011b, −13 or F3h, which is correct.

```
SRA r         CB 28-30    11 001 011   00 101   r
```

The symbolic representation is shown in Fig. 11.6.

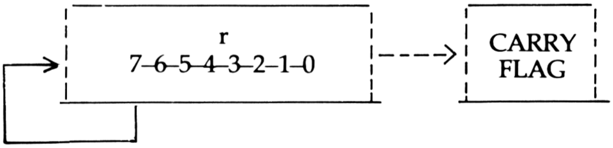
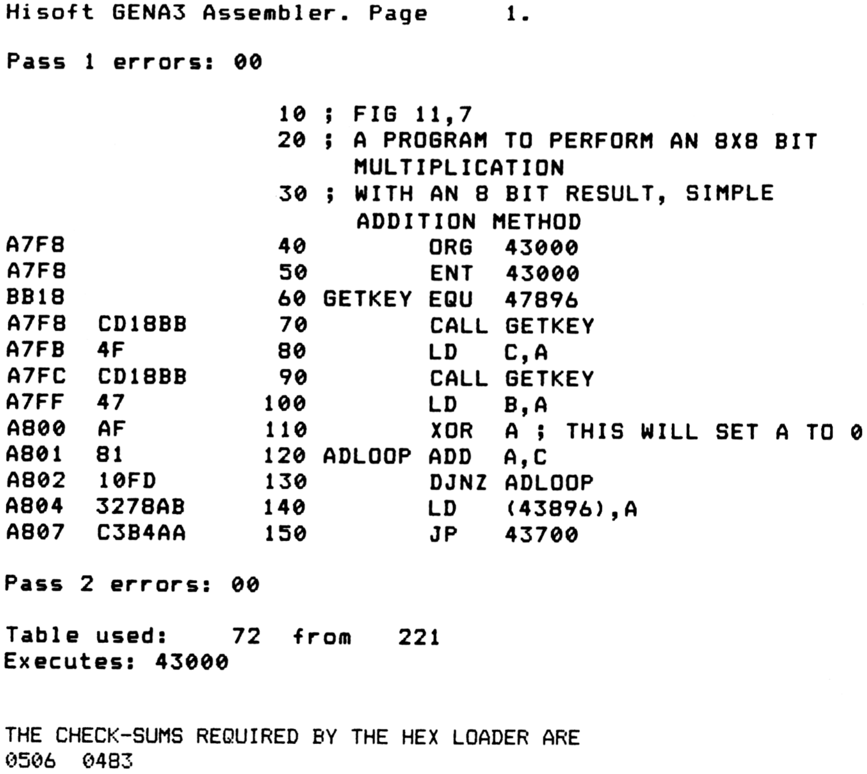


Figure 11.6

Now that you know how to shift right, and shift left, thereby dividing or multiplying by two, the next step is to find out how to multiply and divide numbers by numbers other than two. At this stage assume that not only both parts, but also the result of a calculation, can be accommodated in a single byte. This will start things simply, and give you a chance to grasp the principles of multiplication and division before getting into the really heavy stuff. For unsigned calculations this means that for multiplication the product must be less than 256, and for division that both the divisor and the dividend must be less than 256.

The operation of multiplying is, in effect, simply a process of adding the multiplicand to a result, which is 0 initially, the number of times specified by the multiplier. This can be demonstrated if you load the program given in Fig. 6.14 back into your computer and then add the short program given in Fig. 11.7. This multiplies together the codes of two keys, pressed on the keyboard, by you.



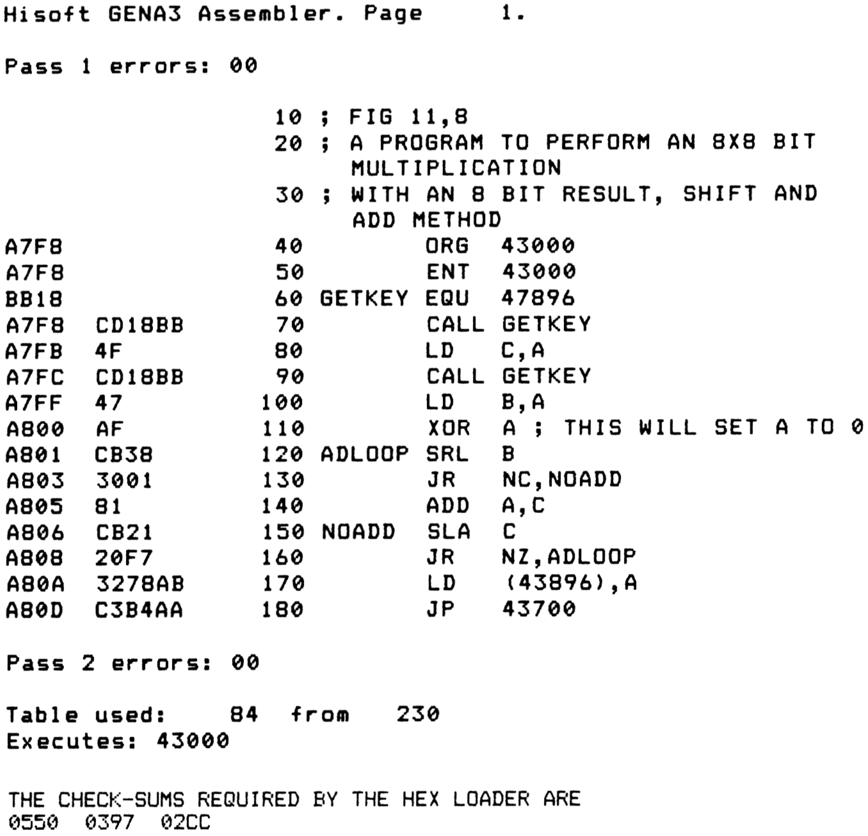
```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        10 ; FIG 11,7
                        20 ; A PROGRAM TO PERFORM AN 8X8 BIT
                             MULTIPLICATION
                        30 ; WITH AN 8 BIT RESULT, SIMPLE
                             ADDITION METHOD
A7F8                    40         ORG  43000
A7F8                    50         ENT  43000
BB18                    60 GETKEY  EQU  47896
A7F8   CD18BB           70         CALL GETKEY
A7FB   4F               80         LD   C,A
A7FC   CD18BB           90         CALL GETKEY
A7FF   47              100         LD   B,A
A800   AF              110         XOR  A ; THIS WILL SET A TO 0
A801   81              120 ADLOOP  ADD  A,C
A802   10FD            130         DJNZ ADLOOP
A804   3278AB          140         LD   (43896),A
A807   C3B4AA          150         JP   43700

Pass 2 errors: 00

Table used:     72  from   221
Executes: 43000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0506  0483
```

Figure 11.7

The vast majority of keys will cause the capacity of a single byte to be exceeded, but many of the "control" codes are useful. These are accessed by pressing the green [CONTROL] key and, whilst still holding it down, another key.

[CONTROL] G will give the BEL code which is 7, and [CONTROL] J will give the line feed code 10 (0Ah), so executing the program by a CALL 43000 or the R command if you are using the assembler, will sit and wait for you to press the keys, and then print out the result of multiplying them together. For example: pressing [CONTROL] G followed by [CONTROL] J will give the answer 70. Appendix III of the Amstrad User Instructions gives a full list of the codes generated by various keys.

Whilst the method employed in Fig. 11.7 works perfectly well for the types of multiplication that it can handle, there is another



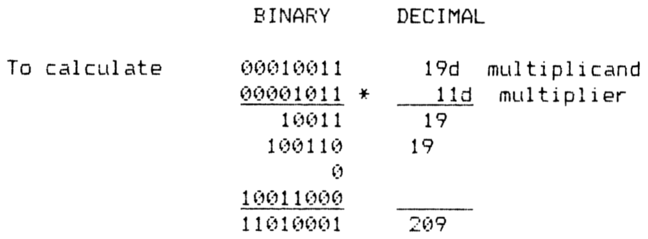
```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                     10 ; FIG 11,8
                     20 ; A PROGRAM TO PERFORM AN 8X8 BIT
                          MULTIPLICATION
                     30 ; WITH AN 8 BIT RESULT, SHIFT AND
                          ADD METHOD
A7F8                 40          ORG  43000
A7F8                 50          ENT  43000
BB18                 60 GETKEY EQU  47896
A7F8    CD18BB       70          CALL GETKEY
A7FB    4F           80          LD   C,A
A7FC    CD18BB       90          CALL GETKEY
A7FF    47          100          LD   B,A
A800    AF          110          XOR  A ; THIS WILL SET A TO 0
A801    CB38        120 ADLOOP SRL  B
A803    3001        130          JR   NC,NOADD
A805    81          140          ADD  A,C
A806    CB21        150 NOADD  SLA  C
A808    20F7        160          JR   NZ,ADLOOP
A80A    3278AB      170          LD   (43896),A
A80D    C3B4AA      180          JP   43700

Pass 2 errors: 00

Table used:    84  from   230
Executes: 43000

THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0550  0397  02CC
```

Figure 11.8

way in which the same task can be performed, but potentially much more efficiently. Needless to say, this uses Shifts and Adds, in place of Adds alone.

The program in Fig. 11.7 would have to circle the addition loop up to 127 times before the result is calculated, and this is for answers below 256. Consider the time that this system will take to arrive at the answer to a multi-byte calculation! Even with a 16 bit (2 byte) limit on the result there can be up to 32767 loops. The smart Alecs out there will be saying: 'Oh no there won't! I would enter it as 2 * 32767 and not 32767 * 2, and the maximum number of loops possible if I always enter the higher number first will be 256, for 16 bit results.'

O.K., fair enough, but what happens when someone else uses the program? Anyway, there is a much better way of doing multiplication, the way one is taught at school for long division. A binary and decimal long division are set out side by side below. Look at what happens during a multiplication in binary, and for that matter in decimal where the multiplier is made up from ones and zeros.



```
                         BINARY      DECIMAL

To calculate            00010011       19d  multiplicand
                        00001011  *  ___11d  multiplier
                           10011       19
                          100110      19
                               0
                        10011000     ______
                        11010001      209
```

At each stage the multiplicand is moved (shifted) one place to the left and if the multiplier is not 0 then the shifted multiplicand is added to the result. When using the decimal system this will only occur occasionally, but with binary it will always be the case, as there can never be anything but 0s and 1s. By using this method the additions are reduced to the minimum, because there can never be more sums than there are non-zero columns in the multiplier. For an eight digit number there can never be more than eight additions and, for a sixteen digit number, there can only be sixteen additions at most.

The program in Fig. 11.8 shows a method of multiplication which duplicates the system above. Once the numbers to be multiplied have been fetched and the result zeroed (the A register for the purposes of this program), the least significant bit of the multiplier is checked. The shift right (SRL) at the label ADLOOP puts the bit into carry so that it can be tested. Then, if the bit was set, the multiplicand is added to the result. At the label NOADD, the multiplicand is shifted one place to the left, exactly as happened in the long multiplication laid out above. A check is then made to see if there is any more to be done, and if so the process is repeated with the new multiplicand, otherwise the calculation is complete, and the result is put away for printing by the subroutine.

Division is roughtly analogous to multiplication but has the added complication that the calculation could go on for ever. The example of this that everybody is aware of is the calculation for Pi (π). The most powerful computers have been put to the task of calculating Pi but as yet there is no sign of a true answer. In all probability there never will be one, after all, who needs to know what Pi is to several million decimal places?

Even in normal maths a recurring number is often arrived at by a simple division, and with computers the normal way of dealing with this is to give a quotient and a remainder. (The quotient is the number of times the divisor can be subtracted from the dividend without making the dividend negative.)

The equivalent to the multiplication program in Fig. 11.7 for division is already very familiar to you. The programs you have been using to print out the results to all the maths you have done so far, operate by successively dividing, by subtraction. Each time the dividend becomes negative the divisor is restored (added to the dividend) to become the dividend for the next division. The proper term for the action which causes the dividend to become negative is an 'overdraw'.

It has been shown how bits which 'fall' out of a register, or memory location, as a result of an instruction being executed, drop into the carry flag, and that, for multiplication, a left shift can save an enormous amount of time in a computation. This left shift can be made to operate on as many bits as required by collecting the bit from carry at subsequent stages of a shift.

To permit this more efficient shift method to be used for division (as well as a great many other things) new operations are necessary. This is because, of the whole repertoire of instructions that have been explained so far, there is not a single one which can collect a carry into the most significant bit, *essential* if division is to be executed by the shift process instead of the repetitive subtraction method. In fact the only way that the carry has been able to be collected by a shift instruction is as shown below.

```
                  Most Sig.Byte    CARRY   Least S.Byte

                  7-6-5-4-3-2-1-0          7-6-5-4-3-2-1-0
                  0 0 0 0 0 0 0 0    0     1 0 1 1 0 1 0 0

SLA LSB           0 0 0 0 0 0 0 0    1     0 1 1 0 1 0 0 0

ADC MSB,MSB       0 0 0 0 0 0 0 1    0     0 1 1 0 1 0 0 0
```

Figure 11.9

This is really a bit of a cheat, because the ADC instruction is being used to simulate a left shift with carry. It is the most economical (in terms of memory and registers used) way of doing this for an eight bit number. A 16 bit left shift can be accomplished with a single instruction, using the same technique as above, but with the ADD HL,HL or the ADC HL,HL instructions.

The Z80 offers a considerable selection of operations, specifically designed to make this sort of task independent of the accumulator registers (the HL register pair is really a 16 bit accumulator, when used for maths operations). All these instructions use the carry flag, both to receive the bit shifted out of the byte and also to provide the bit to be adopted into the position vacated by the operation. Some take the bit from carry before dropping the bit shifted out into carry, allowing the sort of shift achieved by the ADC instruction above. Others put the bit shifted out by their current operation into carry before adopting the carry flag into the vacated bit. Either way they actually do a sort of Rotation, either through carry or including carry.

This can be symbolised as shown in Fig. 11.10. Which as you can see are proper full rotations, and indeed the instructions that act in this manner have the straightforward name 'Rotate', shortened to RR for Rotate Right and RL for Rotate Left. The two functions given in Fig. 11.11, whilst giving the carry flag a copy of the bit which went round and back into the other end of the byte, do not actually take any new information in from the carry. These are called Rotate Circular, and like the above they are shortened, to RRC and RLC, the second R and the L being the direction.

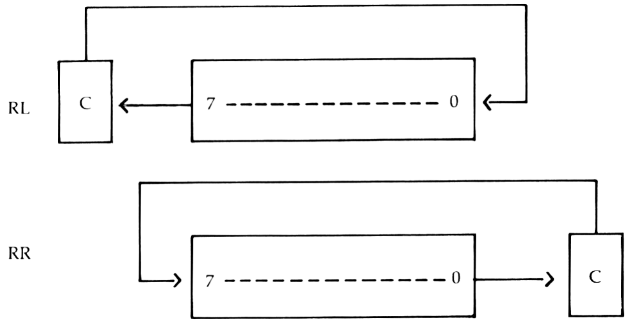


Figure 11.10

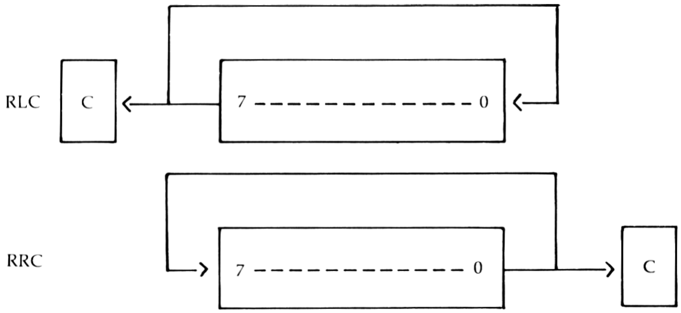


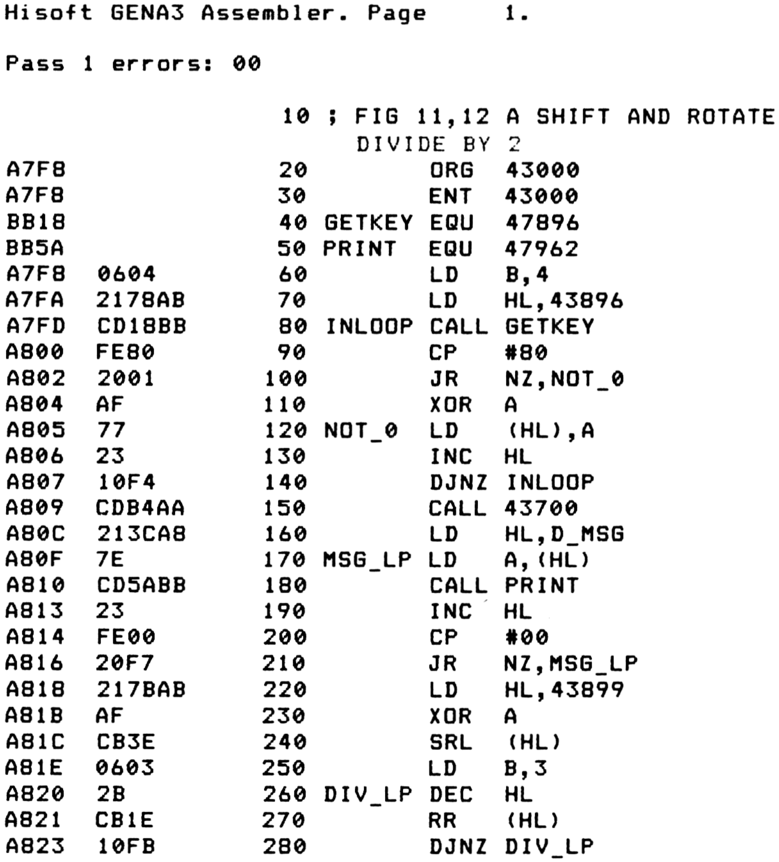
Figure 11.11

The accumulator (the A register) is again favoured with its own special instructions, in addition to the standard opcodes which do an identical job. The full instructions for each of the operations given above are:

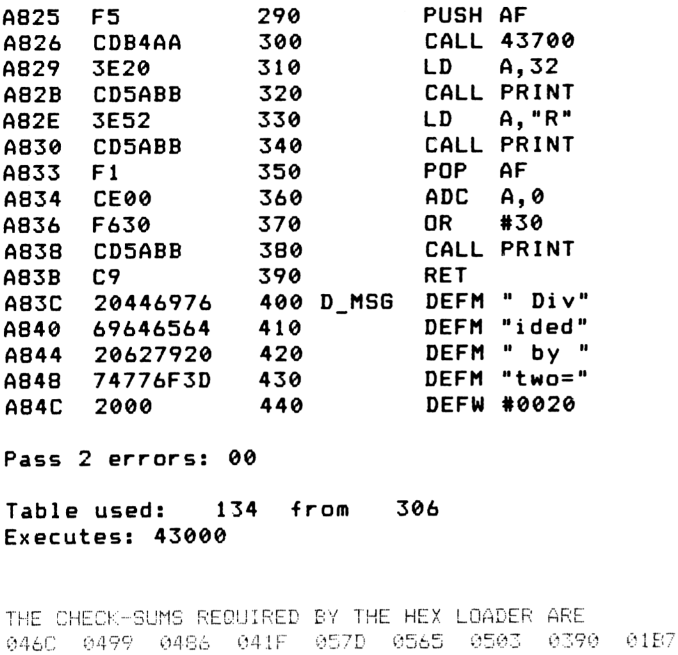
ASSEMBLER	HEX	BINARY	
RL r	CB 10 - 17	11 001 011	00 010 r
RLA	17		00 010 111
RR r	CB 18 - 1F	11 001 011	00 011 r
RRA	1F		00 011 111
RLC r	CB 00 - 07	11 001 011	00 000 r
RLCA	07		00 000 111
RRC r	CB 08 - 0F	11 001 011	00 001 r
RRCA	0F		00 001 111

With the facilities offered by this new set of instructions the gateway to fast division is flung wide open. The division carried out in the number printing routine can never have a quotient that runs to more than nine, so there is no great time lost by using the subtract method, and all the divisors were known in advance. When writing a program where the divisor is not known in advance it is essential to ensure that any attempt to divide by 0 is intercepted. If this precaution is not taken an attempt to divide by 0 will cause the computer to hang up, the result is infinite, so the division will go on for ever.

This test for zero can be carried out in a number of ways. For an 8 bit divisor, the divisor can be put in the A register and tested by an AND A, or for a 16 bit divisor, 1 byte of the divisor is put into the A register and this is ORed with the other byte. Both these methods will set the zero flag if the divisor is 0.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                      10 ; FIG 11,12 A SHIFT AND ROTATE
                           DIVIDE BY 2
A7F8                  20        ORG  43000
A7F8                  30        ENT  43000
BB18                  40 GETKEY EQU  47896
BB5A                  50 PRINT  EQU  47962
A7F8  0604            60        LD   B,4
A7FA  2178AB          70        LD   HL,43896
A7FD  CD18BB          80 INLOOP CALL GETKEY
A800  FE80            90        CP   #80
A802  2001           100        JR   NZ,NOT_0
A804  AF             110        XOR  A
A805  77             120 NOT_0  LD   (HL),A
A806  23             130        INC  HL
A807  10F4           140        DJNZ INLOOP
A809  CDB4AA         150        CALL 43700
A80C  213CA8         160        LD   HL,D_MSG
A80F  7E             170 MSG_LP LD   A,(HL)
A810  CD5ABB         180        CALL PRINT
A813  23             190        INC  HL
A814  FE00           200        CP   #00
A816  20F7           210        JR   NZ,MSG_LP
A818  217BAB         220        LD   HL,43899
A81B  AF             230        XOR  A
A81C  CB3E           240        SRL  (HL)
A81E  0603           250        LD   B,3
A820  2B             260 DIV_LP DEC  HL
A821  CB1E           270        RR   (HL)
A823  10FB           280        DJNZ DIV_LP
```

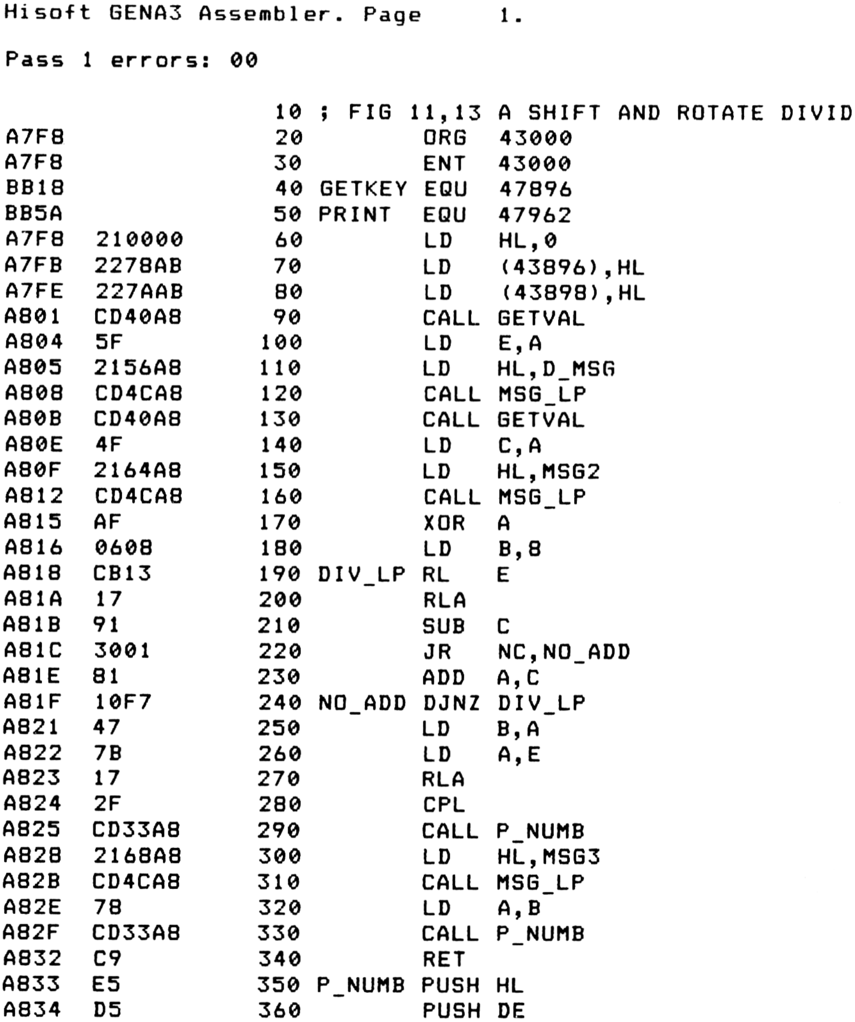


```
A825   F5          290          PUSH AF
A826   CDB4AA      300          CALL 43700
A829   3E20        310          LD   A,32
A82B   CD5ABB      320          CALL PRINT
A82E   3E52        330          LD   A,"R"
A830   CD5ABB      340          CALL PRINT
A833   F1          350          POP  AF
A834   CE00        360          ADC  A,0
A836   F630        370          OR   #30
A838   CD5ABB      380          CALL PRINT
A83B   C9          390          RET
A83C   20446976    400 D_MSG    DEFM " Div"
A840   69646564    410          DEFM "ided"
A844   20627920    420          DEFM " by "
A848   74776F3D    430          DEFM "two="
A84C   2000        440          DEFW #0020

Pass 2 errors: 00

Table used:    134  from   306
Executes: 43000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
046C  0499  0486  041F  057D  0565  0503  0390  01B7
```

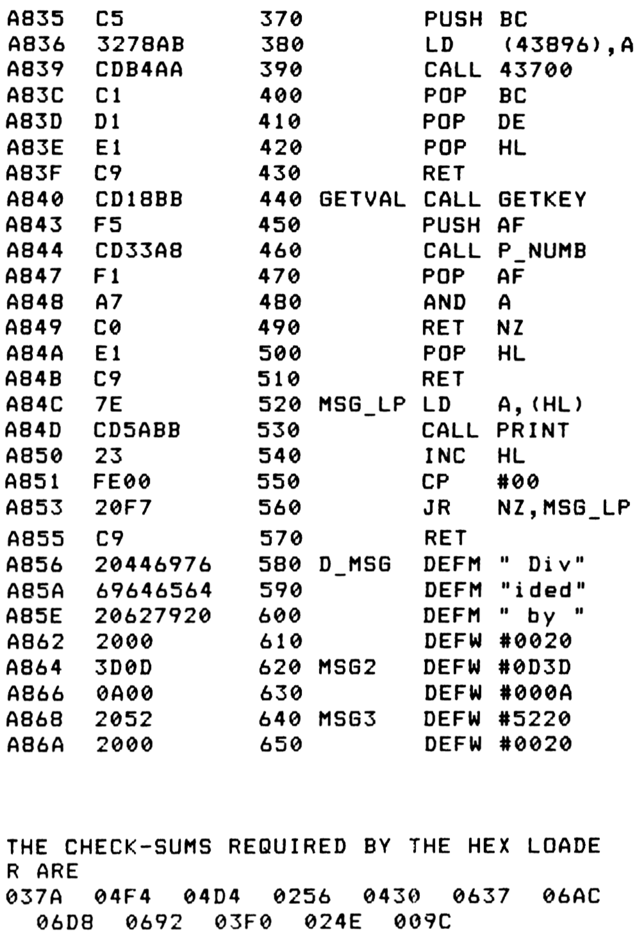
Figure 11.12

Armed with these new instructions a division by two can be performed on any number of bytes by using a SRL or a SRA if the dividend is signed, on the most significant byte, followed by a RR on each of the less significant bytes in order. This is shown in Fig. 11.12. To allow you to enter numbers to be divided an input routine is provided, and this will take the ASCII code of the key you press and use it as one byte of a 32 bit dividend. The code of the first key pressed will become the least significant byte, and each successive key will be the next most significant byte. Since the Amstrad provides no way of generating an ASCII NUL code from the keyboard the 0 key on the numeric keypad is intercepted, and the NUL code 0 is used whenever it is pressed. Yes, Appendix III of the Amstrad User Instructions does say the 0 is generated by [CONTROL] A but they have also got two [CONTROL] Cs. When used with the [CONTROL] key A returns the code 1, B 2 and C 3, the User Instructions are correct thereafter.

Note how the flag register does not need to be saved before using the DJNZ instruction, as this instruction does not corrupt any flags (this is important because the carry flag is holding any remainder at the end of DIV_LP), but that the flags and the A register, which is holding 0 ready for the ADC, *do* need to be saved after the division. Experiment with this program until you are sure that you understand how the Shift and the Rotates achieve the division. Note that the programs in Figs. 11.12 and 11.13 both require the 32 bit number printing routine given in Fig. 6.13 (annotated) to print the number.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                     10 ; FIG 11,13 A SHIFT AND ROTATE DIVID
A7F8                 20           ORG  43000
A7F8                 30           ENT  43000
BB18                 40 GETKEY EQU  47896
BB5A                 50 PRINT  EQU  47962
A7F8  210000         60           LD   HL,0
A7FB  2278AB         70           LD   (43896),HL
A7FE  227AAB         80           LD   (43898),HL
A801  CD40A8         90           CALL GETVAL
A804  5F            100           LD   E,A
A805  2156A8        110           LD   HL,D_MSG
A808  CD4CA8        120           CALL MSG_LP
A80B  CD40A8        130           CALL GETVAL
A80E  4F            140           LD   C,A
A80F  2164A8        150           LD   HL,MSG2
A812  CD4CA8        160           CALL MSG_LP
A815  AF            170           XOR  A
A816  0608          180           LD   B,8
A818  CB13          190 DIV_LP RL   E
A81A  17            200           RLA
A81B  91            210           SUB  C
A81C  3001          220           JR   NC,NO_ADD
A81E  81            230           ADD  A,C
A81F  10F7          240 NO_ADD DJNZ DIV_LP
A821  47            250           LD   B,A
A822  7B            260           LD   A,E
A823  17            270           RLA
A824  2F            280           CPL
A825  CD33A8        290           CALL P_NUMB
A828  2168A8        300           LD   HL,MSG3
A82B  CD4CA8        310           CALL MSG_LP
A82E  78            320           LD   A,B
A82F  CD33A8        330           CALL P_NUMB
A832  C9            340           RET
A833  E5            350 P_NUMB PUSH HL
A834  D5            360           PUSH DE
```

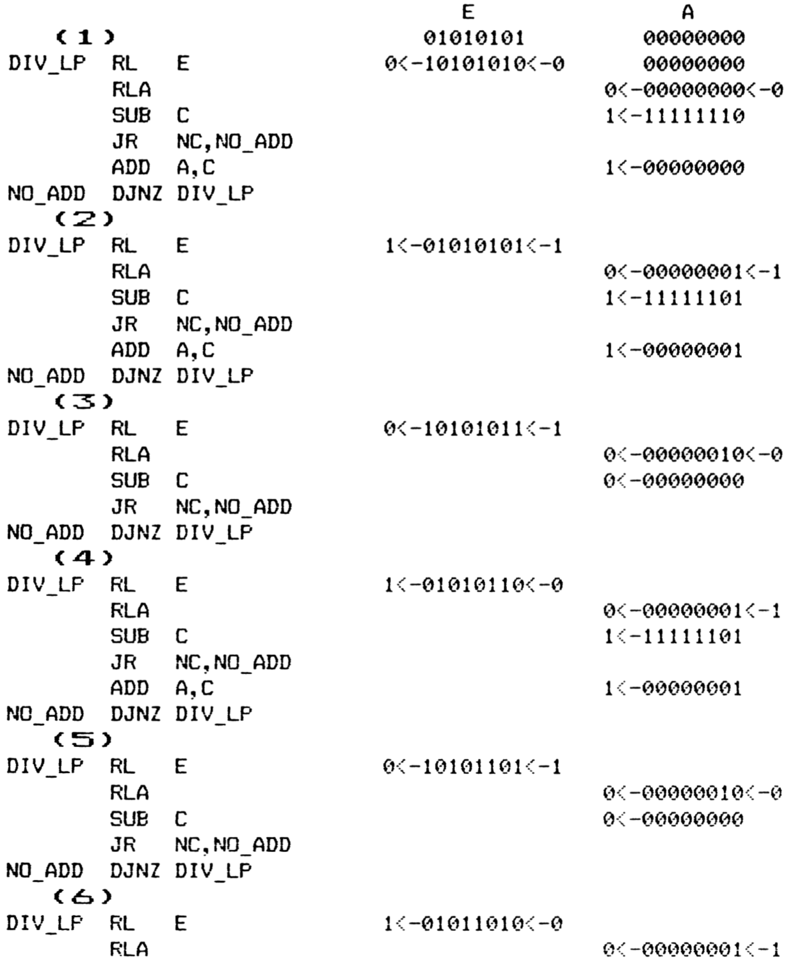


```
A835  C5          370         PUSH BC
A836  3278AB      380         LD   (43896),A
A839  CDB4AA      390         CALL 43700
A83C  C1          400         POP  BC
A83D  D1          410         POP  DE
A83E  E1          420         POP  HL
A83F  C9          430         RET
A840  CD18BB      440 GETVAL  CALL GETKEY
A843  F5          450         PUSH AF
A844  CD33A8      460         CALL P_NUMB
A847  F1          470         POP  AF
A848  A7          480         AND  A
A849  C0          490         RET  NZ
A84A  E1          500         POP  HL
A84B  C9          510         RET
A84C  7E          520 MSG_LP  LD   A,(HL)
A84D  CD5ABB      530         CALL PRINT
A850  23          540         INC  HL
A851  FE00        550         CP   #00
A853  20F7        560         JR   NZ,MSG_LP
A855  C9          570         RET
A856  20446976    580 D_MSG   DEFM " Div"
A85A  69646564    590         DEFM "ided"
A85E  20627920    600         DEFM " by "
A862  2000        610         DEFW #0020
A864  3D0D        620 MSG2    DEFW #0D3D
A866  0A00        630         DEFW #000A
A868  2052        640 MSG3    DEFW #5220
A86A  2000        650         DEFW #0020


THE CHECK-SUMS REQUIRED BY THE HEX LOADE
R ARE
037A  04F4  04D4  0256  0430  0637  06AC
  06D8  0692  03F0  024E  009C
```

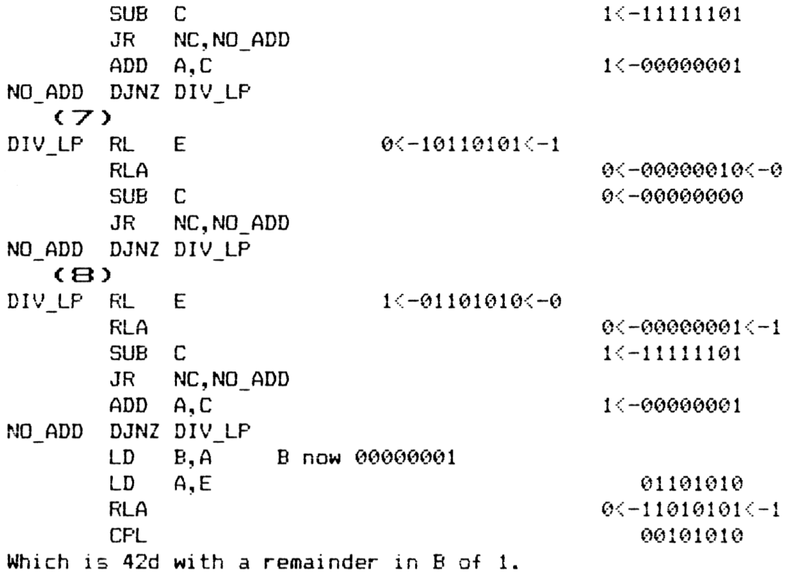
Figure 11.13

Figure 11.13 gives the division equivalent of the program in Fig. 11.9. The similarities are immediately obvious but this time the loop has to be circled for every bit of the calculation. To start with, the dividend is in the E register and the divisor is in the C register, B is used as a counter, with the DJNZ instruction to count the bits of the division. On each pass round the loop the carry flag is rotated into E (the dividend), and the most significant bit of E is rotated into carry. Initially the carry flag was reset, by the XOR A instruction used to clear the A register, so 0 was rotated into the LSB of E. The carry from E is then collected into bit 0 (the least significant bit) of A register by the RLA instruction.

Next an attempt is made to subtract the divisor from the A register; if this causes a carry then the subtraction was not possible, and the A register is immediately restored by adding the divisor back. As with the multiplication routine this is exactly the same process that you perform when doing a long division, as is shown in Fig. 11.14 for dividing 85 by 2. Any carry generated by or taken in by an operation is shown with an arrow indicating the direction.



```
                                          E                 A
    (1)                               01010101          00000000
DIV_LP  RL   E                      0<-10101010<-0      00000000
        RLA                                           0<-00000000<-0
        SUB  C                                        1<-11111110
        JR   NC,NO_ADD
        ADD  A,C                                      1<-00000000
NO_ADD  DJNZ DIV_LP
    (2)
DIV_LP  RL   E                      1<-01010101<-1
        RLA                                           0<-00000001<-1
        SUB  C                                        1<-11111101
        JR   NC,NO_ADD
        ADD  A,C                                      1<-00000001
NO_ADD  DJNZ DIV_LP
    (3)
DIV_LP  RL   E                      0<-10101011<-1
        RLA                                           0<-00000010<-0
        SUB  C                                        0<-00000000
        JR   NC,NO_ADD
NO_ADD  DJNZ DIV_LP
    (4)
DIV_LP  RL   E                      1<-01010110<-0
        RLA                                           0<-00000001<-1
        SUB  C                                        1<-11111101
        JR   NC,NO_ADD
        ADD  A,C                                      1<-00000001
NO_ADD  DJNZ DIV_LP
    (5)
DIV_LP  RL   E                      0<-10101101<-1
        RLA                                           0<-00000010<-0
        SUB  C                                        0<-00000000
        JR   NC,NO_ADD
NO_ADD  DJNZ DIV_LP
    (6)
DIV_LP  RL   E                      1<-01011010<-0
        RLA                                           0<-00000001<-1
```



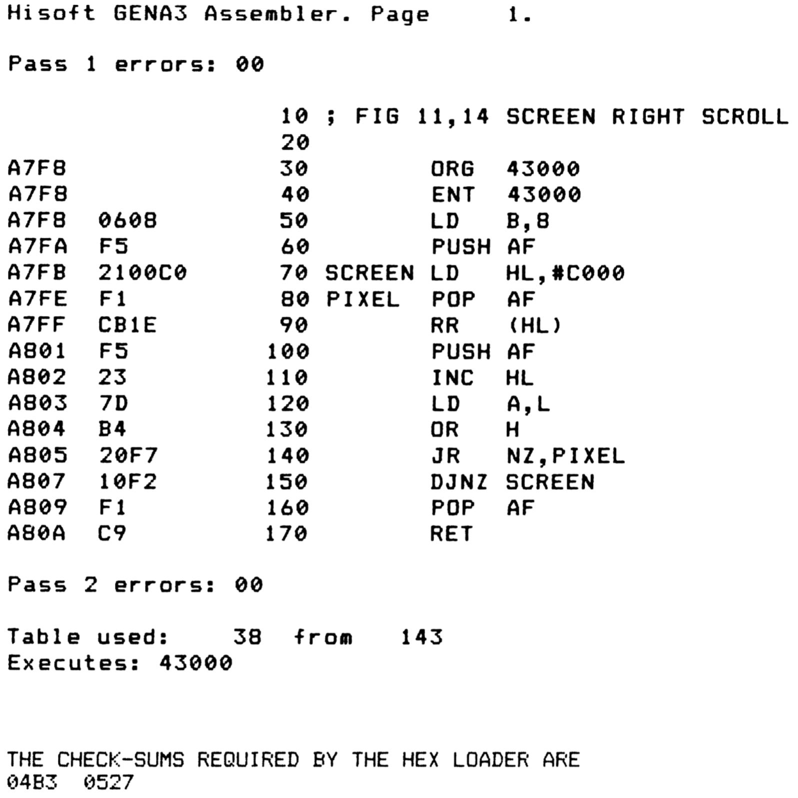
```
        SUB  C                                1<-11111101
        JR   NC,NO_ADD
        ADD  A,C                              1<-00000001
NO_ADD  DJNZ DIV_LP
    (7)
DIV_LP  RL   E               0<-10110101<-1
        RLA                                   0<-00000010<-0
        SUB  C                                0<-00000000
        JR   NC,NO_ADD
NO_ADD  DJNZ DIV_LP
    (8)
DIV_LP  RL   E               1<-01101010<-0
        RLA                                   0<-00000001<-1
        SUB  C                                1<-11111101
        JR   NC,NO_ADD
        ADD  A,C                              1<-00000001
NO_ADD  DJNZ DIV_LP
        LD   B,A     B now 00000001
        LD   A,E                                 01101010
        RLA                                   0<-11010101<-1
        CPL                                      00101010
Which is 42d with a remainder in B of 1.
```

Figure 11.14

It is worth going over the process again and again until you are absolutely certain that you completely understand the means by which the division is perpetrated. This technique of division is known as the restoring method, because of the restoration of the subtraction in the event of a carry. There are other methods for dividing but they are beyond the scope of this book. The restoration method is both efficient and easily adapted to operate on multiple bytes so will enable the programmer to carry out any division required.

There is an interesting alternative use for the shift and rotate instructions. Enter the short program in Fig. 11.15, and once you have entered and saved it, set the Amstrad to mode 2 and put a fair amount of gobbledegook onto the screen. If you are in BASIC a few syntax errors will do the job, or the listing of the Hex loader. Then execute it. If you are using the assembler the [W] command will change the mode.

You will find that the whole screen is scrolled right, one pixel at a time, by 1 character. Try changing the RR (HL) for other opcodes explained in this chapter, and see whether you can predict the result correctly. Note that the flag register is carefully preserved by the program. What will happen if all the PUSHes and POPs are removed? Try it and see. Try also in other modes; can you see why you get the curious effect in other modes? Fig. 11.16 gives the same program for scrolling to the left by 1 pixel; can you see why all the changes had to be made?

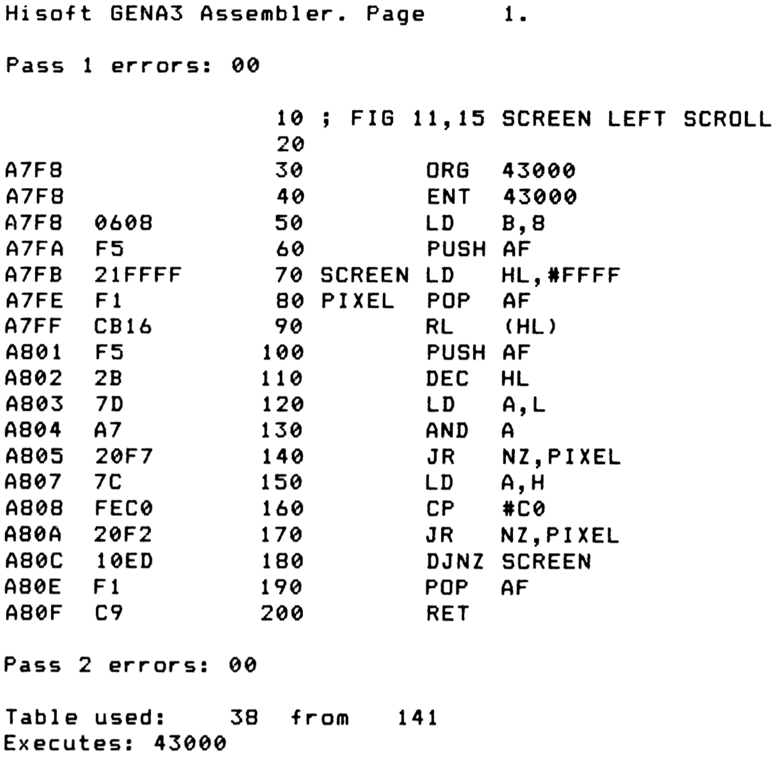


```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                       10 ; FIG 11,14 SCREEN RIGHT SCROLL
                       20
A7F8                   30        ORG  43000
A7F8                   40        ENT  43000
A7F8  0608             50        LD   B,8
A7FA  F5               60        PUSH AF
A7FB  2100C0           70 SCREEN LD   HL,#C000
A7FE  F1               80 PIXEL  POP  AF
A7FF  CB1E             90        RR   (HL)
A801  F5              100        PUSH AF
A802  23              110        INC  HL
A803  7D              120        LD   A,L
A804  B4              130        OR   H
A805  20F7            140        JR   NZ,PIXEL
A807  10F2            150        DJNZ SCREEN
A809  F1              160        POP  AF
A80A  C9              170        RET

Pass 2 errors: 00

Table used:    38  from   143
Executes: 43000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
04B3  0527
```

Figure 11.15

A certain amount of judicious calculation with the aid of the screen map in the appendix will allow you to move pieces of the screen around at will. These routines are slow but when you consider that 16,384 rotates have to be carried out for each pixel to the left or right, and close to 132,000 instructions are executed for a single screen shift by 1 pixel, you may begin to appreciate the speed.

There are two further Rotate instructions, which are shown in the appendix of opcodes. These are the decimal rotates. They are



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                      10 ; FIG 11,15 SCREEN LEFT SCROLL
                      20
A7F8                  30          ORG  43000
A7F8                  40          ENT  43000
A7F8  0608            50          LD   B,8
A7FA  F5              60          PUSH AF
A7FB  21FFFF          70 SCREEN   LD   HL,#FFFF
A7FE  F1              80 PIXEL    POP  AF
A7FF  CB16            90          RL   (HL)
A801  F5             100          PUSH AF
A802  2B             110          DEC  HL
A803  7D             120          LD   A,L
A804  A7             130          AND  A
A805  20F7           140          JR   NZ,PIXEL
A807  7C             150          LD   A,H
A808  FEC0           160          CP   #C0
A80A  20F2           170          JR   NZ,PIXEL
A80C  10ED           180          DJNZ SCREEN
A80E  F1             190          POP  AF
A80F  C9             200          RET

Pass 2 errors: 00

Table used:    38  from   141
Executes: 43000
```

THE CHECKSUMS REQUIRED BY THE HEX LOADER ARE
05E9, 05B2, 02B7

Figure 11.16

outside the scope of this book, and it is most unlikely that you will ever have occasion to use them, except perhaps for operating on the screen. For this purpose the symbolic representation in the appendix is sufficient. They are intended for use in circumstances when binary coded decimal numbers are required, most often for ancillary equipment such as the displays in digital clocks. Binary coded decimal is a system whereby four bits are used to hold a value between 0 and 9 inclusive, only numbers between 0 and 99 can therefore be held in a byte, as opposed to 0 to 255 using the normal Hex numerology. If you are really interested in learning about this sort of instruction you will need to fully comprehend all the concepts explained in this book, and then go on to read a book such as Zaks' *Programming the Z80* ISBN 0 89588 069 5.

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
m = any of r and (HL)
rr = a register pair being used as a 16 bit register
n = an 8 bit number
nn = a 16 bit number
() round a number or register pair = contained in
PC = Program Counter
SP = Stack Pointer

All shifts and all rotates can be used on any of m.

The rotates have a special 1 byte opcode for use on the A register; these special instructions *only* affect the carry flag.

All other rotates and shifts affect all the usable flags according to the contents of m after the operation.

The P/V flag is used to show parity.

The decimal rotates do not affect the carry flag.

For signed division the sign bit can be preserved by use of the SRA instruction.

A circular rotate does not collect the bit put into carry before execution of the instruction.

Right movements divide by 2.

Left movements multiply by 2.

Chapter Twelve

Automated Moves and Searches

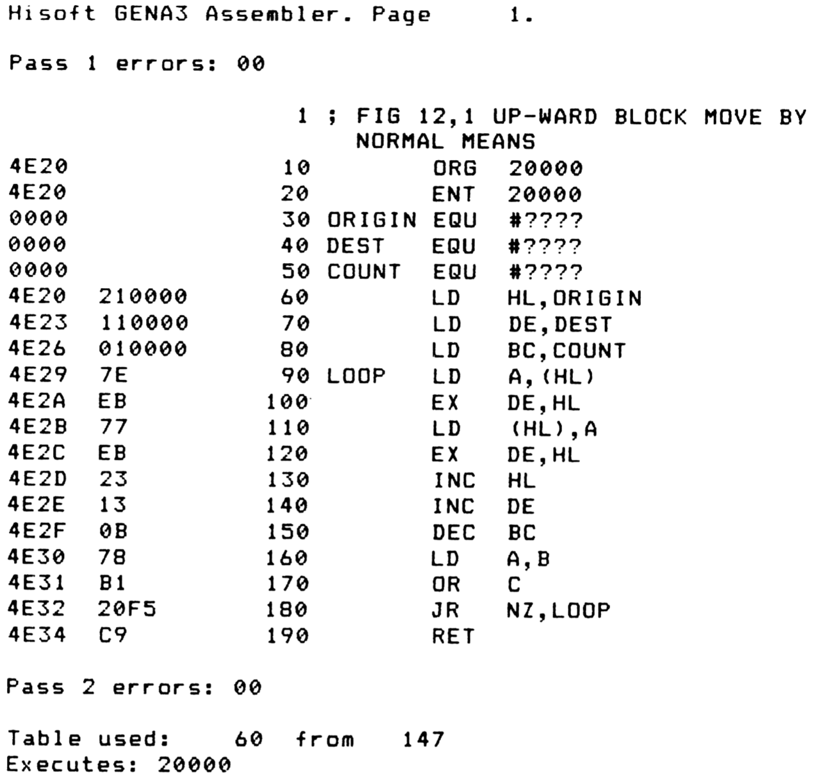
You will probably gather from the chapter heading that the Z80 CPU is well endowed with automated instructions, and you have already learnt about one of them, the DJNZ instruction.

The remaining automated instructions fall into two distinct areas. The block transfer and search group and the block input and output group. The first of these groups will be explained here, and the action of the second will become clear when you read the next chapter.

Suppose for a moment that you wish to move the contents of an area of memory to another area of memory. This could be to create space in a series of records in a data-base program, to save a screen, or part of it in another area or even to scroll the screen as was done in Figs. 11.15 and 16. Assuming that you know how long the block to be moved is, the program would probably look something like Fig. 12.1.

Note how the EX DE,HL instruction is used to allow the address in DE to be loaded from the A register, by temporarily exchanging to put it into HL, and then swapping it back. This could have been achieved equally well (if not better) by using an LD (DE),A instruction, but it demonstrates the use of the exchanges. If for example the A register was not easily available, and less than 257 bytes were to be moved, the program could easily be changed to use the C register and the DJNZ instruction for the loop.

In this program BC is used as a Binary Counter and DE as the DEstination address for the byte being moved. HL acts in its normal role here, that is to point to the address of a byte to be fetched into the A register. Whoever wrote the mnemonic instruction set for the Z80 certainly made it easy to remember

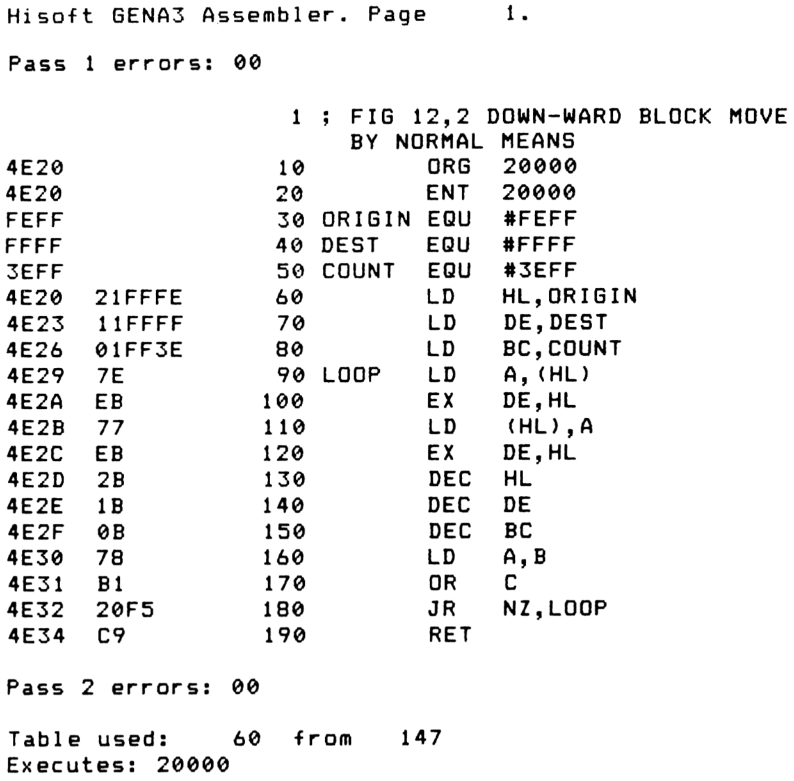


```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        1 ; FIG 12,1 UP-WARD BLOCK MOVE BY
                            NORMAL MEANS
4E20                   10          ORG   20000
4E20                   20          ENT   20000
0000                   30 ORIGIN EQU   #????
0000                   40 DEST     EQU   #????
0000                   50 COUNT   EQU   #????
4E20   210000          60          LD    HL,ORIGIN
4E23   110000          70          LD    DE,DEST
4E26   010000          80          LD    BC,COUNT
4E29   7E              90 LOOP     LD    A,(HL)
4E2A   EB             100          EX    DE,HL
4E2B   77             110          LD    (HL),A
4E2C   EB             120          EX    DE,HL
4E2D   23             130          INC   HL
4E2E   13             140          INC   DE
4E2F   0B             150          DEC   BC
4E30   78             160          LD    A,B
4E31   B1             170          OR    C
4E32   20F5           180          JR    NZ,LOOP
4E34   C9             190          RET

Pass 2 errors: 00

Table used:    60  from   147
Executes: 20000
```

Figure 12.1

roles normally played by register pairs; even they are mnemonics!

Figure 12.1 works fine where the DESTination address in DE is lower than the ORIGIN in HL, but if the destination is above the origin, by less than the count of bytes to be moved, you will get peculiar results. Enter the program and make ORIGIN EQUal to C000h and DEST EQUal to C100h, and set COUNT to EQUal 3EFF. The check sums for the Hex loader will be 036F 04CC 00C9, and then execute it. You will find that the same pattern is repeated several times on the screen. What the program did was copy from the start of the screen memory, C000h, to an address 100h later, and still on the screen. All was fine for the first FFh locations, but thereafter what was copied was not the original contents, but the contents of the start of the screen area, which had been moved there by the first 100h loops.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         1 ; FIG 12,2 DOWN-WARD BLOCK MOVE
                             BY NORMAL MEANS
4E20                    10          ORG   20000
4E20                    20          ENT   20000
FEFF                    30 ORIGIN   EQU   #FEFF
FFFF                    40 DEST     EQU   #FFFF
3EFF                    50 COUNT    EQU   #3EFF
4E20  21FFFE            60          LD    HL,ORIGIN
4E23  11FFFF            70          LD    DE,DEST
4E26  01FF3E            80          LD    BC,COUNT
4E29  7E                90 LOOP     LD    A,(HL)
4E2A  EB               100          EX    DE,HL
4E2B  77               110          LD    (HL),A
4E2C  EB               120          EX    DE,HL
4E2D  2B               130          DEC   HL
4E2E  1B               140          DEC   DE
4E2F  0B               150          DEC   BC
4E30  78               160          LD    A,B
4E31  B1               170          OR    C
4E32  20F5             180          JR    NZ,LOOP
4E34  C9               190          RET

Pass 2 errors: 00

Table used:    60  from   147
Executes: 20000
```

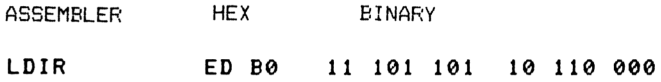
Figure 12.2

This would have been even more fun if you had been making space in a sentence to add something. Consider what will occur with the sentence below if a space is to be made after the 'bro' to insert a 'w'. The sentence starts at address nn, so HL will be loaded with nn + 12, which is where the space is required, and DE will be loaded with nn + 13, because everything after the 'o' of 'bro' is to be moved on by one position. Since five characters are to be moved BC will be loaded with 5.

How now bron cow.	To start
How now bronncow.	after one move,
How now bronnnow.	after two moves,
How now bronnnnw.	after three moves,
How now bronnnnn.	after four moves and
How now bronnnnnn	after the five moves.

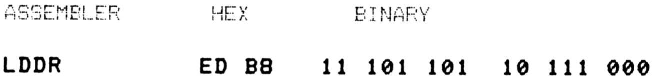
It should now be clear what is going wrong, and the means to overcome the problem should also be apparent if you think about it. Before going on to correct the function of the program the first of the automated block move instructions can be introduced. It will replace lines 90 to 180 inclusive of the assembler listing, which did the LoaDing, the Incrementing and the Repeating.

Being used to the style of this book you know the instruction mnemonic and what it is short for, as well as the operation performed by it, so all that is left to tell are the Hex and Binary forms:



```
ASSEMBLER        HEX         BINARY

LDIR             ED B0    11 101 101   10 110 000
```

There is also a reverse instruction, for moving blocks of memory in the same manner, but starting from the highest address of both the DESTination and the ORIGIN, as opposed to the lowest. Fig. 12.2 gives the long-winded form to move the same block of memory to the same destination using the Decrementing move. The instruction which could be used to replace lines 90 to 180 in this program is:

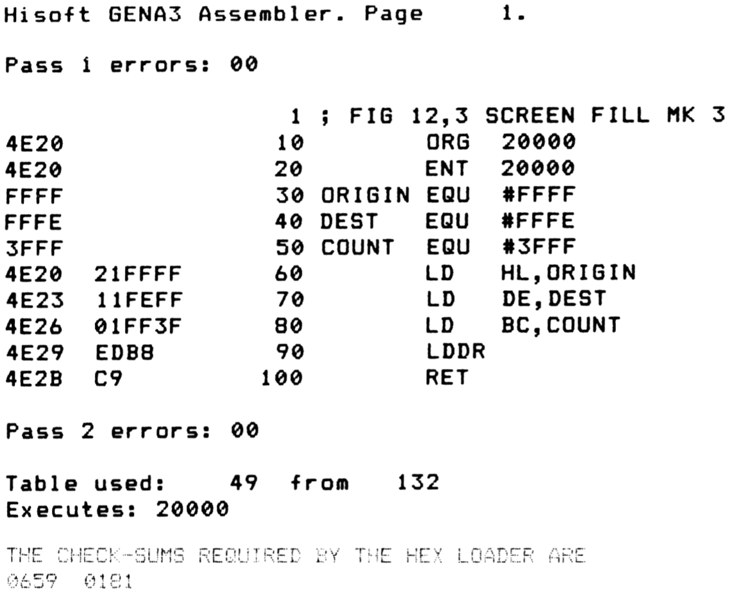


```
ASSEMBLER        HEX         BINARY

LDDR             ED B8    11 101 101   10 111 000
```

Whilst the LDIR and the LDDR instructions perform the same functions as shown in Figs. 12.1 and 2 they do not operate in the same manner. The A register is not used to make the transfer and, instead of employing the zero flag to make the test for the BC register pair reaching zero, the P/V flag is used. This will always be reset on completion of the instruction. No other testable flags are affected in any way by any of the block move instructions.

If the data being moved must not be corrupted, the LDIR instruction will always be safe to use when an area of memory is being moved down, and the LDDR instruction should be employed for any upward move. Obviously the LDDR instruction requires the HL register pair to contain the address at the top of the area to be moved, and the DE register pair to hold the highest address of the memory to be filled. Conversely, when the LDIR instruction is employed, HL should hold the bottom address of the data's origin, and DE the bottom address of the destination.

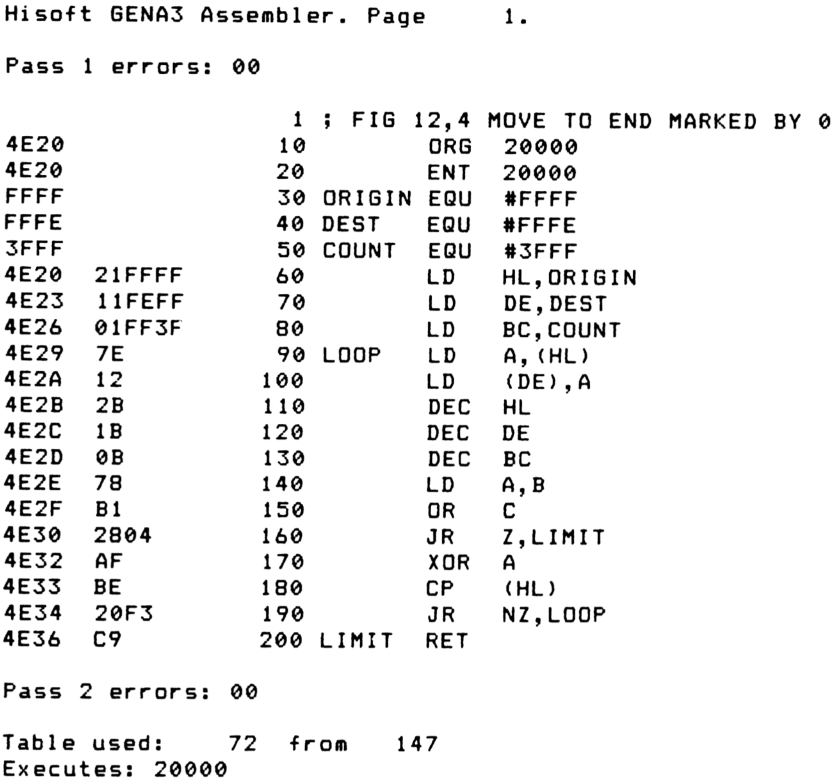
Both instructions can be employed to fill an area with an identical byte, and a SCREEN FILL MK3 version of the program in Fig. 9.2 using the LDDR instruction is given in Fig. 12.3. This deliberately uses the 'overcopying' technique, by moving down by 1 byte. Each byte filled is in turn used as the origin for the next byte to fill. Note that the HL register pair starts at address FFFFh not at 0000h, because the BC is DECremented after the first transfer.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         1 ; FIG 12,3 SCREEN FILL MK 3
4E20                    10         ORG  20000
4E20                    20         ENT  20000
FFFF                    30 ORIGIN  EQU  #FFFF
FFFE                    40 DEST    EQU  #FFFE
3FFF                    50 COUNT   EQU  #3FFF
4E20   21FFFF           60         LD   HL,ORIGIN
4E23   11FEFF           70         LD   DE,DEST
4E26   01FF3F           80         LD   BC,COUNT
4E29   EDB8             90         LDDR
4E2B   C9              100         RET

Pass 2 errors: 00

Table used:    49  from   132
Executes: 20000

THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0659  0181
```

Figure 12.3

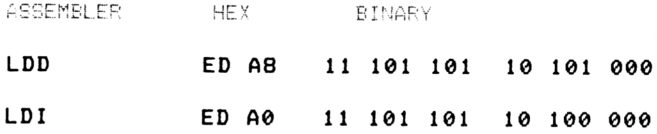
In all the above examples the length of the block to be moved was known, but it would be an awful shame if the full version of the program had to be written whenever it was necessary to include a test for the end being reached. This would end up as shown in Fig. 12.4, supposing that 00 was used to mark the end. BC is still employed to set a limit to the memory that may be moved, otherwise in the event that there was no end marker for some reason, the program would *never* end. Perhaps even worse, the program itself might be written over, and a crash would naturally follow. There are no 'mug traps' at all when you are writing machine code, you have to put them in yourself!



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                          1 ; FIG 12,4 MOVE TO END MARKED BY 0
4E20                     10          ORG  20000
4E20                     20          ENT  20000
FFFF                     30 ORIGIN EQU  #FFFF
FFFE                     40 DEST   EQU  #FFFE
3FFF                     50 COUNT  EQU  #3FFF
4E20  21FFFF             60          LD   HL,ORIGIN
4E23  11FEFF             70          LD   DE,DEST
4E26  01FF3F             80          LD   BC,COUNT
4E29  7E                 90 LOOP   LD   A,(HL)
4E2A  12                100          LD   (DE),A
4E2B  2B                110          DEC  HL
4E2C  1B                120          DEC  DE
4E2D  0B                130          DEC  BC
4E2E  78                140          LD   A,B
4E2F  B1                150          OR   C
4E30  2804              160          JR   Z,LIMIT
4E32  AF                170          XOR  A
4E33  BE                180          CP   (HL)
4E34  20F3              190          JR   NZ,LOOP
4E36  C9                200 LIMIT  RET

Pass 2 errors: 00

Table used:     72  from    147
Executes: 20000
```

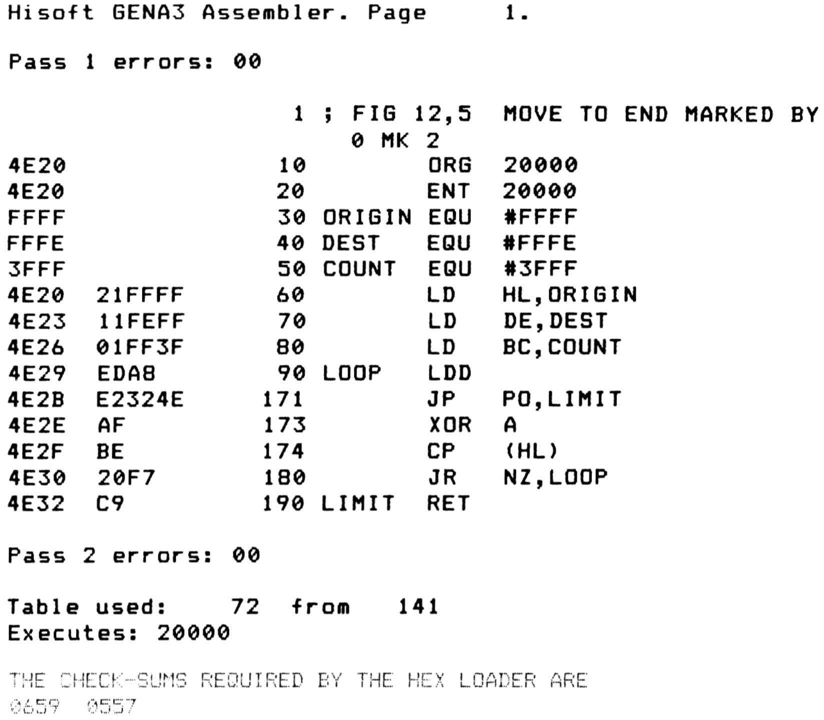
Figure 12.4

Fortunately Zilog have even thought of this, and they have provided a pair of instructions the same as LDIR and LDDR, but without the Repeat, and you'll *never* guess what the mnemonics are, will you?



```
ASSEMBLER       HEX       BINARY

LDD             ED A8    11 101 101   10 101 000

LDI             ED A0    11 101 101   10 100 000
```

It is not quite as simple to change Fig. 12.4 to incorporate the LDD as it might seem at first glance, since the P/V flag is used to indicate the BC register pair reaching zero, instead of the zero flag in the original long program. Using LDD the condition upon which the jump to LIMIT is made must be changed to PO, as the flag is reset when BC = 0. This in turn leads to a further change needing to be made. A relative jump does not have the facility to be made conditional upon the P/V flag, so the jump must therefore be changed to an absolute jump. Fig. 12.5 shows the rewritten program incorporating the LDD instruction.

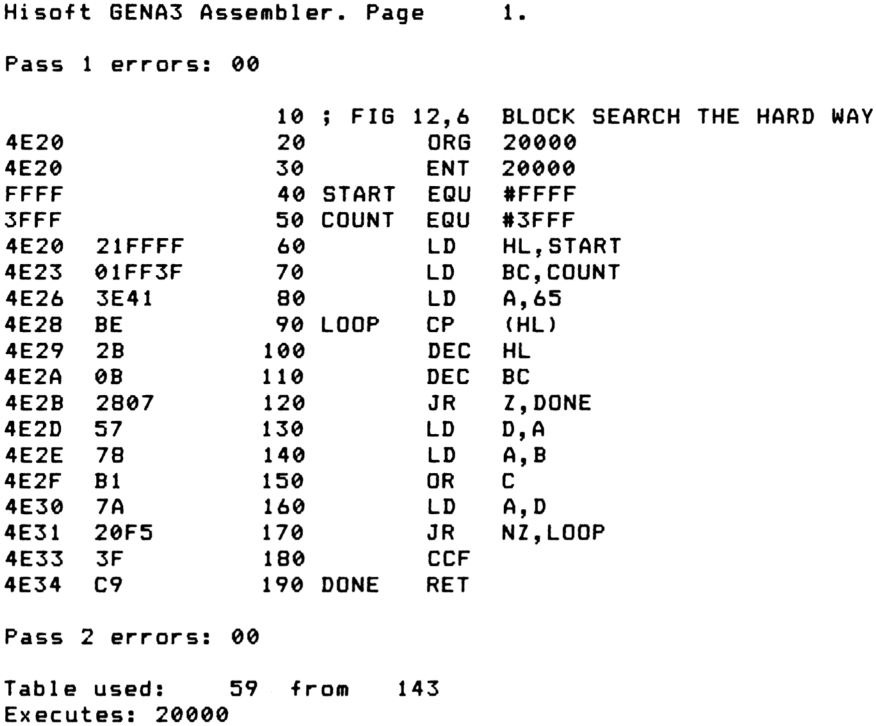


```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         1 ; FIG 12,5  MOVE TO END MARKED BY
                             0 MK 2
4E20                    10         ORG  20000
4E20                    20         ENT  20000
FFFF                    30 ORIGIN EQU  #FFFF
FFFE                    40 DEST   EQU  #FFFE
3FFF                    50 COUNT  EQU  #3FFF
4E20  21FFFF            60         LD   HL,ORIGIN
4E23  11FEFF            70         LD   DE,DEST
4E26  01FF3F            80         LD   BC,COUNT
4E29  EDA8              90 LOOP   LDD
4E2B  E2324E           171         JP   PO,LIMIT
4E2E  AF               173         XOR  A
4E2F  BE               174         CP   (HL)
4E30  20F7             180         JR   NZ,LOOP
4E32  C9               190 LIMIT  RET

Pass 2 errors: 00

Table used:    72  from   141
Executes: 20000

THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0659  0557
```

Figure 12.5

The next automated instructions, and the last to be dealt with in this chapter, are the block search instructions. These follow very closely the form of the block move opcodes but, as their name implies, they are used to search blocks of memory for a byte holding a specific value. To make the same analogies as were made for the block moves Fig. 12.6 gives the long version of a program to find a byte in memory containing 65 (the code for 'A') starting the search at address FFFFh and looking through 3FFFh bytes before giving up if no match is found. When the end of the search loop is reached (the label DONE) the zero flag will be set if a match has been found, and reset if not.

You will see that there have been difficulties saving the A register whilst the test for BC reaching 0 was made. It could not be saved on the stack by a PUSH because the flags would have been



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                      10 ; FIG 12,6  BLOCK SEARCH THE HARD WAY
4E20                  20          ORG  20000
4E20                  30          ENT  20000
FFFF                  40 START    EQU  #FFFF
3FFF                  50 COUNT    EQU  #3FFF
4E20  21FFFF          60          LD   HL,START
4E23  01FF3F          70          LD   BC,COUNT
4E26  3E41            80          LD   A,65
4E28  BE              90 LOOP     CP   (HL)
4E29  2B             100          DEC  HL
4E2A  0B             110          DEC  BC
4E2B  2807           120          JR   Z,DONE
4E2D  57             130          LD   D,A
4E2E  78             140          LD   A,B
4E2F  B1             150          OR   C
4E30  7A             160          LD   A,D
4E31  20F5           170          JR   NZ,LOOP
4E33  3F             180          CCF
4E34  C9             190 DONE     RET

Pass 2 errors: 00

Table used:     59  from   143
Executes: 20000
```

Figure 12.6

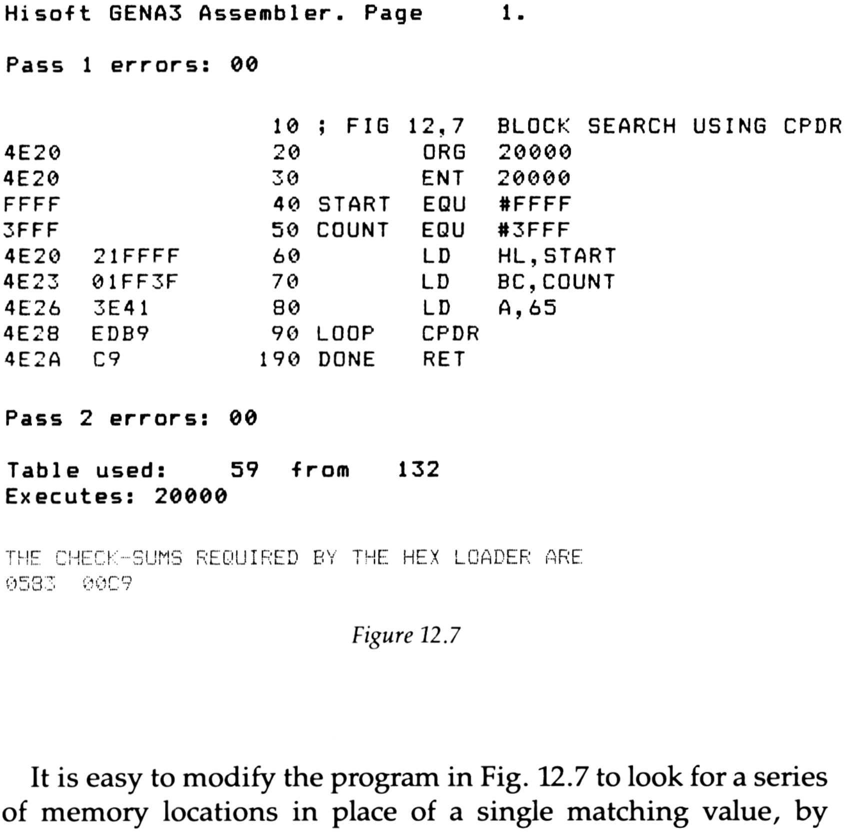
saved as well, and this would have invalidated the test when the POP was made before the jump. The flags would have been restored with the A register before the result of the test could be used. The D register has therefore been put into service as a temporary store for A whilst the tests are made.

The result is a program which successfully does a ComPare Decrement and Repeat through an area of memory, signalling the result with the zero flag when it has finished. The automated instruction which would normally be used for this task, the upward searching version of the same, and the non-repeating versions are:

```
ASSEMBLER       HEX        BINARY

CPIR            ED B1    11 101 101   10 110 001

CPDR            ED B9    11 101 101   10 111 001

CPI             ED A1    11 101 101   10 100 001

CPD             ED A9    11 101 101   10 101 001
```

These instructions use the P/V flag to indicate that the count (in BC) has reached zero, just as the automated LD instructions detailed earlier, but additionally the zero flag is set to indicate a match being found, or reset to show no match.

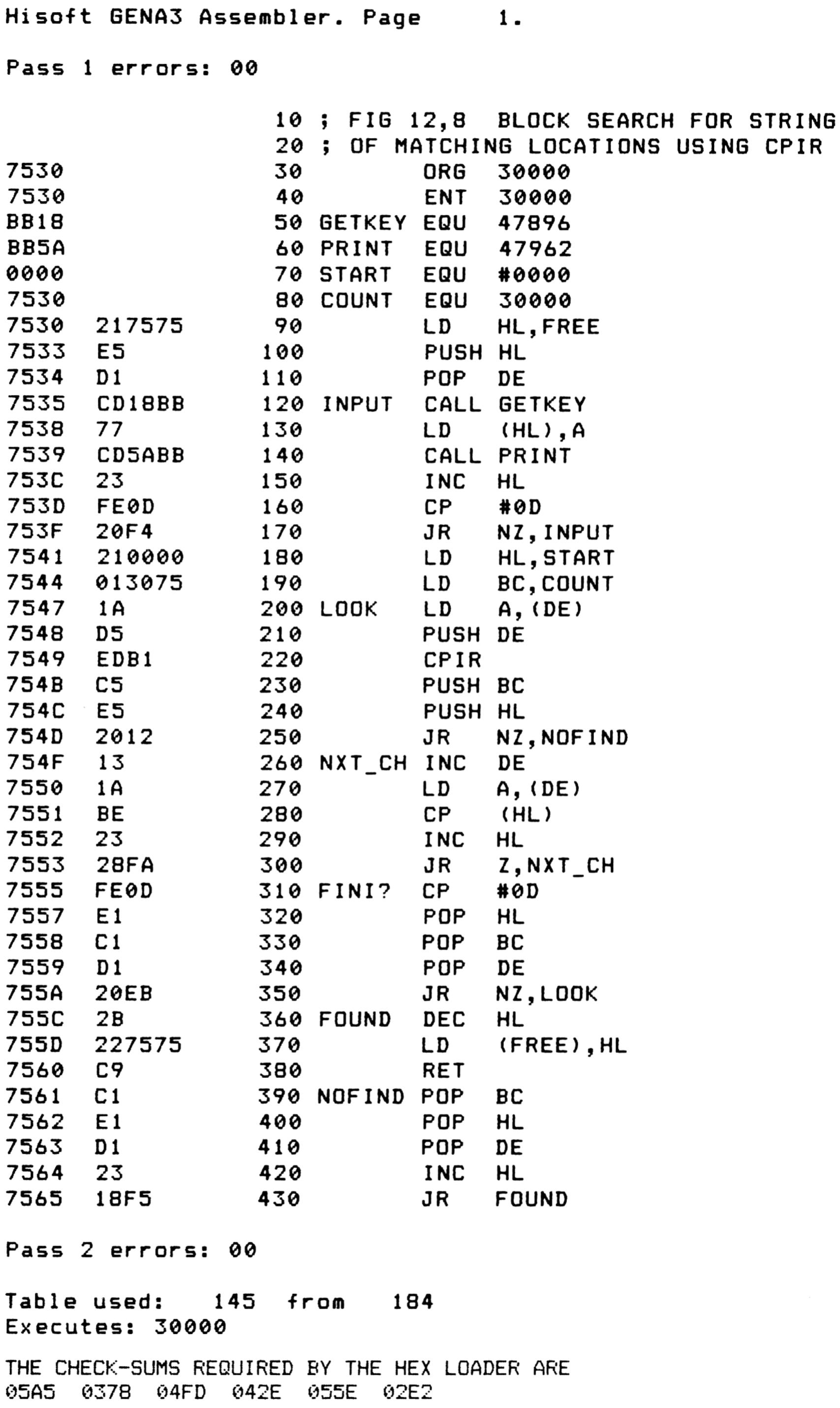
Figure 12.7 shows a rewritten version of Fig. 12.6 using the CPDR instruction. CPIR INCrements the HL register pair after each ComPare instead of its being DECremented, and the two instructions without the R for Repeat, do the same but without the repeat.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                    10 ; FIG 12,7   BLOCK SEARCH USING CPDR
4E20                20         ORG  20000
4E20                30         ENT  20000
FFFF                40 START   EQU  #FFFF
3FFF                50 COUNT   EQU  #3FFF
4E20  21FFFF        60         LD   HL,START
4E23  01FF3F        70         LD   BC,COUNT
4E26  3E41          80         LD   A,65
4E28  EDB9          90 LOOP    CPDR
4E2A  C9           190 DONE    RET

Pass 2 errors: 00

Table used:    59  from   132
Executes: 20000

THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0583  00C9
```

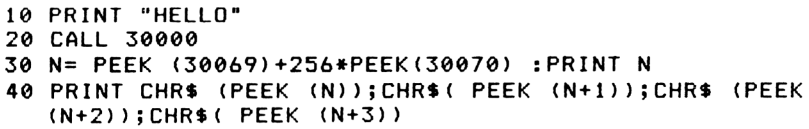
Figure 12.7

It is easy to modify the program in Fig. 12.7 to look for a series of memory locations in place of a single matching value, by adding a small amount of extra code. One application might be to look for an occurrence of a particular word or phrase in stored data, or to look for 'keywords' having been input in an adventure game. Figure 12.8 shows a program to do just this; it allows the input of a string of characters to be searched for, terminated by the [ENTER] key being pressed, and then returns the address of the start of that string, if it occurs in memory.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         10 ; FIG 12,8  BLOCK SEARCH FOR STRING
                         20 ; OF MATCHING LOCATIONS USING CPIR
7530                     30        ORG   30000
7530                     40        ENT   30000
BB18                     50 GETKEY EQU   47896
BB5A                     60 PRINT  EQU   47962
0000                     70 START  EQU   #0000
7530                     80 COUNT  EQU   30000
7530   217575            90        LD    HL,FREE
7533   E5               100        PUSH  HL
7534   D1               110        POP   DE
7535   CD18BB           120 INPUT  CALL  GETKEY
7538   77               130        LD    (HL),A
7539   CD5ABB           140        CALL  PRINT
753C   23               150        INC   HL
753D   FE0D             160        CP    #0D
753F   20F4             170        JR    NZ,INPUT
7541   210000           180        LD    HL,START
7544   013075           190        LD    BC,COUNT
7547   1A               200 LOOK   LD    A,(DE)
7548   D5               210        PUSH  DE
7549   EDB1             220        CPIR
754B   C5               230        PUSH  BC
754C   E5               240        PUSH  HL
754D   2012             250        JR    NZ,NOFIND
754F   13               260 NXT_CH INC   DE
7550   1A               270        LD    A,(DE)
7551   BE               280        CP    (HL)
7552   23               290        INC   HL
7553   28FA             300        JR    Z,NXT_CH
7555   FE0D             310 FINI?  CP    #0D
7557   E1               320        POP   HL
7558   C1               330        POP   BC
7559   D1               340        POP   DE
755A   20EB             350        JR    NZ,LOOK
755C   2B               360 FOUND  DEC   HL
755D   227575           370        LD    (FREE),HL
7560   C9               380        RET
7561   C1               390 NOFIND POP   BC
7562   E1               400        POP   HL
7563   D1               410        POP   DE
7564   23               420        INC   HL
7565   18F5             430        JR    FOUND

Pass 2 errors: 00

Table used:   145  from   184
Executes: 30000

THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
05A5  0378  04FD  042E  055E  02E2
```

Figure 12.8



```
10 PRINT "HELLO"
20 CALL 30000
30 N= PEEK (30069)+256*PEEK(30070) :PRINT N
40 PRINT CHR$ (PEEK (N));CHR$( PEEK (N+1));CHR$ (PEEK
   (N+2));CHR$( PEEK (N+3))
```

This program can be changed to suit almost any search function that you will ever require. Lines 120 to 190 of the assembler listing can be changed to any input routine that is required, and what is done after the search is complete can also be altered to suit. As given, HL will hold either the address of the start of the string searched for, or 0 if it wasn't found. For the purposes of the demonstration program this is also stored in memory, so that it can be accessed by the BASIC program.

The label FINI? is not used except to show that this is where the check to see if a complete match has been achieved is made. Care must always be taken to avoid a routine of this type finding a match in its own sample, otherwise you will never get a 'Not Found' even when there really isn't a copy.

One final *most* important thing about all the automated instructions in this chapter, which you *must* be aware of to make use of them, is the order of events. The HL register pair, and the DE register pair where it is used, are INCremented or DECremented before the BC register pair. They will therefore always point to the next location in the sequence they were following, after they have performed their task.

This can be seen by looking at Fig. 12.8 from the label NXT_CH, where the next character of the string is checked. The DE register pair has to be incremented to point to the next character of the sample, but the HL register pair is already pointing at the next character of the string being tested, on exit from the CPIR loop. This is also the reason for DECrementing HL before finishing the program at the label FOUND. Since it is known that the BC register pair will hold 0 if the allocated amount of memory has been searched and no match has been found, it is this which is popped into the HL register pair at the label NOFIND, saving two bytes even allowing for the INC which allows it to be DECremented back to 0 after the jump to FOUND.

Experiment with this routine and then rewrite it to use the CPDR instruction and see if you get the same results if you search for the "HELLO" in the basic program.

Résumé

There now follows a very brief résumé of the instructions you have learnt in this chapter:

r = a single 8 bit register A, B, C, D, E, H or L
rr = a register pair being used as a 16 bit register
n = an 8 bit number
nn = a 16 bit number
() round a number or register pair = contained in
PC = Program Counter
SP = Stack Pointer

LDIR loads the contents of address HL into address DE, INCrements DE and HL, DECrements BC then, if BC is not 0, repeats.

LDDR loads the contents of address HL into address DE, DECrements DE and HL, DECrements BC then, if BC is not 0, repeats.

LDI and LDD act exactly as the LDIR and LDDR instructions, but without repeating.

CPIR ComPares the contents of the A register with the contents of address HL, INCrements HL, DECrements BC and repeats unless there was a match or BC became 0. If there was a match the zero flag will be set.

CPDR ComPares the contents of the A register with the contents of address HL, DECrements HL, DECrements BC and repeats unless there was a match or BC became 0. If there was a match the zero flag will be set.

CPI and CPD act exactly as the CPIR and CPDR instructions, but without repeating.

For all the above instructions the P/V flag is used to test for BC reaching 0, and the flag is reset when it does, therefore the condition test PO will jump on BC = 0.

Chapter Thirteen

Communicating with the Outside World

All the instructions so far have been concerned with processing information held within the computer. You may have wondered how the computer gets this information to start with, or you may have taken it for granted that when you press a key, your Amstrad knows about it. In fact, if it wasn't specifically told to take information in from outside its own little world of memory, screen and processor, your computer would quite happily sit there, running its programs or whatever, and totally ignore you. You could press keys until you were blue in the face, or died of starvation, and it wouldn't make a blind bit of difference to the computer. The only thing that would worry it, would be having the power switched off. Equally if a program required information to be given *out* from a program to anything other than the screen the computer has to be given the means to do it. The operating system provides the means to access the existing items, such as the keyboard and the sound chip, which are outside the CPU's immediate field of vision, so you are very unlikely to need to access it yourself.

Without going too deep into the technical specification of the CPU and thereby switching to a new subject, here is a *very* basic explanation of how it communicates outside its own boundaries.

You should by now be reasonably familiar with the binary system, and how a series of 1s and 0s can make up a number. You have also learnt that the computer represents a 1 having a switch 'on' and 0 its being 'off'. The Z80 CPU has forty pins, each of which is used for a specific function. The two wires which join the keyboard to the monitor act in a similar way. One line has two wires which give power (electricity) to the computer, and the other wire, which has six cores each of which goes to one pin on the connector, sends the picture information to the monitor.

Sixteen of the pins on the CPU are used to give the address with which the CPU wishes to communicate, and eight are used to send or receive the data. Other pins are used to indicate whether the memory is to be used, or the outside world is to be communicated with, and if the CPU is going to send information or expects to receive it.

The sixteen pins which give the address are known as the *Address Bus*. This is normally shortened to just A, and each pin is referred to by the bit number which it gives. The pin which gives bit 0 (0 or 1 decimal) is called A0, the pin which gives bit 1 (value 2 when set to 1) is called A1, and so on to A15 which gives bit 15 (32768 decimal when set). The eight pins through which data are passed are called the *Data Bus* and these are known as D0 to D7. Figure 13.1 shows this, as well as some of the other pins.

When the CPU executes an instruction such as LD A,(3456) it will signal that it wishes to use memory, and that it wishes to read

```
          SOME OF THE PINS ON THE Z80

                  ----------
                 :          :
      A0  --<---:          :------D0
      A1  --<---:          :
      A2  --<---:          :------D1
      A3  --<---:          :
      A4  --<---:          :------D2
      A5  --<---:          :
      A6  --<---:          :------D3
      A7  --<---:          :
                 :  Z80 CPU :
      A8  --<---:          :------D4
      A9  --<---:          :
     A10  --<---:          :------D5
     A11  --<---:          :
     A12  --<---:          :------D6
     A13  --<---:          :
     A14  --<---:          :------D7
     A15  --<---:          :
                 :          :
     --                     :        ----
     RD   --<---:          :--->--MREQ
                 :          :
     --                     :        ----
     WR   --<---:          :------IORQ
                 :----------:
```

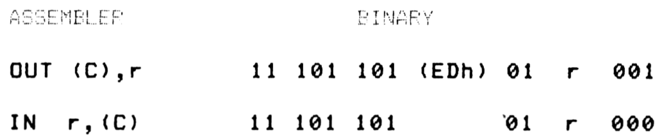
RD signals a ReaD. WR signals a WRite. MREQ signals a Memory REQuest, in other words memory is to be used. IORQ signals an Input or Output ReQuest. The line over these pins signifies that they are active when low (binary 0).

Figure 13.1

information from the address which it has put onto the *Address Bus*. It will then read the contents of that address in memory through the *Data Bus*.

If you want to use something other than memory you will have to specifically tell the CPU that you want to send *out* or receive *in* from somewhere else. This you do with an OUT or an IN instruction. There is quite a number of this type of instruction available in the CPU but, because of the way the Amstrad has been designed, only one IN and one OUT instruction are of any interest.

The address for any outside correspondence with the CPU is specified by the BC register pair, and the instructions are:



```
ASSEMBLER               BINARY

OUT (C),r          11 101 101 (EDh) 01   r   001

IN  r,(C)          11 101 101       01   r   000
```

The B register provides A8 to A15 and the C register provides A0 to A7. Putting 1234h in BC will therefore set A0 to A15 as

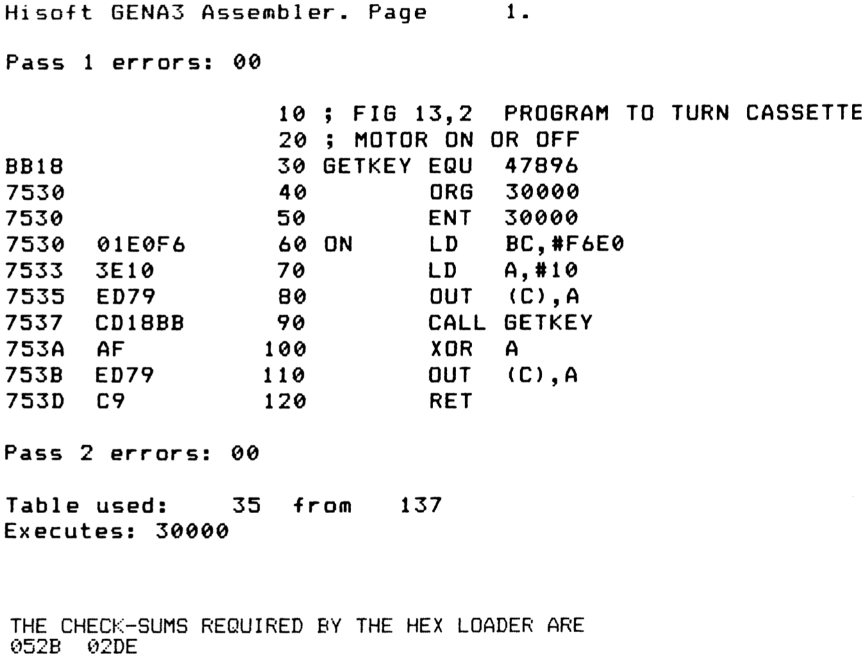
A0) 00010010 00110100 (A15

An address for a piece of external equipment is not normally actually called an address, but is called a 'Port' instead. This avoids confusion over whether an address is internal (that is, in memory) or external. Anything being transferred through a port is known as I/O, which is short for In/Out.

Due to the design of the Amstrad very few values can be used for BC. The most likely use you will have for these instructions will be for use with a peripheral device of some sort. If you are at a level where you are building or controlling this type of equipment, A8 to A15 of the address bus, and therefore the B register, will have to hold either F8h F9h FAh or FBh. For all these lines A10 will be at logic 0 (low). As long as this line is low, and the bottom eight bits of the address bus hold between E0h and FEh inclusive, you will not interfere with any of Amstrad's allocations for existing or future equipment for use with the CPC 464.

Further details of existing equipment already attached to the Amstrad, using the INput and OUTputs, are available in the Amstrad Programmer's Reference Manual.

One quickie program which is of no practical use at all, but which shows the OUT instruction in use, is given in Fig. 13.2. This allows you to switch the cassette motor on and off. The cassette is on the other side of an interface circuit (UPD 8255) which has three I/O channels. Channel A is accessed by port F4xxh, channel B by port F5xxh and port F6xxh is channel C. Control is via port F7xxh. In each the xx can be any value (and will be held in the C register) but unless you use one of the unused addresses for A0 to A7, mentioned above, you will be asking for trouble.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         10 ; FIG 13,2  PROGRAM TO TURN CASSETTE
                         20 ; MOTOR ON OR OFF
BB18                     30 GETKEY EQU  47896
7530                     40        ORG  30000
7530                     50        ENT  30000
7530  01E0F6             60 ON     LD   BC,#F6E0
7533  3E10               70        LD   A,#10
7535  ED79               80        OUT  (C),A
7537  CD18BB             90        CALL GETKEY
753A  AF                100        XOR  A
753B  ED79              110        OUT  (C),A
753D  C9                120        RET

Pass 2 errors: 00

Table used:    35  from   137
Executes: 30000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
052B  02DE
```

Figure 13.2

Chapter Fourteen

Other Instructions

The Z80 CPU has two sets of general purpose registers, as well as a second A register and flag register. With most computers you can use the Z80 instructions to switch between the first and second sets of general purpose registers, and/or the duplicate A and F registers, whenever you wish. If you do not have an intimate knowledge of the operating system, the Amstrad precludes the use of one complete set of these registers.

The instructions which allow you to change the set of registers in use are shown in the appendix, but it is strongly suggested that you forget their existence until you possess, and have mastered, the Programmer's Reference Manual. This will be no easy task, and one magazine recommended that one should 'take several degrees in electronics and computer science' before trying to use it. This was rather overstating the degree of proficiency needed before you can make use of *some* of the information given there. If you have managed to get this far, and assuming you are able to understand the action and operation of the instructions that have been explained, you will find an enormous wealth of usable data therein.

Interrupts

Interrupts are the means by which the Amstrad can execute the BASIC instructions such as EVERY and AFTER; they are generated by the Amstrad at regular intervals. Every time an interrupt is made the Z80 CPU reacts in one of three ways. These are called *Interrupt Modes* shortened to IM to form the mnemonic. The interrupt mode can be set to any of the three modes available by a program but again due to the design of the Amstrad there is only one mode which is worthy of consideration. The instructions to set the interrupt mode are given in the appendix.

The cold start sequence in the Amstrad, which clears the memory and sets the early-morning wake-up state, also initialises the interrupt mode to 1 (IM1). In this mode any interrupt causes a special sort of CALL to address 56 (38h). The program which starts here is known as the *Interrupt Service Routine* and since it can be called into action at *any* point in the main program, an interrupt service routine *must* save any registers it is going to use before modifying them, and restore them before returning to the main program.

The special CALL generated by an interrupt, automatically stops any further interrupts being acknowledged, so before returning from an interrupt service routine to resume execution of the main program the interrupts must be enabled, so that future interrupts are not ignored. The instruction which allows the interrupts to be enabled is called Enable Interrupts, and there is also a corresponding instruction to Disable the Interrupts.

```
ASSEMBLER       HEX       BINARY

DI              F3        11 110 011

EI              FB        11 111 011
```

Locomotive software have fortunately thought of the machine code programmer and given a means by which interrupts can easily be used. With most machines based on the Z80 you have to use IM2 to gain access to the interrupts, and as you will see when you read on, this is not the easiest of things to do. On your Amstrad, once you have written the interrupt service routine, all you have to do is add a JP #B939 at the end, where you would normally have put the RET instruction. To make the interrupt routine active load the HL register pair with the start address of your routine, and then LD (#39),HL. From now on every interrupt will call your routine. To disable your interrupt routine the instructions

```
ASSEMBLER        HEX

LD HL,#B939      21 39 B9

LD (#39),HL      22 39 00
```

should be used.

Do not attempt to change the address at 39h in two steps, because it is possible that an interrupt could occur half way through the change, making the interrupt call the wrong address.

A brief description of IM2 is given below, and this will stand you in good stead for the future. Do not attempt to use this mode, or IM0 without going into *and understanding* the detail given in the Programmer's Reference Manual about using the interrupts. The degrees *will* be helpful when you are trying to cope with this part!

By setting IM2 it is possible to use the interrupts for your own purposes, so long as you end your interrupt servicing routine by re-enabling the interrupts before returning to the CALLing program and end with a RETI instruction.

Remember that you will have to reset the IM1 mode and enable the interrupts before you return to BASIC unless you are using a RST 56 (38H) within the interrupt routine.

The IM2 mode is somewhat convoluted and operates as follows: On receipt of an interrupt the CPU saves the address of the next instruction in the program that it is executing on the machine stack, and disables any further interrupts. It then looks at the location pointed to by the data bus + (256 * the I register) and jumps to the address which is contained in this location + (256 * the contents of thė following location). For example: the I register contains 10 (0AH) and the interrupting device places 200 on the data bus when it makes the interrupt.

10 * 256 = 2560. 2560 + 200 = 2760.

Therefore the address to be jumped to will be taken from the contents of address 2760 + (256 * the contents of address 2816). If 2760 contained 90 and 2761 contained 187 then the address jumped to would be 90 + (256 * 187) which is 47962.

Or if the I register was 187 and the interrupting device gave 90

187 * 256 = 47872. 47872 + 90 = 47962

47962 contains 207 and 47963 contains 0

0 + (207 * 256) = 52992.

So the jump will be to 52992.

This can be represented by imagining the interrupt as an invisible instruction in the program being run. At the moment of the interrupt the invisible instruction is executed as if it were a DI followed by a CALL instruction in the address immediately prior to address pointed to by the I register and the data bus, the address being CALLed is in the next two bytes in the standard Z80 low byte first order. The instruction, being invisible, cannot place its own return address on the machine stack, hence the address after the last instruction executed in your program goes onto the stack, and it is this address that will be returned to after the RETI instruction at the end of the interrupt service routine.

The RETI instruction must be preceded by an EI instruction. The reason for the DI being incorporated in the CALL performed by the interrupt is to ensure that, should the interrupt service routine be longer in execution time than the delay between two interrupts, the program does not become tied up in a loop.

It is quite easy to write a program which changes the address jumped to by an interrupt by loading the vector bytes (the two addresses looked at to determine where the jump is made to) with the desired address within the program.

Whenever interrupt routines are used it is of vital importance that any registers which are used by the interrupt routine are preserved on entry, and restored before going back to the main program, and that no attempt is made to pass data to and from the interrupt routine in registers.

Typical uses of interrupt routines are for SPRITE control and constantly checking for keys being pressed within a program. If it is known how often an interrupt will be generated it is easy to calculate the speed of movement for a SPRITE and, since it will be independent from any other operation within the program, the speed will normally remain constant.

Figure 14.1 gives a short program which, when executed from INIT, will set up the standard interrupt to point to the label START. Every interrupt thereafter the routine between START and FINISH will be performed. This simply sets memory at address 31100 to hold 123. To revert the interrupt to its original address CALL the routine at the label DISARM.

Return to BASIC if you were using the assembler, and before using any of the machine code, try the following as direct commands.

```
? PEEK (31100)
```

should return 0

```
POKE 31100,10: ? PEEK (31100)
```

should return 10 and

POKE 31100,0: ? PEEK (31100)

should return 0 again. Now type

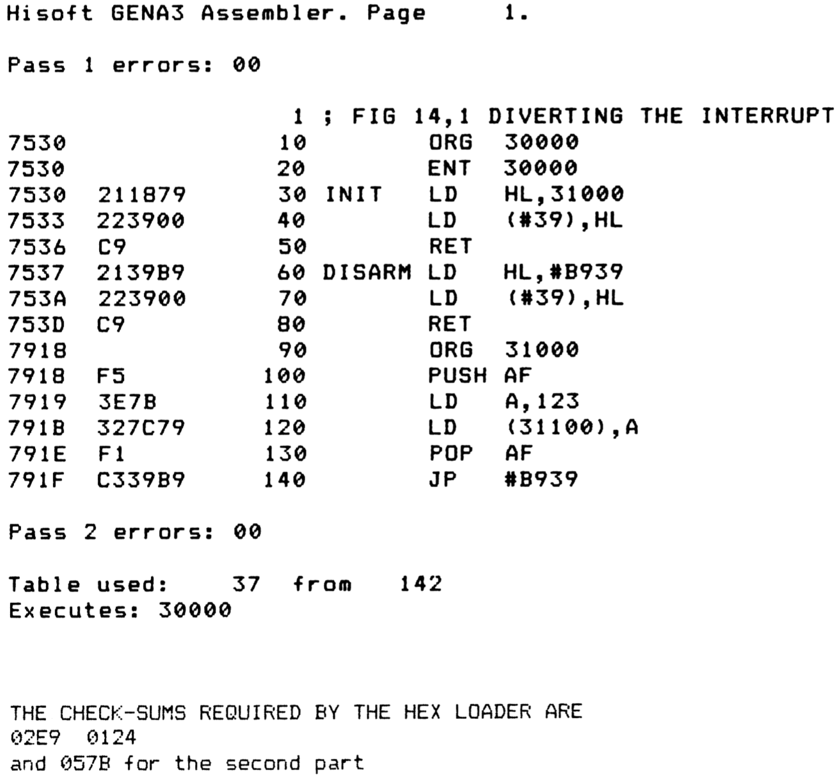
CALL 30000

and try the above again.

If the interrupt routine is being executed correctly you will find that no matter what you do ? PEEK (31100) will give 123 because the interrupt routine is resetting this at every interrupt.

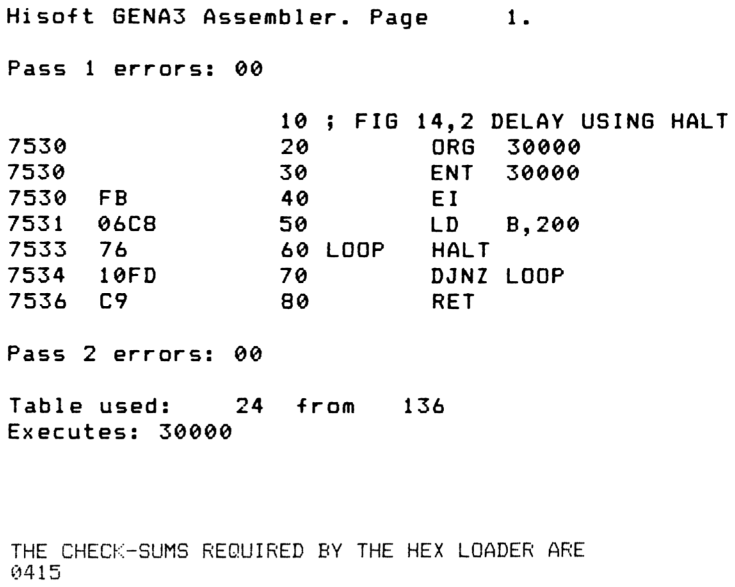
Now CALL 30007 to disarm your interrupt routine and try the above again. All should now be back to normal.

One final point about interrupts: any machine code program which can be run without being interrupted will run faster. Disabling the interrupts will also disable any BASIC timing



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                       1 ; FIG 14,1 DIVERTING THE INTERRUPT
7530                  10          ORG  30000
7530                  20          ENT  30000
7530  211879          30 INIT     LD   HL,31000
7533  223900          40          LD   (#39),HL
7536  C9              50          RET
7537  2139B9          60 DISARM   LD   HL,#B939
753A  223900          70          LD   (#39),HL
753D  C9              80          RET
7918                  90          ORG  31000
7918  F5             100          PUSH AF
7919  3E7B           110          LD   A,123
791B  327C79         120          LD   (31100),A
791E  F1             130          POP  AF
791F  C339B9         140          JP   #B939

Pass 2 errors: 00

Table used:    37  from   142
Executes: 30000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
02E9  0124
and 057B for the second part
```

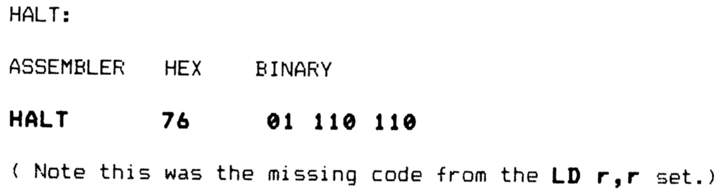
Figure 14.1



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                     10 ; FIG 14,2 DELAY USING HALT
7530                 20          ORG  30000
7530                 30          ENT  30000
7530  FB             40          EI
7531  06C8           50          LD   B,200
7533  76             60 LOOP     HALT
7534  10FD           70          DJNZ LOOP
7536  C9             80          RET

Pass 2 errors: 00

Table used:    24  from   136
Executes: 30000



THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
0415
```

Figure 14.2

functions, as well as the AFTER and EVERY commands, but very little else will be affected.



```
HALT:

ASSEMBLER   HEX    BINARY

HALT        76     01 110 110

( Note this was the missing code from the LD r,r set.)
```

This seems like a suitable instruction to explain following the last paragraph. The HALT instruction stops the CPU from doing anything until the next interrupt is received. If executed when the interrupts are disabled the computer will go completely to sleep, so be sure that the interrupts are *definitely on* when the HALT instruction is used.

The most common use of the HALT is to allow long delays to be achieved without multiple loops in the delay program. Figure 14.2 shows a program which uses the HALT instruction for this purpose.

You can see the time difference for the delay given by the HALT instruction if you POKE &7533,0, which replaces the HALT with a NOP, and then execute the program again.

RST

The CPU has eight addresses that can be called by a special single byte instruction, known as a ReSTart. The Amstrad uses most of these for its own purposes, making them of no practical use to you, the programmer. They have, however, left one ReSTart available for your use. This is RST 30h.

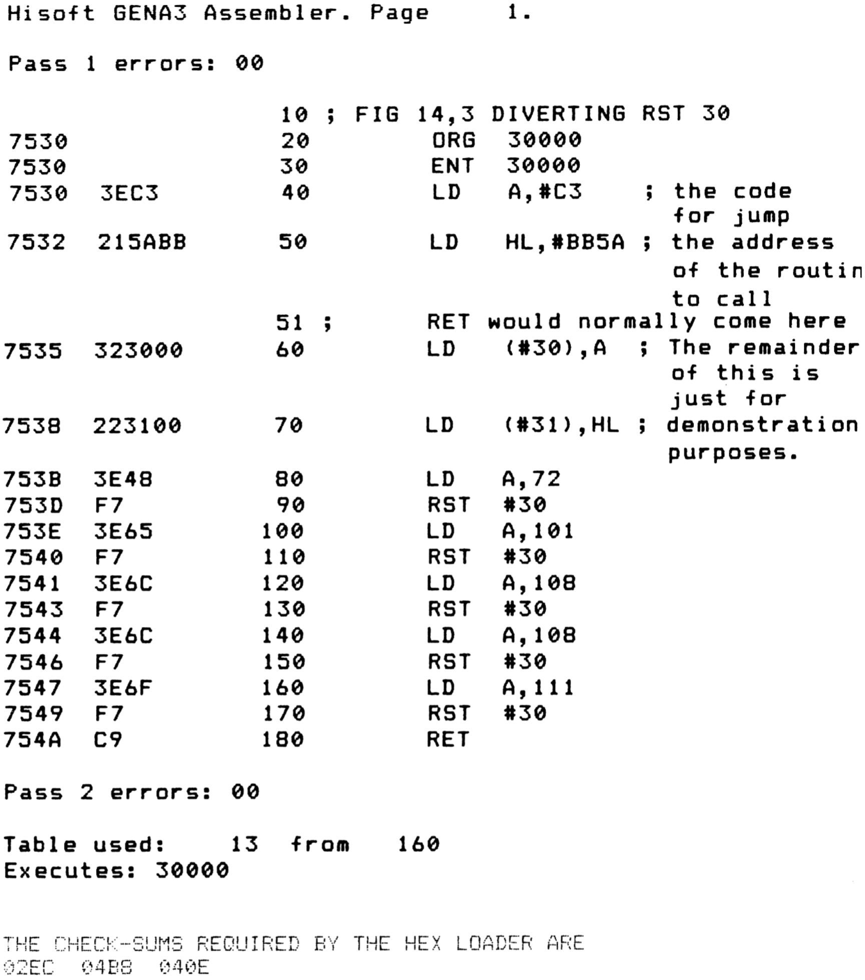
The address following the RST is the address called, when the instruction is executed. This acts in an identical manner to a normal CALL and the routine called by a RST should be ended with a RET, exactly as it would if a CALL had been used.

ASSEMBLER	BINARY	p	t	p	t
RST p	**11 t 111**	00h	000	20h	100
		08h	001	28h	101
RST 30h	**11 110 111**	10h	010	30h	110
		18h	011	38h	111

You will have noticed that the last ReSTart is the address used by the interrupt routine, and the method employed to gain access to the user ReSTart is almost identical to the manner in which the interrupt was diverted.

The Cold Start routine sets RST 30h to jump back to the cold start routine if it is used. If you are brave you can test this by CALLing address 56 (38h); this will cause a system reset.

To change it to jump to your routine you must put a jump instruction at address 30h to call your routine. You could, for example, decide that you were calling the PRINT routine at 47962 (BB5Ah) so often, that it would be worth while to use the RST instead, saving 2 bytes for every CALL. To do this you would put the code for JP into address 30h, and the address to be jumped to in addresses 31h and 32h, in normal Z80 low byte first fashion. Fig. 14.3 shows this being done, note that the RET in line 51 is part of a comment and not an instruction.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                        10 ; FIG 14,3 DIVERTING RST 30
7530                    20       ORG  30000
7530                    30       ENT  30000
7530  3EC3              40       LD   A,#C3     ; the code
                                                  for jump
7532  215ABB            50       LD   HL,#BB5A ; the address
                                                  of the routin
                                                  to call
                        51 ;     RET would normally come here
7535  323000            60       LD   (#30),A   ; The remainder
                                                  of this is
                                                  just for
7538  223100            70       LD   (#31),HL ; demonstration
                                                  purposes.
753B  3E48              80       LD   A,72
753D  F7                90       RST  #30
753E  3E65             100       LD   A,101
7540  F7               110       RST  #30
7541  3E6C             120       LD   A,108
7543  F7               130       RST  #30
7544  3E6C             140       LD   A,108
7546  F7               150       RST  #30
7547  3E6F             160       LD   A,111
7549  F7               170       RST  #30
754A  C9               180       RET

Pass 2 errors: 00

Table used:    13  from   160
Executes: 30000


THE CHECK-SUMS REQUIRED BY THE HEX LOADER ARE
02EC  04B8  040E
```

Figure 14.3

None of the instructions in this chapter so far, affects any flags in any way.

Indexed Addressing

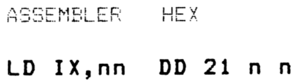
There are two registers of which no mention whatsoever has been made so far; these are the IX and the IY registers. The I in each of their names stands for Index. They are therefore Index X and Index Y and, as with any normal index, they are used to tell where a particular piece of information can be found. As well as serving as index registers they can also be used for anything that the HL register pair can be used for.

'How,' you will ask, 'can a single register be used to replace a register pair?' The answer is simple, they are not really single registers but two 8 bit registers used in unison. The manufacturers of the Z80 CPU were unable to get reliable results from these two pairs of registers, when used individually, so they do not publish the instructions to use each register independently, because any or all of these instructions may not work.

On the Amstrad used for developing this book all the single register instructions that were tried on the Index register pairs worked. Even though these instructions are not published it is extremely easy to discover what they are, once you know how all the instructions that use these registers are made up. Those of you who have an assembler will be reduced to almost the same level of programming as those without, if you wish to employ the two halves of the index registers separately. Apart from this, there is no way that it can be guaranteed that a program using these extra instructions will work on another Amstrad CPC464, so it is probably best to leave well alone. Having got that out of the way use of the index registers can be detailed.

Any instruction apart from ADC and SBC which uses the HL register pair can be made to act on the IX register instead, by simply placing DDh (221) in front of the instruction opcode for HL or, to use the IY register, put FDh (253) in front of the HL instruction. When the instruction is using the index register to point to an address in memory an extra byte is needed after the first byte of the original instruction. This extra byte gives a signed displacement from the address pointed to be the register. This probably seems a bit confusing; look at the examples below before giving up.

The Hex instruction to LD HL,nn is 21, and this is followed by two bytes which provide the 16 bit number nn. To change this instruction so as to make it operate on the IX register you would prefix it with DDh. The instruction in has now become:



```
ASSEMBLER    HEX

LD IX,nn   DD 21 n n
```

To make it operate on the IY register pair instead of the IX, the instruction is:

ASSEMBLER	HEX
LD IY,nn	FD 21 n n

All instructions which act directly on the index registers have the same format. Some examples are given below.

ASSEMBLER	HEX	WITH IX	WITH IY
LD (nn),HL	22 n n	DD 22 n n	FD 22 n n
PUSH HL	E5	DD E5	FD E5
DEC HL	2B	DD 2B	FD 2B
JP (HL)	E9	DD E9	FD E9
ADD HL,BC	09	DD 09	FD 09

This last instruction raises an interesting point. What do you suppose will happen when the instruction: ADD HL,HL is modified to act on the IX or IY register? This was the instruction which could have been employed to perform a 16 bit left shift in the multiplication programs in Chapter 11.

If the instruction became ADD IY,HL when prefixed by FDh and ADD IX,HL when DDh was put in front it would not be a direct replacement (albeit using one extra byte) for the instruction using HL. In fact any reference to the HL register pair is altered to refer to the index register when an opcode using the HL register pair is preceded by either of the two index register prefixes. ADD HL,HL becomes ADD IX,IX when prefixed by DDh or, with the FDh prefix, ADD IY,IY.

So far no instruction which uses HL to point to memory has been shown. This is because of the added complication of the Indexed addressing. As previously mentioned there is an extra byte needed to give a signed displacement from the actual address pointed to by the register.

Suppose, for a moment, that you have written a Database program, and each record has a sort of header, telling the length of each piece of information in the record. Taking a very simple example (which book and magazine writers seem to think all micro-users will want, though heaven knows why!), the *Address Book*.

There is no way in which you can say that the length of a person's name, the number of lines in the address and the length of these lines, or the phone number, will be the same as for another person. This can be allowed for in two basic ways.

1) You allow each record to take up space which will allow for the longest name and address.
2) You keep a record of the length of each line, and the number of lines, and also the total length of the record.

Method 1 is very wasteful on space, every record uses the same amount of memory, but keeping track of the details of each record using method 2 could be a problem. Not with the Index registers though!

If you start the records at a known address, say 10000, you could have an index at the start of each record such as that below.

ADDRESS	DETAIL
10000/1	length of this record, in 16 bits
10002	length of the name
10003	length of address line 1
10004	length of address line 2
10005	length of address line 3
10006	length of address line 4
10007	length of address line 5
10008	length of phone number

If entry 1 was Martin Pudwick 14
27 New Road
Mudford
Sussex
0123 456789

10000 = 49 the total length of the record, in Z80 low byte
10001 = 0 first form.
10002 = 14 name
10003 = 11 address 1
10004 = 7 address 2
10005 = 6 address 3
10006 = 0 address 4
10007 = 0 address 5
10008 = 11 phone number

Nine bytes will always be used for the index; so 49 + 9 = 58, the start of the index for the next record will be at 10058. Now you can make use of the index registers.

If IX is loaded with 10000 then (IX + 0) and (IX + 1) will be the total length of the record, (IX + 2) will be the length of the name, and so on. You can write your program to increase the total length for every character added to any line of the record, and increase the line being worked on in the index as well, then when you want to move on to the next record, all you have to do is add (IX + 0) and (IX + 1) to the IX register, to have the details of this record available, and ready for use by the same program, without modification.

The instructions in assembler, to use the index registers with a displacement, have exactly the form you would expect. Instead of LD A,(HL) the instruction becomes LD A,(IX + d) when DDh is placed before the original opcode, and LD A,(IY + d) when FDh is used. d is any value from −128 to +127.

The displacement is mandatory for all instructions which use index registers to address memory, even when there is a 0 displacement, and the byte which holds this displacement always comes immediately after the first byte of the original opcode. For example:

LD A,(HL) is 7Eh LD A,(IX + d) is DDh 7Eh d
INC (HL) is 34h INC (IY + d) is FDh 34h d

The rotates and shifts, as well as the bit set, reset and test group of instructions already have a prefix, CBh, but this makes no difference, the displacement byte still follows the first byte of the opcode:

RLC (HL) is CBh 06h and RLC (IX + d) is DDh CBh d 06h
SET 4,(HL) is CBh E6h and SET 4,(IY + d) is FDh CBh d E6h

The same applies where a number is involved:

LD (HL),n is 36h n and LD (IX + d),n is DDh 36h d n

One further use for the index registers is for changing the axis of the screen, to enable a screen dump to be output to the printer. Printers require a byte to hold information vertically, whereas the screen uses a byte horizontally. In mode 2 one byte holds the information for eight pixels across the screen, an Epson printer needs the eight pixels from the same position on the Y axis but from eight consecutive places on the X axis. A screen would need to be rotated by 90 degrees before it could be copied directly to the printer.

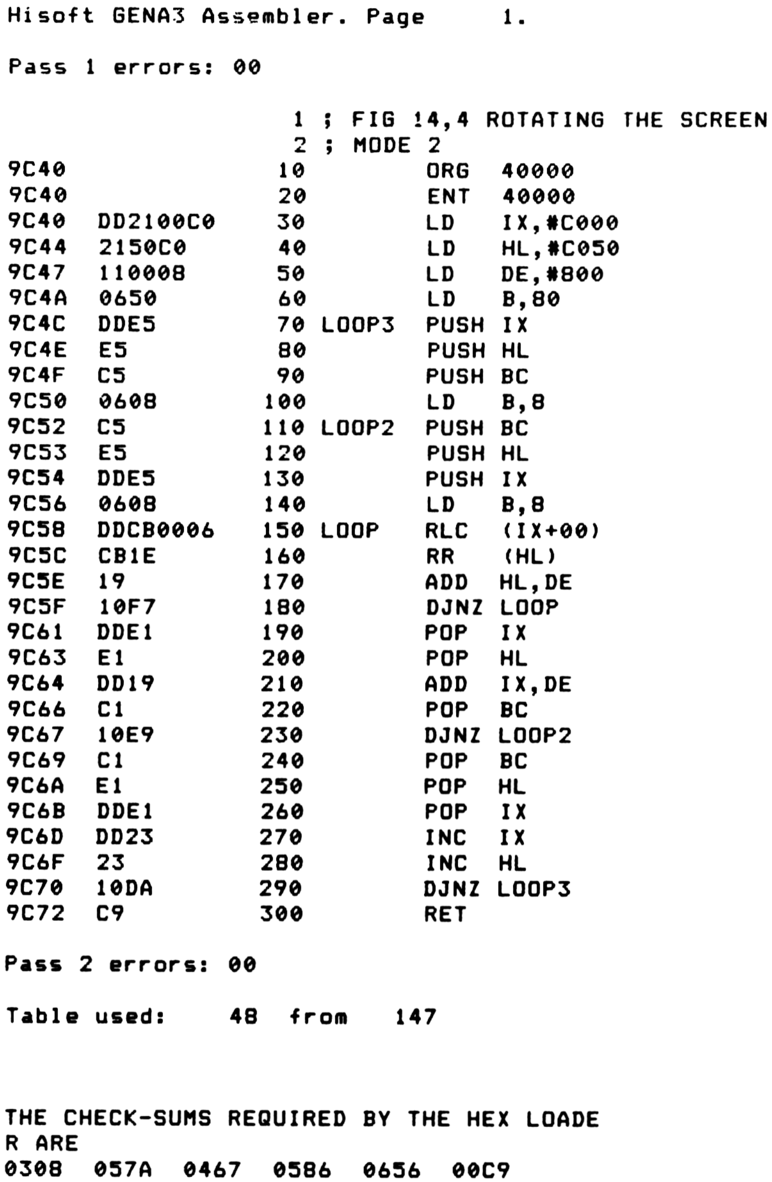
The most economical way of doing this is not to actually rotate on the screen, but to reserve a space in memory and rotate each byte in turn into that, until you have a screen character line ready to be output to the printer, and then output it, and repeat for the next character line. Figure 14.4 shows this being done for the screen line starting at C000h, in mode 2. Because not everyone will have a printer, and even if they did many printers require different control codes to be put into graphics mode, always assuming they can be used for graphics printing, this program operates on the screen.

The screen map moves whenever the screen scrolls, so to make sure it is set to start at C000h, press W until you find yourself in mode 2, if you are using the assembler, or enter mode 2 as a direct command if you are using the HEX LOADER. Do not list to get something on the screen but move the cursor to the top line and write some characters there. Next, again without causing the screen to scroll, press [ENTER]. You will get an error message; don't worry. Now execute the program. You will find a copy of the first character line appears on line 2, but each character is rotated 90 degrees clockwise.

This program is completely relocatable so you can incorporate it into a program of your own, if you are writing a screen dump or some similar program. Note that screen dumps are easy in mode 2 but become quite tricky in other modes, because one bit does not represent one pixel.

That just about finishes this book, and all that is left is for you to put into practice what you have learnt. The next chapter will give you some hints, and the addresses to CALL to use some of the operating system routines. The appendices are there to help you with routines that you are writing, saving you from searching high and low for a simple answer to a query.

If you are going to take up programming seriously it will help if you buy a stencil for flow charts, both W. H. Smiths and Menzies do them at about two pounds, so don't pay much more than this, and an almost inestimable assistance can be given by a calculator which can work in Hex and binary as well as the normal decimal system. The Casio fx450 is quite excellent (as well as being half the price of the Texas offering) and costs about twenty pounds.



```
Hisoft GENA3 Assembler. Page     1.

Pass 1 errors: 00

                         1 ; FIG 14,4 ROTATING THE SCREEN
                         2 ; MODE 2
9C40                    10          ORG   40000
9C40                    20          ENT   40000
9C40   DD2100C0         30          LD    IX,#C000
9C44   2150C0           40          LD    HL,#C050
9C47   110008           50          LD    DE,#800
9C4A   0650             60          LD    B,80
9C4C   DDE5             70 LOOP3    PUSH  IX
9C4E   E5               80          PUSH  HL
9C4F   C5               90          PUSH  BC
9C50   0608            100          LD    B,8
9C52   C5              110 LOOP2    PUSH  BC
9C53   E5              120          PUSH  HL
9C54   DDE5            130          PUSH  IX
9C56   0608            140          LD    B,8
9C58   DDCB0006        150 LOOP     RLC   (IX+00)
9C5C   CB1E            160          RR    (HL)
9C5E   19              170          ADD   HL,DE
9C5F   10F7            180          DJNZ  LOOP
9C61   DDE1            190          POP   IX
9C63   E1              200          POP   HL
9C64   DD19            210          ADD   IX,DE
9C66   C1              220          POP   BC
9C67   10E9            230          DJNZ  LOOP2
9C69   C1              240          POP   BC
9C6A   E1              250          POP   HL
9C6B   DDE1            260          POP   IX
9C6D   DD23            270          INC   IX
9C6F   23              280          INC   HL
9C70   10DA            290          DJNZ  LOOP3
9C72   C9              300          RET

Pass 2 errors: 00

Table used:     48  from   147


THE CHECK-SUMS REQUIRED BY THE HEX LOADE
R ARE
0308   057A   0467   0586   0656   00C9
```

Figure 14.4

Many Rymans branches stock it but not many other places. This calculator will also perform the logical AND OR XOR as well as NEG and CPL (which they call NOT).

Chapter Fifteen

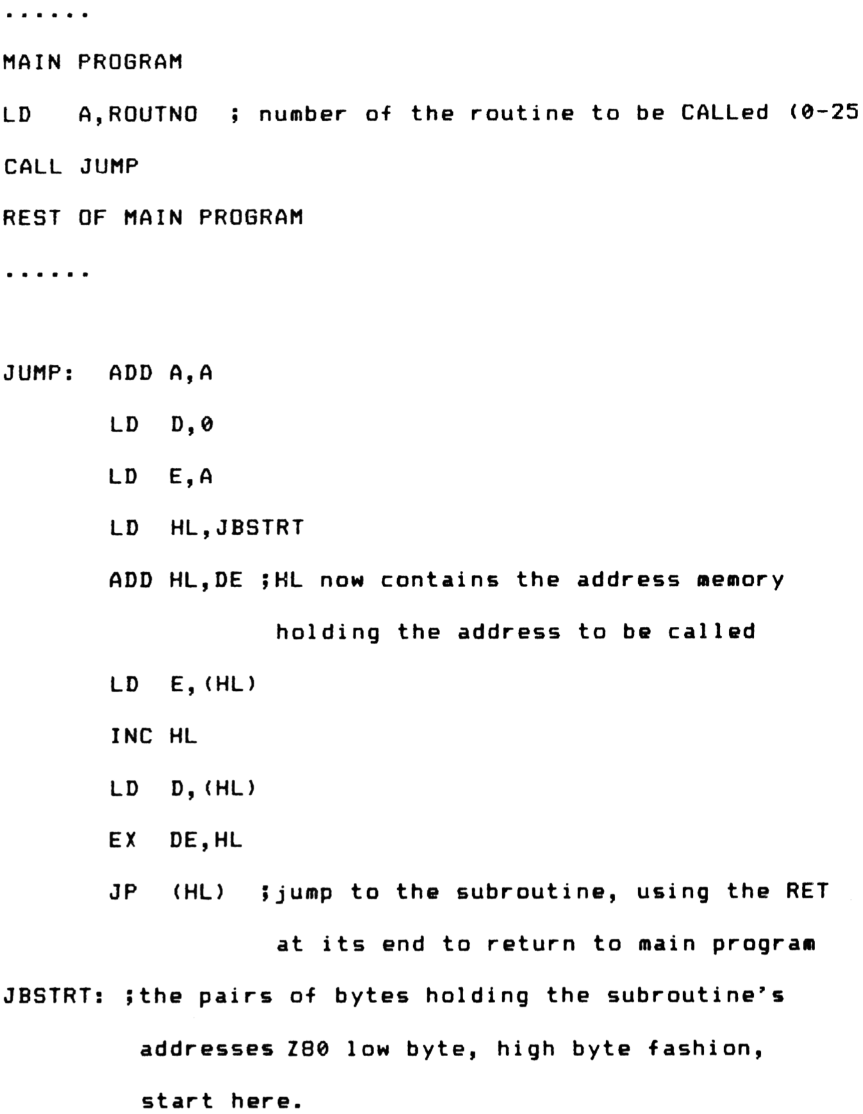
Programming Hints, and Using the Firmware

The Amstrad's operating system can be broken down into nine fundamental areas. A few routines from each of these areas will give you a good start in your programming. You have already been introduced to two routines, 'text output' and 'wait key', which have been referred to as 'GETKEY' and 'PRINT' respectively. 'Text output' is part of the Key Manager, and 'wait key' is one of the Key Manager routines. The other seven areas are:

The Graphics VDU
The Screen Pack
The Cassette Manager
The Sound Manager
The Kernel
The Machine Pack
The Jumper.

It is not the purpose of this book to go into detail of how these sections are constructed, nor to describe the operating system. The Programmer's Reference Manual covers these in great detail, but in order to get you started here are some pointers that you may find helpful.

Access to the operating system is, for the greater part, given by the programmer CALLing addresses which are located in RAM (Random Access Memory), known as 'Jump-blocks'. A jump-block is often made up from just a series of addresses, ready to be loaded into the HL register pair, before a JP (HL) instruction. This is shown in Fig. 15.1, as it is a useful technique which you may wish to employ in your programs.



```
......

MAIN PROGRAM

LD   A,ROUTNO  ; number of the routine to be CALLed (0-25

CALL JUMP

REST OF MAIN PROGRAM

......


JUMP:   ADD A,A

        LD  D,0

        LD  E,A

        LD  HL,JBSTRT

        ADD HL,DE ;HL now contains the address memory

                   holding the address to be called

        LD  E,(HL)

        INC HL

        LD  D,(HL)

        EX  DE,HL

        JP  (HL)  ;jump to the subroutine, using the RET

                   at its end to return to main program

JBSTRT: ;the pairs of bytes holding the subroutine's

         addresses Z80 low byte, high byte fashion,

         start here.
```

Figure 15.1

One reason for using jump-blocks is that it is possible to alter the way all uses of a particular routine are dealt with, by simply changing the address in the jump-block, instead of having to go through the program and alter every occurrence of a CALL to the handling routine. They also have the advantage of allowing a program to calculate its own actions.

A typical case where a jump-block would be advantageous is, if a program was to use the printer, or a different window, for all its output, from that originally chosen. This could allow debugging, and running on the screen, with the printer only used when required. In BASIC you might use a PRINT statement followed by a variable, which could be assigned to the stream number which you wish to use at a particular time. With a machine code program there is no operating system to keep track of variables, and respond globally to resetting of one, hence the jump-block.

Your Amstrad cannot use the system above, because of the ability to switch areas of memory between ROMs and RAM. Unless the programmer were to actually check that the ROM containing the routine to be CALLed was switched in, and switch it in if not, there can be no guarantee that the right routine will be accessed! Furthermore, if the screen needs to be read, the Upper ROM, which occupies the same addresses, must be switched out.

Amstrad's solution to this is to dedicate a number of the ReSTarts to CALLing ROM routines. These ensure that the correct ROM is switched in and that the screen is available for reading if required. They POP the return address from the restart instruction off the stack and then use this to look at the bytes following the restart, which give the address of the routine to be called and the ROM state to be set. The full details of how these ReSTarts operate are given in the Programmer's Reference Manual, but it is unlikely that you will need to know any more about the way they work, until you have done a great deal more programming. Their actions are totally transparent to the user, and ROM settings are restored to their status before the jump-block was used, on return to your program.

Any of the many routines available in the operating system can be used by making a straightforward CALL to a jump-block. The address returned to by the return at the end of the ROM routine, is that put there by your call to the jump-block; there is NO return to the jump-block ReSTart.

There are two distinctly different jump-blocks, one which is used by the basic interpreter, and the other is used by the firmware, or operating system. You can alter either set by changing the three bytes, starting with the RST address. If the new routine to be accessed is in the firmware, copy the three bytes in the jump table for new routine to replace the old entry. If the new routine is your own simply replace the entry in the jump table with a JP to the address of the routine you want to be CALLed to handle the task, and as long as you end this routine with a RET things will continue to happen 'correctly'. Changing the main jump-block, which lies between BB00h and BDCBh inclusive, will not affect the workings of any routine in the operating system. Altering any entries in jump-blocks outside this area may have unforeseen side-effects. This is because routines may use these other jump-blocks to call other routines, which are necessary to complete a task.

If (when?) you get into such a pickle that you haven't a clue what is calling what, where, how, when and why, there is one entry to the jump-block which is vital. This resets every jump to its original destination! CALL BD37h.

The screen and the sound are dealt with by hardware as opposed to software, and this makes it quite tricky to access directly the functions you may wish to employ, as well as rendering software keyboard scanning nigh-on impossible. Fortunately there are inbuilt firmware facilities available to perform these tasks for you. The joysticks can be read for any combination of direction and fire button, enabling movement in eight directions to be detected. Some of the fundamental firmware jump addresses are given in the appendix.

Try always to use labels in your programs which relate to the section which they are used for. Whilst excessive use of comments is not only undesirable, but wasteful on space, do remember that you may come back to the source code at some time in the future, without a clue of how you organised the program.

Please note that this book has made no attempt whatsoever to teach any of the finer details of how the Z80 works, nor have timing considerations been examined. The aim was to get you started in programming, to understand the instructions set and be able to use the firmware. If you wish to delve deeper into the finer nuances of Z80 programming, or investigate the hardware operation, books that can be recommended are:

1) For the programmer *Programming the Z80* by Rodney Zaks.
2) For hardware design and expansion *The Z80 Technical Manual* from Zilog and *Z80 Applications* by James Coffron.

The Zaks and Coffron books are both Sybex publications, and are not cheap (over ten pounds each) so make sure they are what you really want before buying.

Once you become adept at assembly language programming you may wish to consider learning other languages. Pascal is probably the best 'second tongue' to learn for serious programming, being almost BASIC-like in many respects, but offering many of the facilities of assembler as well. It is a compiled language like assembler, and a program written in Pascal is normally very portable in its source code form. Many computers have implementations written for them, and a source file can be compiled to run on any computer with a Pascal compiler with the minimum of alteration. The main limitations of Pascal are speed, assembler leaves it standing, and space, where assembler is much more economical. Pascal itself is much much faster than BASIC and the compiled code is less space consuming than a comparable BASIC program. Many programmers use a combination of Pascal generated code and true machine code, compiled from assembly language, where speed or economy of space is at a premium.

Both Pascal and assembly language, once compiled, can be run without the program used to write them being present, and the source code can be saved for further use as required, whilst only the object code need be saved and loaded for using the program. There is an implementation of Pascal available for the Amstrad, written by Highsoft.

Appendix A

The Z80 Instruction Set Courtesy of ZILOG Inc.

Instruction Set

The Z80 microprocessor has one of the most powerful and versatile instruction sets available in any 8-bit microprocessor. It includes such unique operations as a block move for fast, efficient data transfers within memory or between memory and I/O. It also allows operations on any bit in any location in memory.

The following is a summary of the Z80 instruction set and shows the assembly language mnemonic, the operation, the flag status, and gives comments on each instruction. The *Z80 CPU Technical Manual* (03-0029-01) and *Assembly Language Programming Manual* (03-0002-01) contain significantly more details for programming use.

The instructions are divided into the following categories:

□ 8-bit loads
□ 16-bit loads
□ Exchanges, block transfers, and searches
□ 8-bit arithmetic and logic operations
□ General-purpose arithmetic and CPU control
□ 16-bit arithmetic operations
□ Rotates and shifts
□ Bit set, reset, and test operations
□ Jumps
□ Calls, returns, and restarts
□ Input and output operations.

A variety of addressing modes are implemented to permit efficient and fast data transfer between various registers, memory locations, and input/output devices. These addressing modes include:

- □ Immediate
- □ Immediate extended
- □ Modified page zero
- □ Relative
- □ Extended
- □ Indexed
- □ Register
- □ Register indirect
- □ Implied
- □ Bit

8-bit Load Group

Mnemonic	Symbolic Operation	S	Z		Flags H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
LD r, r'	r ← r'	•	•	X	•	X	•	•	•	01 r r'		1	1	4	r, r' Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110 ← n →		2	2	7	
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	
LD r, (IX + d)	r ← (IX + d)	•	•	X	•	X	•	•	•	11 011 101 01 r 101 ← d →	DD	3	5	19	
LD r, (IY + d)	r ← (IY + d)	•	•	X	•	X	•	•	•	11 111 101 01 r 110 ← d →	FD	3	5	19	
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7	
LD (IX + d), r	(IX + d) ← r	•	•	X	•	X	•	•	•	11 011 101 01 110 r ← d →	DD	3	5	19	
LD (IY + d), r	(IY + d) ← r	•	•	X	•	X	•	•	•	11 111 101 01 110 r ← d →	FD	3	5	19	
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110 ← n →	36	2	3	10	
LD (IX + d), n	(IX + d) ← n	•	•	X	•	X	•	•	•	11 011 101 00 110 110 ← d → ← n →	DD 36	4	5	19	
LD (IY + d), n	(IY + d) ← n	•	•	X	•	X	•	•	•	11 111 101 00 110 110 ← d → ← n →	FD 36	4	5	19	
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7	
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7	
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010 ← n → ← n →	3A	3	4	13	
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7	
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7	
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010 ← n → ← n →	32	3	4	13	
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101 01 010 111	ED 57	2	2	9	
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101 01 011 111	ED 5F	2	2	9	
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101 01 000 111	ED 47	2	2	9	
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101 01 001 111	ED 4F	2	2	9	

NOTES: r, r' means any of the registers A, B, C, D, E, H, L.
IFF the content of the interrupt enable flip-flop, (IFF) is copied into the P/V flag.
For an explanation of flag notation and symbols for mnemonic tables, see Symbolic Notation section following tables.

16-bit Load Group

Mnemonic	Symbolic Operation	S	Z	Flags	H		P/V	N	C	76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn + 1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_H ← (nn + 1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn + 1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn + 1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn + 1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn + 1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn + 1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn + 1) ← IY_H (nn) ← IY_L	•	•	X	•	X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 11 111 001	FD F9	2	2	10	
PUSH qq	(SP − 2) ← qq_L (SP − 1) ← qq_H SP → SP − 2	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	qq Pair 00 BC 01 DE 10 HL 11 AF
PUSH IX	(SP − 2) ← IX_L (SP − 1) ← IX_H SP → SP − 2	•	•	X	•	X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	
PUSH IY	(SP − 2) ← IY_L (SP − 1) ← IY_H SP → SP − 2	•	•	X	•	X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	qq_H ← (SP + 1) qq_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10	
POP IX	IX_H ← (SP + 1) IX_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	
POP IY	IY_H ← (SP + 1) IY_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

NOTES: dd is any of the register pairs BC, DE, HL, SP.
qq is any of the register pairs AF, BC, DE, HL.
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g., BC_L = C, AF_H = A.

Exchange, Block Transfer, Block Search Groups

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	X	•	X	•	•	•	11 101 011	EB	1	1	4	
EX AF, AF′	AF ↔ AF′	•	•	X	•	X	•	•	•	00 001 000	08	1	1	4	
EXX	BC ↔ BC′ DE ↔ DE′ HL ↔ HL′	•	•	X	•	X	•	•	•	11 011 001	D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP + 1) L ↔ (SP)	•	•	X	•	X	•	•	•	11 100 011	E3	1	5	19	
EX (SP), IX	IX_H ↔ (SP + 1) IX_L ↔ (SP)	•	•	X	•	X	•	•	•	11 011 101 11 100 011	DD E3	2	6	23	
EX (SP), IY	IY_H ↔ (SP + 1) IY_L ↔ (SP)	•	•	X	•	X	•	•	•	11 111 101 11 100 011	FD E3	2	6	23	
LDI	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1	•	•	X	0	X	① ↕	0	•	11 101 101 10 100 000	ED A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE + 1 HL ← HL + 1 BC ← BC − 1 Repeat until BC = 0	•	•	X	0	X	① 0	0	•	11 101 101 10 110 000	ED B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0

NOTE: ① P/V flag is 0 if the result of BC − 1 = 0, otherwise P/V = 1

Mnemonic	Symbolic Operation	S	Z	Flags	H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
LDD	(DE) ← (HL) DE ← DE − 1 HL ← HL − 1 BC ← BC − 1	•	•	X	0	X	① ↕	0	•	11 101 101 10 101 000	ED A8	2	4	16	
LDDR	(DE) ← (HL) DE ← DE − 1 HL ← HL − 1 BC ← BC − 1 Repeat until BC = 0	•	•	X	0	X	② 0	0	•	11 101 101 10 111 000	ED B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A − (HL) HL ← HL + 1 BC ← BC − 1	↕	③ ↕	X	↕	X	① ↕	1	•	11 101 101 10 100 001	ED A1	2	4	16	
CPIR	A − (HL) HL ← HL + 1 BC ← BC − 1 Repeat until A = (HL) or BC = 0	↕	③ ↕	X	↕	X	① ↕	1	•	11 101 101 10 110 001	ED B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A − (HL) HL ← HL − 1 BC ← BC − 1	↕	③ ↕	X	↕	X	① ↕	1	•	11 101 101 10 101 001	ED A9	2	4	16	
CPDR	A − (HL) HL ← HL − 1 BC ← BC − 1 Repeat until A = (HL) or BC = 0	↕	③ ↕	X	↕	X	① ↕	1	•	11 101 101 10 111 001	ED B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

NOTES: ① P/V flag is 0 if the result of BC − 1 = 0, otherwise P/V = 1.
② P/V flag is 0 at completion of instruction only.
③ Z flag is 1 if A = (HL), otherwise Z = 0.

8-bit Arithmetic and Logical Group

ADD A, r	A ← A + r	↕	↕	X	↕	X	V	0	↕	10 $\boxed{000}$ r		1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	X	↕	X	V	0	↕	11 $\boxed{000}$ 110 ← n →		2	2	7	000 B 001 C 010 D
ADD A, (HL)	A ← A + (HL)	↕	↕	X	↕	X	V	0	↕	10 $\boxed{000}$ 110		1	2	7	011 E
ADD A, (IX + d)	A ← A + (IX + d)	↕	↕	X	↕	X	V	0	↕	11 011 101 10 $\boxed{000}$ 110 ← d →	DD	3	5	19	100 H 101 L 111 A
ADD A, (IY + d)	A ← A + (IY + d)	↕	↕	X	↕	X	V	0	↕	11 111 101 10 $\boxed{000}$ 110 ← d →	FD	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	X	↕	X	V	0	↕	$\boxed{001}$					s is any of r, n, (HL), (IX + d), (IY + d) as shown for ADD instruction. The indicated bits replace the $\boxed{000}$ in the ADD set above.
SUB s	A ← A − s	↕	↕	X	↕	X	V	1	↕	$\boxed{010}$					
SBC A, s	A ← A − s − CY	↕	↕	X	↕	X	V	1	↕	$\boxed{011}$					
AND s	A ← A ∧ s	↕	↕	X	1	X	P	0	0	$\boxed{100}$					
OR s	A ← A ∨ s	↕	↕	X	0	X	P	0	0	$\boxed{110}$					
XOR s	A ← A ● s	↕	↕	X	0	X	P	0	0	$\boxed{101}$					
CP s	A − s	↕	↕	X	↕	X	V	1	↕	$\boxed{111}$					
INC r	r ← r + 1	↕	↕	X	↕	X	V	0	•	00 r $\boxed{100}$		1	1	4	
INC (HL)	(HL) ← (HL) + 1	↕	↕	X	↕	X	V	0	•	00 110 $\boxed{100}$		1	3	11	
INC (IX + d)	(IX + d) ← (IX + d) + 1	↕	↕	X	↕	X	V	0	•	11 011 101 00 110 $\boxed{100}$ ← d →	DD	3	6	23	
INC (IY + d)	(IY + d) ← (IY + d) + 1	↕	↕	X	↕	X	V	0	•	11 111 101 00 110 $\boxed{100}$ ← d →	FD	3	6	23	
DEC m	m ← m − 1	↕	↕	X	↕	X	V	1	•	$\boxed{101}$					m is any of r, (HL), (IX + d), (IY + d) as shown for INC. DEC same format and states as INC. Replace $\boxed{100}$ with $\boxed{101}$ in opcode.

General-purpose Arithmetic and CPU Control Groups

Mnemonic	Symbolic Operation	S	Z		Flags H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands.	↕	↕	X	↕	X	P	•	↕	00 100 111	27	1	1	4	Decimal adjust accumulator.
CPL	A ← $\bar{A}$	•	•	X	1	X	•	1	•	00 101 111	2F	1	1	4	Complement accumulator (one's complement).
NEG	A ← 0 – A	↕	↕	X	↕	X	V	1	↕	11 101 101 01 000 100	ED 44	2	2	8	Negate acc. (two's complement).
CCF	CY ← $\overline{CY}$	•	•	X	X	X	•	0	↕	00 111 111	3F	1	1	4	Complement carry flag.
SCF	CY ← 1	•	•	X	0	X	•	0	1	00 110 111	37	1	1	4	Set carry flag.
NOP	No operation	•	•	X	•	X	•	•	•	00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	•	•	01 110 110	76	1	1	4	
DI ★	IFF ← 0	•	•	X	•	X	•	•	•	11 110 011	F3	1	1	4	
EI ★	IFF ← 1	•	•	X	•	X	•	•	•	11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	•	•	11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	•	•	11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	•	•	11 101 101 01 011 110	ED 5E	2	2	8	

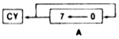
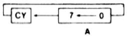
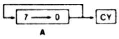
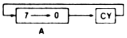
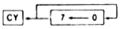
NOTES: IFF indicates the interrupt enable flip-flop.
CY indicates the carry flip-flop.
★ indicates interrupts are not sampled at the end of EI or DI.

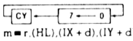
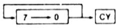
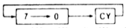
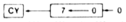
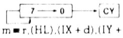
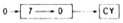
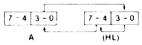
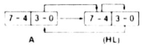
16-bit Arithmetic Group

ADD HL, ss	HL ← HL + ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss Reg. 00 BC 01 DE 10 HL 11 SP
ADC HL, ss	HL ← HL + ss + CY	↕	↕	X	X	X	V	0	↕	11 101 101 01 ss1 010	ED	2	4	15	
SBC HL, ss	HL ← HL – ss – CY	↕	↕	X	X	X	V	1	↕	11 101 101 01 ss0 010	ED	2	4	15	
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101 01 pp1 001	DD	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101 00 rr1 001	FD	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6	
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	
DEC ss	ss ← ss – 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6	
DEC IX	IX ← IX – 1	•	•	X	•	X	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	
DEC IY	IY ← IY – 1	•	•	X	•	X	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

NOTES: ss is any of the register pairs BC, DE, HL, SP.
pp is any of the register pairs BC, DE, IX, SP.
rr is any of the register pairs BC, DE, IY, SP.

Rotate and Shift Group

RLCA		•	•	X	0	X	•	0	↕	00 000 111	07	1	1	4	Rotate left circular accumulator.
RLA		•	•	X	0	X	•	0	↕	00 010 111	17	1	1	4	Rotate left accumulator.
RRCA		•	•	X	0	X	•	0	↕	00 001 111	0F	1	1	4	Rotate right circular accumulator.
RRA		•	•	X	0	X	•	0	↕	00 011 111	1F	1	1	4	Rotate right accumulator.
RLC r		↕	↕	X	0	X	P	0	↕	11 001 011 00 $\boxed{000}$ r	CB	2	2	8	Rotate left circular register r. r Reg 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (HL)		↕	↕	X	0	X	P	0	↕	11 001 011 00 $\boxed{000}$ 110	CB	2	4	15	
RLC (IX + d)	 r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	11 011 101 11 001 011 ← d → 00 $\boxed{000}$ 110	DD CB	4	6	23	
RLC (IY + d)		↕	↕	X	0	X	P	0	↕	11 111 101 11 001 011	FD CB	4	6	23	

Mnemonic	Symbolic Operation	S	Z		Flags H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
RL m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	← d → 00 $\boxed{000}$ 110 $\boxed{010}$					Instruction format and states are as shown for RLC's. To form new opcode replace $\boxed{000}$ or RLC's with shown code.
RRC m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	$\boxed{001}$					
RR m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	$\boxed{011}$					
SLA m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	$\boxed{100}$					
SRA m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	$\boxed{101}$					
SRL m	 m ■ r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	$\boxed{111}$					
RLD		↕	↕	X	0	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL).
RRD		↕	↕	X	0	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	The content of the upper half of the accumulator is unaffected

Bit Set, Reset and Test Group

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
BIT b, r	$Z \leftarrow r_b$	X	↕	X	1	X	X	0	•	11 001 011 01 b r	CB	2	2	8	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A b Bit Tested 000 0 001 1 010 2 011 3 100 4 101 5 110 6 111 7
BIT b, (HL)	$Z \leftarrow (\overline{HL})_b$	X	↕	X	1	X	X	0	•	11 001 011 01 b 110	CB	2	3	12	
BIT b, (IX + d)$_b$	$Z \leftarrow (\overline{IX + d})_b$	X	↕	X	1	X	X	0	•	11 011 101 11 001 011 ← d → 01 b 110	DD CB	4	5	20	
BIT b, (IY + d)$_b$	$Z \leftarrow (\overline{IY + d})_b$	X	↕	X	1	X	X	0	•	11 111 101 11 001 011 ← d → 01 b 110	FD CB	4	5	20	
SET b, r	$r_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011 $\boxed{11}$ b r	CB	2	2	8	
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 001 011 $\boxed{11}$ b 110	CB	2	4	15	
SET b, (IX + d)	$(IX + d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 011 101 11 001 011 ← d → $\boxed{11}$ b 110	DD CB	4	6	23	
SET b, (IY + d)	$(IY + d)_b \leftarrow 1$	•	•	X	•	X	•	•	•	11 111 101 11 001 011 ← d → $\boxed{11}$ b 110	FD CB	4	6	23	
RES b, m	$m_b \leftarrow 0$ m ■ r, (HL), (IX + d), (IY + d)	•	•	X	•	X	•	•	•	$\boxed{10}$					To form new opcode replace $\boxed{11}$ of SET b, s with $\boxed{10}$. Flags and time states for SET instruction.

NOTES: The notation m_b indicates bit b (0 to 7) or location m

Jump Group

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
JP nn	PC ← nn	•	•	X	•	X	•	•	•	11 000 011 ← n → ← n →	C3	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	X	•	X	•	•	•	11 cc 010 ← n → ← n →		3	3	10	cc Condition 000 NZ non-zero 001 Z zero 010 NC non-carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
JR e	PC ← PC + e	•	•	X	•	X	•	•	•	00 011 000 ← e − 2 →	18	2	3	12	
JR C, e	If C = 0, continue If C = 1, PC ← PC + e	•	•	X	•	X	•	•	•	00 111 000 ← e − 2 →	38	2 2	2 3	7 12	If condition not met. If condition is met.
JR NC, e	If C = 1, continue If C = 0, PC ← PC + e	•	•	X	•	X	•	•	•	00 110 000 ← e − 2 →	30	2 2	2 3	7 12	If condition not met. If condition is met.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210 Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
JP Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000 28 ← e−2 →	2	2	7	If condition not met.
	If Z = 1, PC ← PC+e										2	3	12	If condition is met.
JR NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000 20 ← e−2 →	2	2	7	If condition not met.
	If Z = 0, PC ← PC+e										2	3	12	If condition is met.
JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001 E9	1	1	4	
JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101 DD 11 101 001 E9	2	2	8	
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101 FD 11 101 001 E9	2	2	8	
DJNZ, e	B ← B−1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000 10 ← e−2 →	2	2	8	If B = 0.
	If B ≠ 0, PC ← PC+e										2	3	13	If B ≠ 0.

NOTES: e represents the extension in the relative addressing mode.
e is a signed two's complement number in the range < −126, 129 >.
e−2 in the opcode provides an effective address of pc+e as PC is incremented by 2 prior to the addition of e.

Call and Return Group

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210 Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
CALL nn	(SP−1) ← PC_H (SP−2) ← PC_L PC ← nn	•	•	X	•	X	•	•	•	11 001 101 CD ← n → ← n →	3	5	17	
CALL cc, nn	If condition cc is false	•	•	X	•	X	•	•	•	11 cc 100 ← n →	3	3	10	If cc is false.
	continue, otherwise same as CALL nn									← n →	3	5	17	If cc is true.
RET	PC_L ← (SP) PC_H ← (SP+1)	•	•	X	•	X	•	•	•	11 001 001 C9	1	3	10	
RET cc	If condition cc is false	•	•	X	•	X	•	•	•	11 cc 000	1	1	5	If cc is false.
	continue, otherwise same as RET										1	3	11	If cc is true.
RETI	Return from interrupt	•	•	X	•	X	•	•	•	11 101 101 ED 01 001 101 4D	2	4	14	
RETN[1]	Return from non-maskable interrupt	•	•	X	•	X	•	•	•	11 101 101 ED 01 000 101 45	2	4	14	
RST p	(SP−1) ← PC_H (SP−2) ← PC_L PC_H ← 0 PC_L ← p	•	•	X	•	X	•	•	•	11 t 111	1	3	11	

cc		Condition
000	NZ	non-zero
001	Z	zero
010	NC	non-carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

NOTE: [1]RETN loads $IFF_2 \rightarrow IFF_1$

Input and Output Group

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210 Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
IN A, (n)	A ← (n)	•	•	X	•	X	•	•	•	11 011 011 DB ← n →	2	3	11	n to A_0 ~ A_7 Acc. to A_8 ~ A_{15}
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	•	11 101 101 ED 01 r 000	2	3	12	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INI	(HL) ← (C) B ← B−1 HL ← HL + 1	X	① ↕ ②	X	X	X	X	1	X	11 101 101 ED 10 100 010 A2	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INIR	(HL) ← (C) B ← B−1 HL ← HL + 1	X	1	X	X	X	X	1	X	11 101 101 ED 10 110 010 B2	2	5 (If B≠0)	21	C to A_0 ~ A_7 B to A_8 ~ A_{15}
	Repeat until B = 0										2	4 (If B=0)	16	
IND	(HL) ← (C) B ← B −1 HL ← HL−1	X	① ↕ ②	X	X	X	X	1	X	11 101 101 ED 10 101 010 AA	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
INDR	(HL) ← (C) B ← B−1 HL ← HL−1	X	1	X	X	X	X	1	X	11 101 101 ED 10 111 010 BA	2	5 (If B≠0)	21	C to A_0 ~ A_7 B to A_8 ~ A_{15}
	Repeat until B = 0										2	4 (If B=0)	16	

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
OUT (n), A	(n) ← A	•	•	X	•	X	•	•	•	11 010 011 ← n →	D3	2	3	11	n to A_0 ~ A_7 Acc. to A_8 ~ A_{15}
OUT (C), r	(C) ← r	•	•	X	•	X	•	•	•	11 101 101 01 r 001	ED	2	3	12	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OUTI	(C) ← (HL) B ← B − 1 HL ← HL + 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 100 011	ED A3	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OTIR	(C) ← (HL) B ← B − 1 HL ← HL + 1 Repeat until B = 0	X	② 1	X	X	X	X	1	X	11 101 101 10 110 011	ED B3	2 2	5 (If B≠0) 4 (If B=0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}
OUTD	(C) ← (HL) B ← B − 1 HL ← HL − 1	X	① ↕	X	X	X	X	1	X	11 101 101 10 101 011	ED AB	2	4	16	C to A_0 ~ A_7 B to A_8 ~ A_{15}

NOTE: ① If the result of B − 1 is zero the Z flag is set, otherwise it is reset.
② Z flag is set upon instruction completion only.

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
OTDR	(C) ← (HL) B ← B − 1 HL ← HL − 1 Repeat until B = 0	X	① 1	X	X	X	X	1	X	11 101 101 10 111 011	ED	2 2	5 (If B≠0) 4 (If B=0)	21 16	C to A_0 ~ A_7 B to A_8 ~ A_{15}

NOTE ① Z flag is set upon instruction completion only.

Summary of Flag Operation

Instruction	D_7 S	Z		H		P/V	N	D_0 C	Comments
ADD A, s; ADC A, s	↕	↕	X	↕	X	V	0	↕	8-bit add or add with carry.
SUB s; SBC A, s; CP s; NEG	↕	↕	X	↕	X	V	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator.
AND s	↕	↕	X	1	X	P	0	0	Logical operations
OR s, XOR s	↕	↕	X	0	X	P	0	0	
INC s	↕	↕	X	↕	X	V	0	•	8-bit increment.
DEC s	↕	↕	X	↕	X	V	1	•	8-bit decrement.
ADD DD, ss	•	•	X	X	X	•	0	↕	16-bit add.
ADC HL, ss	↕	↕	X	X	X	V	0	↕	16-bit add with carry.
SBC HL, ss	↕	↕	X	X	X	V	1	↕	16-bit subtract with carry.
RLA, RLCA, RRA; RRCA	•	•	X	0	X	•	0	↕	Rotate accumulator.

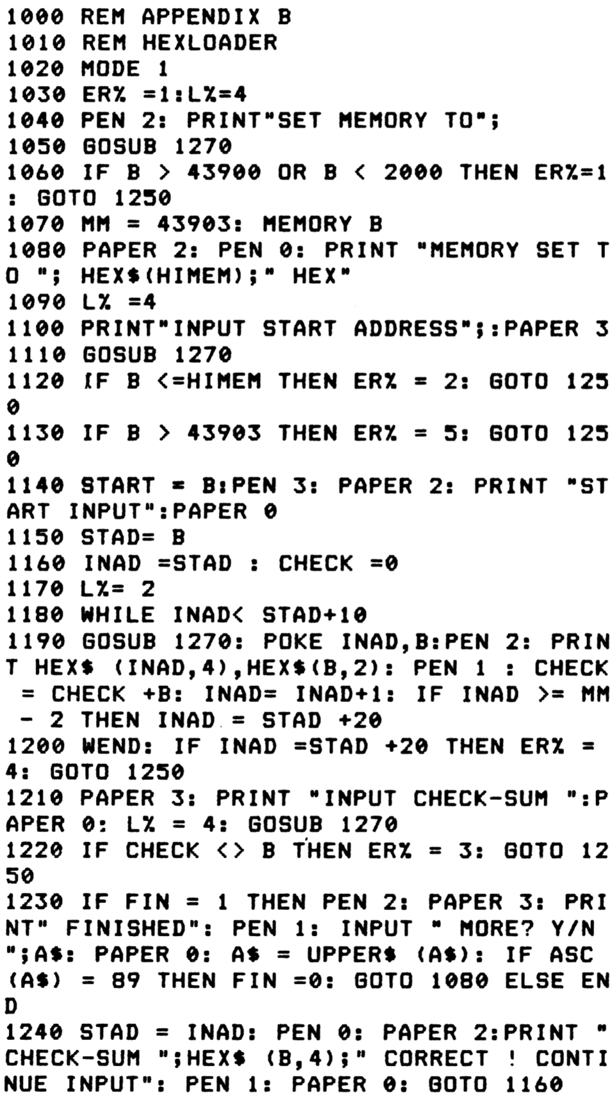
RL m; RLC m; RR m;
RRC m; SLA m;
SRA m; SRL m
	↕	↕	X	0	X	P	0	↕	Rotate and shift locations.

RLD; RRD	↕	↕	X	0	X	P	0	•	Rotate digit left and right.
DAA	↕	↕	X	↕	X	P	•	↕	Decimal adjust accumulator.
CPL	•	•	X	1	X	•	1	•	Complement accumulator.
SCF	•	•	X	0	X	•	0	1	Set carry.
CCF	•	•	X	X	X	•	0	↕	Complement carry.
IN r (C)	↕	↕	X	0	X	P	0	•	Input register indirect.
INI, IND, OUTI; OUTD	X	↕	X	X	X	X	1	•	Block input and output. Z = 0 if B ≠ 0 otherwise Z = 0.
INIR; INDR; OTIR; OTDR	X	1	X	X	X	X	1	•	
LDI; LDD	X	X	X	0	X	↕	0	•	Block transfer instructions. P/V = 1 if BC ≠ 0, otherwise P/V = 0.
LDIR; LDDR	X	X	X	0	X	0	0	•	
CPI; CPIR; CPD; CPDR	X	↕	X	X	X	↕	1	•	Block search instructions. Z = 1 if A = (HL), otherwise Z = 0. P/V = 1 if BC ≠ 0, otherwise P/V = 0.
LD A, I, LD A, R	↕	↕	X	0	X	IFF	0	•	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag.
BIT b, s	X	↕	X	1	X	X	0	•	The state of bit b of location s is copied into the Z flag.

Symbolic Notation

Symbol	Operation
S	Sign flag. S = 1 if the MSB of the result is 1.
Z	Zero flag. Z = 1 if the result of the operation is 0.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V = 1 if the result of the operation is even, P/V = 0 if result is odd. If P/V holds overflow, P/V = 1 if the result of the operation produced an overflow.
H	Half-carry flag. H = 1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N = 1 if the previous operation was a subtract.
H & N	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
C	Carry/Link flag. C = 1 if the operation produced a carry from the MSB of the operand or result.

Symbol	Operation
↕	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii·	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range < 0, 255 >.
nn	16-bit value in range < 0, 65535 >.



```
1000 REM APPENDIX B
1010 REM HEXLOADER
1020 MODE 1
1030 ER% =1:L%=4
1040 PEN 2: PRINT"SET MEMORY TO";
1050 GOSUB 1270
1060 IF B > 43900 OR B < 2000 THEN ER%=1
: GOTO 1250
1070 MM = 43903: MEMORY B
1080 PAPER 2: PEN 0: PRINT "MEMORY SET T
O "; HEX$(HIMEM);" HEX"
1090 L% =4
1100 PRINT"INPUT START ADDRESS";:PAPER 3
1110 GOSUB 1270
1120 IF B <=HIMEM THEN ER% = 2: GOTO 125
0
1130 IF B > 43903 THEN ER% = 5: GOTO 125
0
1140 START = B:PEN 3: PAPER 2: PRINT "ST
ART INPUT":PAPER 0
1150 STAD= B
1160 INAD =STAD : CHECK =0
1170 L%= 2
1180 WHILE INAD< STAD+10
1190 GOSUB 1270: POKE INAD,B:PEN 2: PRIN
T HEX$ (INAD,4),HEX$(B,2): PEN 1 : CHECK
 = CHECK +B: INAD= INAD+1: IF INAD >= MM
 - 2 THEN INAD = STAD +20
1200 WEND: IF INAD =STAD +20 THEN ER% =
4: GOTO 1250
1210 PAPER 3: PRINT "INPUT CHECK-SUM ":P
APER 0: L% = 4: GOSUB 1270
1220 IF CHECK <> B THEN ER% = 3: GOTO 12
50
1230 IF FIN = 1 THEN PEN 2: PAPER 3: PRI
NT" FINISHED": PEN 1: INPUT " MORE? Y/N
";A$: PAPER 0: A$ = UPPER$ (A$): IF ASC
(A$) = 89 THEN FIN =0: GOTO 1080 ELSE EN
D
1240 STAD = INAD: PEN 0: PAPER 2:PRINT "
CHECK-SUM ";HEX$ (B,4);" CORRECT ! CONTI
NUE INPUT": PEN 1: PAPER 0: GOTO 1160
```

```
1250 RESTORE 1390: PEN 3: PAPER 1:: FOR
N% = 1 TO ER%: READ D$:NEXT:PRINT D$;"
TRY AGAIN"; CHR$ (7)
1260 PEN 1: PAPER 0:ON ER% GOTO 1030,109
0,1160,1030,1090
1270 A%= 0:B= 0
1280 PEN 1: INPUT ST$:PRINT CHR$ (11);:S
T$= UPPER$ (ST$):IF ST$= "END" THEN 1370
1290 IF LEN(ST$)<> L% THEN 1360
1300 FOR N%= 1 TO L%
1310 A$=MID$ (ST$,N%,1):IF A$> "F" OR A$
< "0" OR(A$> "9" AND A$< "A") THEN 1360
1320 IF A$> "9" THEN A%= ASC(A$):A%= (A%
 AND &F)+9 ELSE A%= VAL(A$)
1330 IF N%<> L% THEN B= B+ (A%* 16^(L%-N
%)) ELSE B= B+ A%
1340 NEXT
1350 RETURN
1360 PEN 3:PAPER 1: PRINT"INVALID INPUT,
 TRY AGAIN"; CHR$(7): PEN 1:PAPER 0: GOT
O 1270
1370 REM END
1380 FIN= 1: GOTO 1210
1390 DATA UNREALISTICALLY LOW OR HIGH,UN
PROTECTED MEMORY AREA,CHECKSUM DOES NOT
MATCH  YOU WILL HAVE  TO RE-ENTER FROM T
HE LAST CHECK, OUT OF MEMORY, TOO HIGH
```

Appendix C

Hex to Decimal Conversion MSB

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30976	31232	31488	31744	32000	32256	32512
8	32768	33024	33280	33536	33792	34048	34304	34560	34816	35072	35328	35584	35840	36096	36352	36608
9	36864	37120	37376	37632	37888	38144	38400	38656	38912	39168	39424	39680	39936	40192	40448	40704
A	40960	41216	41472	41728	41984	42240	42496	42752	43008	43264	43520	43776	44032	44288	44544	44800
B	45056	45312	45568	45824	46080	46336	46592	46848	47104	47360	47616	47872	48128	48384	48640	48896
C	49152	49408	49664	49920	50176	50432	50688	50944	51200	51456	51712	51968	52224	52480	52736	52992
D	53248	53504	53760	54016	54272	54528	54784	55040	55296	55552	55808	56064	56320	56576	56832	57088
E	57344	57600	57856	58112	58368	58624	58880	59136	59392	59648	59904	60160	60416	60672	60928	61184
F	61440	61696	61952	62208	62464	62720	62976	63232	63488	63744	64000	64256	64512	64768	65024	65280

Appendix D

Hex to Decimal Conversion LSB

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

NIBBLES

HEX	DEC	BIN	HEX	DEC	BIN
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

2s Complement Hex to Decimal Conversion MSB

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	256	512	768	1024	1280	1536	1792	2048	2304	2560	2816	3072	3328	3584	3840
1	4096	4352	4608	4864	5120	5376	5632	5888	6144	6400	6656	6912	7168	7424	7680	7936
2	8192	8448	8704	8960	9216	9472	9728	9984	10240	10496	10752	11008	11264	11520	11776	12032
3	12288	12544	12800	13056	13312	13568	13824	14080	14336	14592	14848	15104	15360	15616	15872	16128
4	16384	16640	16896	17152	17408	17664	17920	18176	18432	18688	18944	19200	19456	19712	19968	20224
5	20480	20736	20992	21248	21504	21760	22016	22272	22528	22784	23040	23296	23552	23808	24064	24320
6	24576	24832	25088	25344	25600	25856	26112	26368	26624	26880	27136	27392	27648	27904	28160	28416
7	28672	28928	29184	29440	29696	29952	30208	30464	30720	30976	31232	31488	31744	32000	32256	32512
8	-32768	-32512	-32256	-32000	-31744	-31488	-31232	-30976	-30720	-30464	-30208	-29952	-29696	-29440	-29184	-28928
9	-28672	-28416	-28160	-27904	-27648	-27392	-27136	-26880	-26624	-26368	-26112	-25856	-25600	-25344	-25088	-24832
A	-24576	-24320	-24064	-23808	-23552	-23296	-23040	-22784	-22528	-22272	-22016	-21760	-21504	-21248	-20992	-20736
B	-20480	-20224	-19968	-19712	-19456	-19200	-18944	-18688	-18432	-18176	-17920	-17664	-17408	-17152	-16896	-16640
C	-16384	-16128	-15872	-15616	-15360	-15104	-14848	-14592	-14336	-14080	-13824	-13568	-13312	-13056	-12800	-12544
D	-12288	-12032	-11776	-11520	-11264	-11008	-10752	-10496	-10240	-9984	-9728	-9472	-9216	-8960	-8704	-8448
E	-8192	-7936	-7680	-7424	-7168	-6912	-6656	-6400	-6144	-5888	-5632	-5376	-5120	-4864	-4608	-4352
F	-4096	-3840	-3584	-3328	-3072	-2816	-2560	-2304	-2048	-1792	-1536	-1280	-1024	-768	-512	-256

Where a 16 bit signed number is negative, the value shown here should be further reduced by the 8 bit unsigned value of the low 8 bits.

2s Complement Hex to Decimal Conversion LSB

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Appendix F

The Amstrad Screen Map

The Screen map on the CPC 464 is not straightforward, by any stretch of the imagination. The start address can change, and a pixel is represented by different bits according to the current mode.

The screen is always devoted to 16K of memory. Unless a program has moved the start of the screen memory it will be at C000h (49152). The other possible start is 4000h (16384) but this must be set by a program. It is unlikely that the screen memory area will be moved so all the following is based on the assumption that the screen memory starts at C000h.

The screen is always made up from 200 one-pixel-high lines, and eighty consecutive bytes from an address which is C000h plus a multiple of 80 always corresponds to a screen line, from the left-hand side to the right. A character is always eight pixels square, so it can be seen that, in mode 2, there is a one-to-one ratio of bits to pixels. A set bit shows that the pixel is ink 1 and a reset bit is ink 0.

Initially (before the screen has been scrolled) the top left of the screen starts at C000h. The first 80 bytes from the top line, the next 80 bytes *do not* form the second line. They are the top line of the next character row, that is the ninth pixel line, the next 80 bytes are the 17th pixel line, and so on for all 25 character lines. Only after dealing with the first pixel lines of all 25 character lines, are the second pixel lines specified.

Initially therefore the screen addresses for the first byte and last byte of pixel lines 1 to 24, will be as shown on opposite page.

There are routines provided in the operating system to allow you to calculate addresses for a character position or for a pixel

	ADDRESS			ADDRESS	
LINE No.	LEFT	RIGHT	LINE No.	LEFT	RIGHT
1	C000	C04F	13	E050	E09F
2	C800	C84F	14	E850	E89F
3	D000	D04F	15	F050	F09F
4	D800	D84F	16	F850	F89F
5	E000	E04F	17	C0A0	C0DF
6	E800	E84F	18	C8A0	C8DF
7	F000	F04F	19	D0A0	D0DF
8	F800	F84F	20	D8A0	D8DF
9	C050	C09F	21	E0A0	E0DF
10	C850	C89F	22	E8A0	E8DF
11	D050	D09F	23	F0A0	F0DF
12	D850	D89F	24	F8A0	F8DF

position, and the CALL addresses for these are given in Appendix G.

The bits within each byte in modes other than 2, do not have a one-to-one relationship with the pixels on the screen, as each byte is required to encode more than two ink colours. The byte order across the screen remains constant in all modes but in mode 1 each byte provides the information for four pixels, and in mode 0 each byte only details two pixels.

Each byte represents pixels from left to right as follows:

Mode 1; from left to right
bits 3&7 2&6 1&5 0&4
Mode 0; from left to right
bits 1,5,3&7 0,4,2&6

The bits are given in order of significance and they encode the ink colour in standard binary.

For example: Mode 1, byte address C000h 00110101b will give the first four pixels (top left of screen) in ink 1, 2, 3 and 4 respectively.

In mode 0 this would have given two pixels, the first in ink 4 and the second in ink 14. To give four pixels in inks 1, 2, 3 and 4 in mode 0 would require two bytes set as follows:

01000000 10011000

When the whole screen is scrolled by the hardware this is achieved by changing the offset of the starting pixel from the start of screen memory by 80, and hence the address of the first (top left) pixel could be C000h + 80 to 80 * 25 mod 2048. Fortunately the firmware provides routines to establish the start address (see Appendix G).

Appendix G

Useful Call Addresses

The Key Manager

Expansion tokens are not expanded unless stated. The keyboard buffer is not cleared unless stated.

HEX CALL		CORRUPTED
ADDRESS	FUNCTION	REGISTERS
BB00	COMPLETELY RESETS THE KEY MANAGER CLEARS BUFFER	AF BC DE HL ENABLES INTS
BB12	GET EXPANSION STRING. ON ENTRY A TO BE EXPANSION TOKEN AND L THE CHARACTER NUMBER. ON EXIT A = CHAR AND CARRY SET, ELSE NO CHAR AVAILABLE	AF DE
BB18	WAITS FOR KEYPRESS, RETURNS CODE IN THE A REGISTER	AF
BB1B	READS KEYBOARD, SETS CARRY FLAG IF KEY PRESSED, AND RETURNS CODE IN A CAN BE USED TO CLEAR BUFFER	AF

BB1E	TESTS IF KEY WHOSE NUMBER IS IN A IS PRESSED. ZERO SET IF NOT PRESSED	AF HL. C (bit 7 if CTRL bit 5 if SHIFT)
BB24	GET JOYSTICK A&H = JOY 0 L = JOY 1 BIT SET IF ACTION TAKEN. 0,UP 1,DOWN 2,LEFT 3,RIGHT 4,FIRE2 5,FIRE1 6,UNALLOCATED 7,ALWAYS RESET	AF HL

THE TEXT VDU

BB4E	FULL INITIALISATION	AF BC DE HL
BB5A	PRINT CHARACTER IN A TO SCREEN	NONE
BB60	READ CHARACTER FROM CURRENT CURSOR POSITION. A CONTAINS CHAR READ IF VALID CARRY SET.	AF
BB75	SET CURSOR TO CHARACTER COLUMN H LINE L.	AF HL

MOST OF THE OTHER POSSIBLE ACTIONS FOR THE TEXT VDU CAN BE ACHIEVED BY "PRINTING" CONTROL CODES, SEE CHAPTER 9 PAGE 2 OF THE AMSTRAD USER INSTRUCTIONS.

GRAPHICS VDU

BB8A	FULL INITIALISATION	AF BC DE HL
BBC0	SET GRAPHIC ORIGIN TO DE (X) HL (Y)	AF BC DE HL

BBDE	SET GRAPHICS PEN. A CONTAINS INK No.	AF
BBEA	PLOT DE (X), HL (Y)	AF BC DE HL
BBF6	DRAW LINE FROM CURRENT ORIGIN TO DE (X), HL (Y) AND UPDATE ORIGIN	AF BC DE HL
BBFC	WRITE CHAR AT GRAPHICS ORIGIN, A = CHARACTER CODE, ORIGIN IS TOP LEFT ORIGIN IS MOVED 8 PIXELS RIGHT.	AF BC DE HL

THE SCREEN PACK

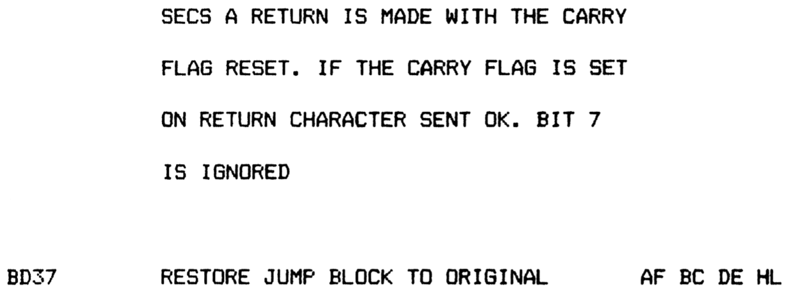
BBFF	FULL INITIALISATION	AF BC DE HL
BC05	SET OFFSET. HL CONTAINS THE OFFSET REQUIRED (EVEN NOs ONLY) OFFSET MOD 80 WILL SCROLL THE SCREEN	AF HL
BC1A	**RETURNS IN HL THE ADDRESS OF THE TOP LEFT OF THE CHARACTER POSITION H (COLUMN) L (LINE). B WILL CONTAIN THE NUMBER OF BYTES FOR A CHARACTERS WIDTH**	**AF**
BC1D	**RETURNS IN HL THE ADDRESS OF PIXEL DE (X) HL (Y), WITH THE MASK IN C AND PIXELS PER BYTE -1 IN B**	**AF DE**

THE NEXT FOUR CALLS ALL REQUIRE THE HL REGISTER PAIR TO CONTAIN THE ADDRESS OF A SCREEN LOCATION, AND WILL RETURN THEIR RESULT

```
IN THE HL PAIR. MOVING OFF THE SCREEN IS NOT PREVENTED AND  MUST

BE CHECKED FOR, AND PREVENTED.


BC20          RIGHT 1 BYTE                           AF


BC23          LEFT 1 BYTE                            AF


BC26          DOWN 1 PIXEL                           AF


BC29          UP 1 PIXEL                             AF


BC38          SET BORDER COLOURS TO B,C              AF BC DE HL


BC3E          SET FLASH PERIODS H,L                  AF HL

THE CASSETTE MANAGER

BC65          FULL INITIALISATION                    AF BC DE HL
```

The cassette manager requires detailed knowledge prior to its use, such as is given in the Firmware Specification Manual, as do the Sound Manager and Kernel entries. It is suggested that any cassette or sound handling is done by returning to BASIC, and then using BASIC to perform the required functions. A CALL can then be made to re-enter the machine code program. Note that you must have CALLed the machine code program from BASIC initially, to be able to return to BASIC. You cannot use the "RUN" command to load and execute your machine code.

```
BD2B          SENDS THE CHARACTER WHOSE CODE IS    AF

              IN THE A REGISTER TO THE CENTRONICS

              (PRINTER) PORT. IF CHARACTER NOT

              SUCCESSFULLY SENT AFTER ABOUT 0.4
```



```
            SECS A RETURN IS MADE WITH THE CARRY

            FLAG RESET. IF THE CARRY FLAG IS SET

            ON RETURN CHARACTER SENT OK. BIT 7

            IS IGNORED


BD37        RESTORE JUMP BLOCK TO ORIGINAL        AF BC DE HL
```

The routines above will allow you to access some of the most useful firmware functions. There are literally hundreds of further routines available, some of which will save time and effort even when using the routines given here. The Firmware Specification Manual gives fuller details of all the firmware routines as well as an overview of the hardware, and detailed examination of the techniques employed by the firmware. If you find that you are serious about machine code programming you have no better recourse than to buy the Firmware Specification Manual. (*SOFT* 158) from Amstrad.

Index

Other titles available from Micro Press:

THE MICRO MAZE: A GUIDE TO PERSONAL COMPUTING
Wynford James
0 7447 0000 0

20 GAMES FOR THE ORIC-1
Wynford James
0 7447 0003 5

QUALITY PROGRAMS FOR THE BBC MICRO
Simon
0 7447 0001 9

QUALITY PROGRAMS FOR THE BBC MICRO (Cassette)
Simon
0 7447 0011 6

15 GRAPHIC GAMES FOR THE SPECTRUM
Richard G. Hurley
0 7447 0002 7

MASTERING THE TI-99
Peter Brooks
0 7447 0008 6

ADVANCING WITH THE ELECTRON
Peter Seal
0 7447 0012 4

QUALITY PROGRAMS FOR THE ELECTRON
Simon
0 7447 0004 3

MAKING THE MOST OF YOUR SPECTRUM MICRO DRIVES
Richard G. Hurley
0 7447 0005 1

GRAPHIC ADVENTURES FOR THE SPECTRUM 48K
Richard G. Hurley
0 7447 0013 2

SPECTRUM SUPERGAMES
Richard G. Hurley
0 7447 0017 5

EDUCATIONAL GAMES FOR THE BBC MICRO
Ian Soutar
0 7447 0016 7

THE ATMOS BOOK OF GAMES
Wynford James
0 7447 0018 3

THE COMMODORE 64 BOOK OF SOUND AND GRAPHICS
Simon
0 7447 0015 9

QL SUPERBASIC: A PROGRAMMER'S GUIDE
John Wilson
0 7447 0020 5

THE QL BOOK OF GAMES
Richard G. Hurley
0 7447 0022 1

THE SPECTRUM OPERATING SYSTEM
Steve Kramer
0 7447 0019 1

BASIC PROGRAMMING ON THE AMSTRAD
Wynford James
0 7447 0024 8

BASIC PROGRAMMING ON THE COMMODORE 64
Gordon Davis and Fin Fahey
0 7447 0026 4

MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

The Amstrad CPC 464 is perhaps the most exciting new computer to appear since the Sinclair Spectrum. It offers many advanced features from BASIC which could previously be accomplished only by vastly more expensive machines.

This book is intended for the beginner wishing to learn how to use Machine Code on the Amstrad CPC 464. It progresses from the concepts of programming in Machine Code, explains the instructions that the Z80 CPU understands and how to use them, and introduces some of the routines in the operating system. Short programs are given to help in entering Machine Code programs, and to inspect and alter or move the contents of part of the memory. Extensive use is also made of the machine operating system allowing results from programs to be seen immediately.

The Author

Steve Kramer has been working with computers for twelve years. He acts as an independent software consultant to a number of software houses in the field of business applications. He is also the author of several games and home-use programs written for popular micros.

GB £ NET +007.95

ISBN 0-7447-0025-6

00795

9 780744 700251

MACHINE CODE FOR BEGINNERS ON THE AMSTRAD

Kramer

MP
MICRO PRESS

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/