Assembly Language Assembled for the Sinclair ZX81

Anthony Woods

Assembly Language Assembled for the Sinclair ZX81

Macmillan Computing Books

Assembly Language Programming for the BBC Microcomputer
Ian Birnbaum

Advanced Programming for the 16K ZX81 Mike Costello

Microprocessors and Microcomputers — their use and

programming Eric Huggins

The Alien, Numbereater, and Other Programs for Personal Computers - with notes on how they were written John Race

Beginning BASIC Peter Gosling

Continuing BASIC Peter Gosling

Program Your Microcomputer in BASIC Peter Gosling

Practical BASIC Programming Peter Gosling

The Sinclair ZX 81 - Programming for Real Applications
Randle Hurley

More Real Applications for the Spectrum and ZX81 Randle Hurley

Assembly Language Assembled - for the Sinclair ZX81 Tony Woods

Digital Techniques Noel Morris

Microprocessor and Microcomputer Technology Noel Morris

Understanding Microprocessors B. S. Walker

Codes for Computers and Microprocessors P. Gosling and
Q. Laarhoven

Z80 Assembly Language Programming for Students Roger Hutty

Assembly Language Assembled for the Sinclair ZX81

Anthony Woods

Lecturer in Computer Science
Peterborough Technical College

First published 1983 by
THE MACMILLAN PRESS LTD
London and Basingstoke
Companies and representatives
throughout the world

ISBN 978-0-333-34588-7 ISBN 978-1-349-06620-9 (eBook)
DOI 10.1007/978-1-349-06620-9

Contents

Preface

This book is for ZX81 users who feel competent at writing programs in BASIC but now want to take their ZX81, and themselves, further.

Assembly language programming takes you into the heart, or should we say brain, of the ZX81 and lets you program it in much finer detail than is possible with BASIC.

The usual reason for writing programs in assembly language is to gain increased processing speed either to give better moving graphics or faster processing of calculations that are repeated often.

Assembly language programming is also used to enable a better program to fit into a small area of memory. All readers will be aware of the limitations of the ZX81 when used with 1K of memory and programmed in BASIC. As you will see later, 1K of memory means that the computer has approximately 1000 places in its memory, which can be used to store data.

If you write a BASIC program, approximately 30 lines of program will fill this memory. If the program is written in assembly language and then translated into machine code, 30 lines of program will only fill about forty or fifty of the places in memory. However, to be truly comparable an assembly language program which is equivalent to the 30-line BASIC program, will probably need to be 100 or 150 lines long. This will still only use something like 200 places in memory, leaving enough memory for a display which fills all of the screen.

Until you have written and run some assembly language programs, it is difficult to appreciate the increase in speed which it will give. As a rough guide an assembly language program will run at least twenty times faster than the equivalent BASIC program.

The main problem with programs in assembly language is that you are working directly with the microprocessor of the computer and you are not protected from it, or it from you, as you are when you write BASIC programs. This means that if you make an error in your program, and everyone does, then instead of stopping and waiting for your correction, the computer is likely to go off on its own and it will try to run anything that is in the memory, as though it was a program. This is a computer 'crash' and the only way of regaining control of the ZX81 is to switch off the machine and start again.

Before an assembly language can be run on a computer it must be translated into the language used by the computer. This is known as machine code language and consists of a series of numbers; all programs that run on the computer, even BASIC programs, are translated into this form before they are run. The translation from assembly language to machine code language can be performed manually or it can be performed by the computer using a program called an assembler. The assembler program used and described in this book is called ZXAS and I consider it to be the best assembler available for the Sinclair ZX81.

Because machine code language does not have any facilities for finding program errors, finding and correcting errors can be a long and difficult task. Many of the problems involved in correcting a machine code program can be eased by using another computer program which enables you to see what is happening inside the computer while your program is running. The ZXDB program has been used in producing this book and it carries out this function admirably.

Remember that assembly language is just another programming language, like BASIC. You will find that it is no more difficult to learn than BASIC, and possibly easier.

After the first two chapters, each chapter consists of a series of short sections. Each section introduces a new idea and usually includes an exercide on the idea. The exercises are meant to be challenging and they encourage readers to make many discoveries for themselves. Some of the exercise answers include notes on the answers, to extend the reader's understanding.

At the end of each chapter the reader is asked to write a program; the program should be coded, run and tested before starting the next chapter. Each program is designed, as far as possible, to include the ideas introduced in the chapter.

All exercises should be done when they are encountered, before continuing with the text. The whole text is designed on the premise that the best way to learn a programming language is by plenty of practical experience. Proficiency in programming is only achieved by programming.

Finally I would like to express my thanks to my wife, Marilyn, for typing the text and her support during the preparation of the book. Thanks are due too to Stephen for acting as a guinea pig and working through the text; by his efforts he has helped to remove the worst of the errors in the text.

Anthony Woods

1 Inside the computer

1.1 MICROCOMPUTERS

Computing, like many other processes, has three main parts:

input - processing - output

In a computer system, data (numbers and words) is input by an input device, the processing is performed by a central processing unit and data is output by an output device.

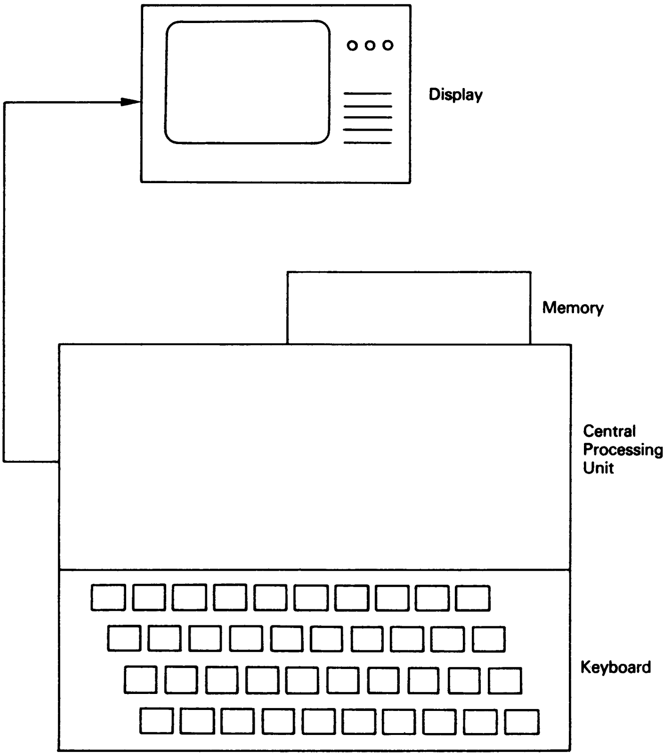


Figure 1.1

A computer can use many types of device but as far as the Sinclair ZX81 is concerned the normal system is shown in figure 1.1.

The keyboard is used to input programs, and data for the programs. A program is a list of instructions to tell the computer what to do during the processing stage. The television set is used to output information, such as results from a program. The memory is used to store programs and data.

If you are not familiar with binary and hexadecimal number systems and signed (two's complement) and unsigned numbers, you should work through Appendix A before continuing with the text.

1.2 THE Z80 CENTRAL PROCESSING UNIT

There are several different microprocessor central processing units available. This book is concerned with the Sinclair ZX81 which uses a Z80 microprocessor.

The components of the Z80 which are of most importance to a programmer are the registers, shown in figure 1.2.

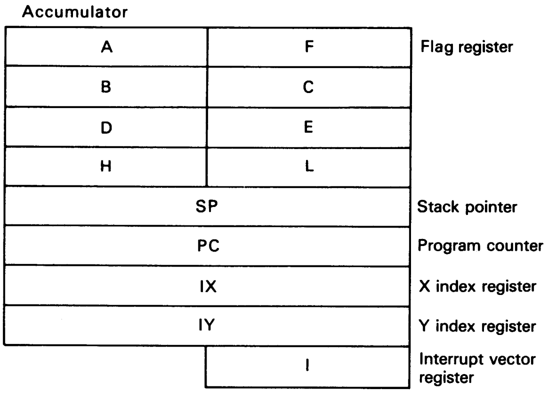


Figure 1.2

A register is a memory area which is built into the central processor. It is used to hold data on a temporary basis while it is being processed, or is waiting to be processed. Each register can hold a single item of data in the form of a binary number. Registers which can hold a binary number with 8 digits are called 8-bit registers and those which can hold 16-digit binary numbers are called 16-bit registers. The term bit is a shorthand way of referring to binary digits.

The accumulator is an 8-bit register which is used for arithmetic and logical operations. For example, to add two numbers in the Z80 microprocessor the first number must be in the accumulator, and the second number is added to the accumulator, leaving the sum in the accumulator.

The flag register is used to hold information about the results of some operations. When arithmetic and logic operations are carried out, the processor automatically tests the result and sets particular bits, or digits, in the flag register to 1 or 0. For example, there are bits in the flag register which indicate whether the result of adding a number to the accumulator is positive, zero or negative.

The B, C, D, E, H and L registers are often referred to as secondary registers and are used mainly to store data temporarily. They can be used as single 8-bit registers or in pairs as 16-bit registers when they are referred to as BC, DE and HL. These secondary registers tend to be used, by convention, in particular ways as you will see throughout the book. In particular, the HL register pair is usually used to point to data in memory.

The 16-bit stack pointer is used to provide a stack facility - this will be explained later.

The 16-bit program counter is used by the central processing unit to keep a trace of the place in memory where the next instruction to be obeyed is located. The program counter normally changes automatically to point to the next instruction, but it can be modified by the programmer to allow the program to jump to an instruction out of the normal sequence. The use of the IX and IY index registers and the interrupt vector register will be explained later.

EXERCISE 1.1

Which register would you expect to be used to subtract one number from another, and which register would you expect to indicate whether the result is positive, zero or negative?

1.3 MEMORY

The memory of a Z80 microprocessor system consists of locations, usually called bytes, which are 8 bits long. Look at figure 1.3.

The number of bytes in a memory varies from one system to another but will normally be 9K or 25K of which 1K or 16K respectively are available to the programmer. 1K is equivalent to 1024 locations or memory bytes.

EXERCISE 1.2

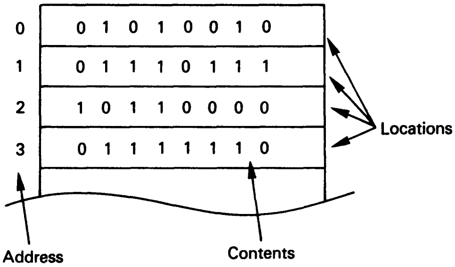
How many bytes are there in a 16K memory expressed as a decimal number and a hexadecimal number?

The bytes of a memory are numbered sequentially starting at zero; the number of a memory byte is referred to as its address. In the ZX81, the bytes at the beginning of memory are used by the ROM and the memory which is available to the programmer starts at address 16509.

Each byte contains an 8-bit binary number which is referred to as the contents of the byte. The 8-bit number may represent any one of several quantities such as an instruction, a number or a character.

EXERCISE 1.3

Referring to figure 1.3, what is the content (in hexadecimal) of the byte whose address is 2?



0	0 1 0 1 0 0 1 0
1	0 1 1 1 0 1 1 1
2	1 0 1 1 0 0 0 0
3	0 1 1 1 1 1 1 0

Figure 1.3

A shorthand form of writing 'the content of a byte whose address is' is to enclose the address in brackets so that, for example, (3) is hexadecimal 7E.

In the remainder of this text binary numbers are denoted by a suffix B and hexadecimal numbers with a prefix $. Numbers with no letter may be assumed to be decimal.

1.4 INSTRUCTIONS

Have a quick glance at Appendix E where you will see the complete Z80 instruction set.

In general, an instruction consists of an operation and an operand. The operation indicates what has to be done and the operand indicates what is to be used in the operation. For example, the binary number 10010010B can represent an instruction. The five digits 10010B represent the operation code (op-code) which specifies that the operand has to be 'subtracted from the accumulator' and the digits 010B are the operand which specifies the 'contents of the register D' are to be used in the subtraction. This means that the binary number 10010010B represents the instruction to subtract the contents of the D register from the accumulator, leaving the result in the accumulator.

An instruction may occupy 1, 2, 3 or 4 bytes depending mainly on how the operand is specified. Look at the first five instructions listed in table E.5 of Appendix E. All five instructions have an op-code of 'add to the accumulator'. Look down the column headed No. of Bytes and you will see that the number of bytes occupied by the instructions varies from one to three. This is because there are differences in the specification of the operand's address. The different ways in which operands may be specified are called 'addressing modes'.

EXERCISE 1.4

What is the instruction 'add register C to the accumulator' in binary? Use table E.5 in Appendix E.

1.5 ASSEMBLY LANGUAGE

In the computer itself instructions are held in binary. Fortunately few computers can be programmed directly in binary since this would be very tedious and error-prone. We could use the POKE statement to put the decimal form of our instruction directly into memory. For the instruction 10010010B we could POKE into memory the number 146. This is an improvement but it is still tedious and error-prone for any reasonably sized program.

A more convenient way of writing programs is to use an assembly language. An assembly language has many facilities to make programming easier.

To start with, mnemonics may be used in place of operation codes. Mnemonics are usually chosen to help a programmer by indicating what the operation is. For example, the instruction referred to in the previous section, 'subtract register D from the accumulator' may be written as:

```
SUB D
```

which is easier to remember than 10010010B, or even $92. Notice also that the operand part of the instruction, in this case register D, may be specified as the letter D in place of the code 010B.

There are many more facilities provided by the ZX81 assembly language; these will be introduced to you throughout the text.

EXERCISE 1.5

What is the assembly language instruction to 'increment register B by one'? Use table E.5 in Appendix E.

Assembly language programs cannot be executed directly by the computer - they have to be converted to their equivalent binary codes. This conversion can be performed by a program such as ZXAS, or it can be converted by hand into decimal or hexadecimal numbers and POKEd into memory, either directly or for hexadecimal numbers using a simple program such as the example in Appendix B. A program which carries out the conversion from assembly language to binary is called an assembler.

2 Assembly language programs

2.1 ASSEMBLING PROGRAMS

The most convenient way of writing and running assembly language programs on the ZX81 computer is to use an assembler program to carry out the translation from assembly language to machine code. A suitable assembler program is ZXAS. If an assembler is not available, programs can be translated manually; this process is known as hand assembly and is described in detail in Appendix B. After the program has been translated into machine code it can be run by the computer and any errors in the program can be found and corrected. This is a more difficult process with assembly language programs than it is with BASIC programs. The process can be made easier by the use of a program which helps you to examine what is happening inside the computer while your program is running. A program of this type is called a debugger and for the ZX81 the program ZXDB performs this invaluable task.

One of the major problems with machine code programs is that pressing the break key has no effect on the program, unless this facility is specially written into the machine code program. The method of doing this is described later in this chapter. If a program goes into an indefinite loop or appears to be doing nothing, the only way of recovering control is to switch off the computer.

On the ZX81 all machine code programs must be run as subroutines of a BASIC program and they must always end with the instruction RET, which returns control back to the BASIC program.

The main part of this chapter describes the use of the ZXAS assembler program and the ZXDB debugger program. Although the details are specific to these two programs, the general ideas presented will apply to any assembler/debugger programs.

2.2 THE ZXAS PROGRAM

ZXAS is a program which will translate a program written in assembly language into machine code so that it can be executed by the computer. It uses 5K of memory, at the top of memory. When it is loaded and run, it automatically resets the ramtop variable so that it is protected from being overwritten. The program recognises all the standard Z80 mnemonics although some details of the format of the instructions have been modified to facilitate entry of the instructions. Table 2.1 gives a full list of all the instructions

TABLE 2.1 LIST OF INSTRUCTIONS RECOGNISED BY ZXAS

ADC A.mm
ADC A.r
ADC A.data
ADC HL.rr
ADC HL.SP
ADD A.mm
ADD A.r
ADD A.data
ADD HL.rr
ADD HL.SP
ADD IX.BC
ADD IX.DE
ADD IX.IX
ADD IX.SP
ADD IY.BC
ADD IY.DE
ADD IY.IY
ADD IY.SP
AND mm
AND r
AND data
BIT n.mm
BIT n.r
CALL mem
CALL c.mem
CCF
CP mm
CP r
CP data
CPD
CPDR
CPI
CPIR
CPL
DAA
DEC mm
DEC r
DEC xx
DI
DJNZ.dis
EI
EX (SP).HL
EX (SP).IX
EX (SP).IY
EX AF.AF'
EX DE.HL
EXX
HALT
IMO
IM1
IM2
IN r.(C)
IN A.port
INC mm
INC r
INC xx
IND
INDR
INI
INIR
JP (HL)
JP (IX)
JP (IY)
JP mem
JP c.mem
JR dis
JR C.dis
JR NC.dis
JR NZ.dis
JR Z.dis
LD (mem).A
LD (mem).xx
LD (rr).A
LD mm.r
LD mm.data
LD A.(mem)
LD A.(rr)
LD r.r
LD r.data
LD r.mm
LD A.I
LD A.R
LD xx.(mem)
LD xx.data
LD I.A
LD R.A
LD SP.HL
LD SP.IX
LD SP.IY
LDD
LDDR
LDI
LDIR
NEG
NOP
OR mm
OR r
OR data
OTDR
OTIR
OUT (C).r
OUT port.A
OUTD
OUTI
POP AF
POP rr
POP IX
POP IY
PUSH AF
PUSH rr
PUSH IX
RL mm
RL r
RLA
RLC mm
RLC r
RLCA
RLD
RST 00
RST 08
RST 10
RST 18
RST 20
RST 28
RST 30
SET n.mm
SET n.r
SLA mm
SLA r
SRA mm
SRA r
SRL mm

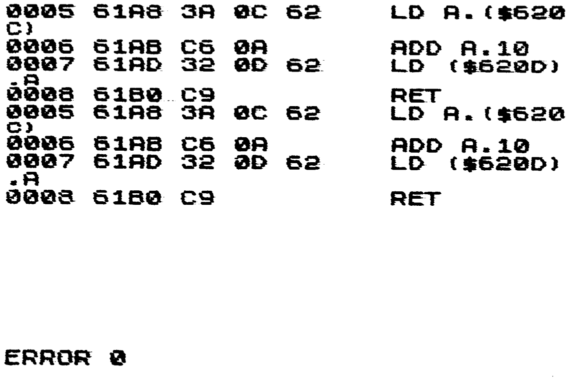
PUSH IY	RR mm	RST 38	SRL r
RES n.mm	RR r	SBC A.mm	SUB mm
RES n.r	RRA	SBC A.r	SUB r
RET	RRC mm	SBC A.data	SUB data
RET c	RRC r	SBC HL.rr	XOR mm
RETI	RRCA	SBC HL.SP	XOR r
RETN	RRD	SCF	XOR data

The format of the instructions is very important; they should be entered exactly as shown here. Extra or missing spaces can cause the instruction to be illegal.

Abbreviations used in the list of instructions:

r	any single 8-bit register, i.e. A, B, C, D, E, H, L
dis	a one byte displacement, in the range -128 to +127
data	immediate data
mm	memory pointing registers, i.e. HL, IX + dis, IY + dis
rr	any register pair, i.e. BC, DE, HL
mem	the address of a location in memory
xx	any 16-bit register, i.e. BC, DE, HL, SP, IX, IY
n	the position of a bit in a memory byte, 0 to 7
c	a tested condition, i.e. C, NC, M, P, Z, NZ, PO, PE

recognised and their format. The main change is the use of full stops in place of commas. All the programs in this book have been written using this assembler. All numbers used in the programs are taken to be decimal unless they are preceded by $ where they are taken to be hexadecimal. As programs are assembled a listing is displayed on the screen, as shown in figure 2.1.



```
0005 61A8 3A 0C 62     LD A.($620
C)
0006 61AB C6 0A        ADD A.10
0007 61AD 32 0D 62     LD ($620D)
.A
0008 61B0 C9           RET
0005 61A8 3A 0C 62     LD A.($620
C)
0006 61AB C6 0A        ADD A.10
0007 61AD 32 0D 62     LD ($620D)
.A
0008 61B0 C9           RET




ERROR 0
```

Figure 2.1

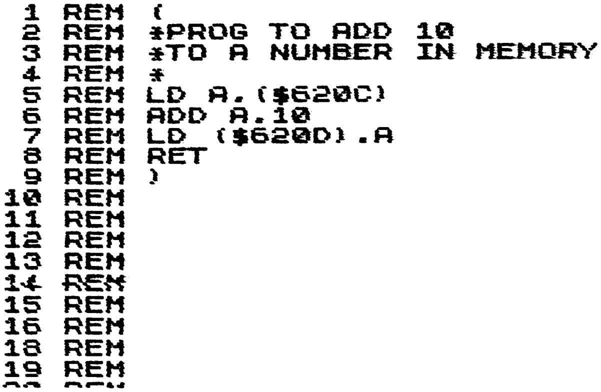
The first column is the line number of the instruction, the second column is the address of the memory byte which holds the first byte of the instruction, the third column gives the machine code for the instruction, in hexadecimal, and the final column is the assembly language instruction.

When the screen is full you have to press any key for assembly to continue. Pressing the break key at any time during assembly stops the program.

2.3 USING ZXAS

Load the program from cassette tape, using the file name ZXAS. Run the program; this transfers the assembler program to the top of the memory and sets the RAMTOP variable so that the program cannot be overwritten with a BASIC program or deleted by the NEW command. Lines 10, 20 and 30, which are used to hold the assembler as a machine code program, can now be deleted and the assembler is ready for use.

The assembly language instructions are entered into the BASIC program inside REM statements. Instructions can be entered with line numbers up to 8999. Although all the example programs in this book show only one instruction to a line, this is for clarity only. More than one instruction per line is allowed if the instructions are separated by semicolons. The entire assembly language program must be enclosed in brackets which are themselves in REM statements, as shown in figure 2.2.



```
 1 REM (
 2 REM *PROG TO ADD 10
 3 REM *TO A NUMBER IN MEMORY
 4 REM *
 5 REM LD A,($620C)
 6 REM ADD A,10
 7 REM LD ($620D),A
 8 REM RET
 9 REM )
10 REM
11 REM
12 REM
13 REM
14 REM
15 REM
16 REM
18 REM
19 REM
```

Figure 2.2

To translate into machine code enter the command GOTO 9000. The computer will then wait for a memory address to be entered before translating the instruction; this will be the first byte of the machine code program. The address should be entered as a decimal number. (Section 2.7 discusses suitable areas of memory for the machine code program.) When the screen is full, assembly stops until any key is pressed; if the break key is pressed at any time during assembly, the program stops. When the assembly is completed, an error message is displayed at the bottom of the screen. The meanings of these messages are given in table 2.2.

TABLE 2.2 ASSEMBLY ERROR MESSAGES

Message	*Meaning*
ERROR 0	No mistakes, assembly is error free.
ERROR 1	No (found, no opening bracket in a REM statement.
ERROR 2	No) found, no closing bracket in a REM statement.
ERROR 3	Illegal label, jump to non-existent label.
ERROR 4	Illegal instruction, next instruction is not understood. Check with table 2.1.
ERROR 5	Number out of range.
ERROR 6	Relative jump out of range.

When assembly is finished one of the above error messages will be displayed at the bottom of the screen.

Not all the instructions can be completely translated immediately; for example, a jump can refer to a label which is further on in the program, and the assembler processes every instruction twice.

The machine code program can be run by a USR call to the address of the first byte of the program. This is carried out by using a BASIC statement such as:

```
RAND USR 16514   or
GOTO USR 16514
```

Either of these statements will start the machine code program, beginning at memory byte 16514.

2.4 THE ZXDB PROGRAM

ZXDB is a program which is used mainly to help find any errors in a program. It is used in two major ways: to look at or change what is in the computer's memory and to run programs under the control of the programmer. When the program is loaded it occupies the first 4K of memory. The program is then ready to accept a command; each command has a single letter assigned to it and the letter, when entered, calls the appropriate command. If more data is required by the command, the program will display a further prompt or prompts. All inputs to the program must be made in hexadecimal, and if more than four characters are input only the last four are used. Entering EDIT during any input will return you to the command prompt. The best way to learn how to use ZXDB is to experiment with all the commands.

2.5 ZXDB COMMANDS

When ZXDB is run, it displays an asterisk prompt, *, to show that it is ready for a command. This section describes the commands available in ZXDB, with the exception of the single stepping command which is described separately. Some of the commands require further inputs after the single command letter has been entered. These are indicated on the computer by either = or > being displayed as a prompt. This section shows the command letter, a brief description of the command and then any further prompts which will be displayed, followed by the input required by the prompt.

V Display memory in hexadecimal

= Input the starting address

This command displays on the screen the contents of the eight consecutive memory bytes, starting from the address that has been input. The contents are given as eight hexadecimal numbers. Entering NEWLINE will display the contents of the next eight bytes of memory. Entering Q will return to ZXDB for the next command.

A Display memory as characters

= Input the starting address

This command displays on the screen the contents of the 32 consecutive memory bytes starting from the address that has been input. The contents will be displayed as characters, using the Sinclair character codes (Appendix D), if possible. If the contents of a memory byte do not give a valid character code a space will be displayed. Entering NEWLINE will display the contents of the next 32 memory bytes. Entering Q will return to the = prompt for input of a new starting address. Input of Q in response to the = prompt returns to ZXDB for the next command.

G Execute a program

= Input starting address of program

This command is very useful when developing and testing a program. It means that a machine code program can be run independently of any accompanying BASIC program. The program should end with JP 4100 to return to ZXDB or JP 410 to return to BASIC.

Q Leave ZXDB and return to BASIC

F Fill a block of memory

> Input value to be placed in memory

= Input starting address of block of memory

= Input end of block of memory

This command enables the user to fill all of the bytes in a block of memory with the same value.

E Hexadecimal loader

= Input the starting address

This command allows values or machine code programs to be entered into memory bytes. The values should be hexadecimal numbers. Only inputs consisting of two-digit hexadecimal numbers followed by NEWLINE will change the value in memory. Inputting a Q will return to ZXDB command mode. Any other input will leave the value in the displayed memory byte unchanged and display the address and value of the next memory byte.

M Move a block of memory

= Input the start of block to be moved

= Input the end of the block to be moved

= Input the starting address for the new block

This command will move a copy of the values in the original block of memory bytes to the new block of memory bytes. If the starting address of the new block is less than the starting address of the original block, then end the input of the starting address of the new block with I. If the starting address of the new block is greater than the starting address of the original block, then end input of the starting address of the new block with a NEWLINE.

C Compare two blocks of memory

= Input the start of the first block of memory

= Input the end of the first block of memory

= Input the start of the second block of memory

This command compares the contents of two blocks of memory and displays the differences between the two blocks. Whenever a difference is found the program displays the addresses and values in the two memory bytes and then waits for an input. Entering NEWLINE will continue the comparison. Entering EDIT will return to ZXDB for the next command.

S Search memory

After entering S, ZXDB waits for the user to input the string that it is to search for; there is no prompt for the input of the string. The string must be input as a series of hexadecimal numbers separated by full stops and ending with NEWLINE; the character before NEWLINE must be a full stop. The characters in the string must be converted into their code numbers using the table of character codes in Appendix D. After the string has been input, the program prompts with "=" for the address of the first memory byte to be examined. Each time it finds the string in memory it will display the address of the first character in the string and then pause. At each pause enter NEWLINE to continue the search, or enter Q to return to ZXDB command mode. If when the string of hexadecimal numbers is input, a four-digit hexadecimal number is input instead of a two-digit number, then the first two digits are taken as a mask for the second two digits. The effect of this is that if the bit in the binary equivalent of the mask is 1, then the corresponding bit of the input does not need to match the memory byte for a 'match' to be displayed. For example, typing CD.FF00.IF00. will cause a search to be made for the string of binary numbers 11001101.bbbbbbbb.000bbbbb where b is either binary value, 0 or 1. When a match has been found and the program pauses, entering C allows the string in memory to be changed by entering a new string in the same way as the original search string was entered. The new string may be longer, shorter or the same length as the original string.

W Set a display window

= Input number of lines to be used

This command sets the number of lines, counting from the bottom of the screen, used by ZXDB for its display. When a program is

TABLE 2.3 DISASSEMBLER EXCEPTIONS

Disassembler	*Standard*
LD B,IX + dis	LD B,(IX + dis)
LDA mem	LD A,(mem)
STA mem	LD (mem),A
LDAX rr	LD A,(rr)
STAX rr	LD (rr),A
LHLD mem	LD HL,(mem)
SHLD mem	LD (mem),HL
SPHL	LD SP,HL
XCHG	EX DE,HL
EX	EX AF,AF'
PCHL	JP (HL)
XTHL	EX (SP),HL
IN port	IN A,(port)
OUT port	OUT (port),A
COUT r	OUT (C),r
CIN r	IN r,(C)

In addition (HL) is always referred to as M.

The dissassembler option uses the standard Z80 mnemonics (instructions) with the above exceptions.

being single stepped (see next section) this command allows the program to use the screen in addition to the ZXDB display.

D Disassemble memory contents

= Input the start address

The effect of this command is to interpret the contents of memory as instructions and to display the instructions using the standard Z80 mnemonics with the exceptions shown in table 2.3. After an instruction has been displayed the computer waits for an input. Entering NEWLINE will display the next instruction in memory. Entering a new address in hexadecimal, followed by NEWLINE, will cause the instruction at the new address to be displayed. To end the disassembly enter Q, which will cause a return to ZXDB command mode.

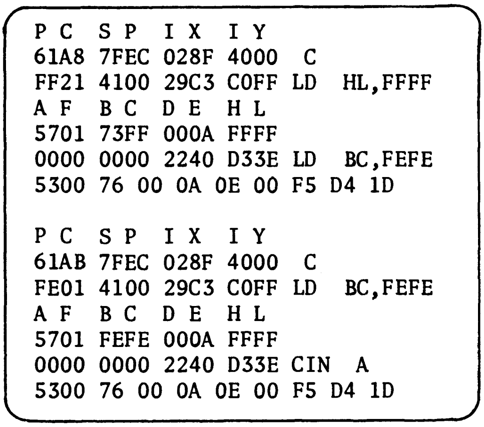
2.6 SINGLE STEPPING

The most important command in the ZXDB program is the B command. This command allows a program in memory to be run until it reaches a breakpoint specified by the user. At the breakpoint the program stops and the contents of the registers, the state of any flags, the instruction which has just run, the next instruction to be run and the values in eight consecutive memory bytes are all displayed on the screen. Figure 2.3 shows the format of the display, and figure 2.4 shows a typical display.

```
  PC       SP       IX       IY
 XXXX     XXXX     XXXX     XXXX     Flags set
(XXXX)   (XXXX)   (XXXX)   (XXXX)    Present Instruction
  AF       BC       DE       HL
 XXXX     XXXX     XXXX     XXXX
(XXXX)   (XXXX)   (XXXX)   (XXXX)    Next Instruction
Address  XX    XX    XX    XX    XX    XX    XX    XX
```

The display shows the value in each register pair and in the pair of memory bytes pointed to by each register pair. It also shows the contents of the eight memory bytes starting from address in the display; this address is set by the window command.

Figure 2.3



```
P C  S P  I X  I Y
61A8 7FEC 028F 4000  C
FF21 4100 29C3 C0FF LD  HL,FFFF
A F  B C  D E  H L
5701 73FF 000A FFFF
0000 0000 2240 D33E LD  BC,FEFE
5300 76 00 0A 0E 00 F5 D4 1D

P C  S P  I X  I Y
61AB 7FEC 028F 4000  C
FE01 4100 29C3 C0FF LD  BC,FEFE
A F  B C  D E  H L
5701 FEFE 000A FFFF
0000 0000 2240 D33E CIN  A
5300 76 00 0A 0E 00 F5 D4 1D
```

Figure 2.4

The format of the breakpoint command is as follows:

B Set a breakpoint

= Input the address of the instruction on which the program is to stop

= Input the starting address of the program to be tested

When the program has stopped at the breakpoint, continuation of the program can be in single step mode or according to one of the single step commands. Single stepping means running the program one instruction at a time with an updated register display, as shown in figures 2.3 and 2.4, after each instruction. Running a program in single step mode is a useful way for a programmer to determine exactly what the program is doing.

The following is a list of the commands which can be used when the program is in single step mode, that is, after a breakpoint has been reached or when the program is already running in single steps. These commands are completely separate from the main ZXDB commands described in the previous section.

NEWLINE Execute the next instruction (single step)

G Run the program normally

nW Display the eight memory bytes starting at n

R Can only be used at a CALL instruction, executes the subroutine and returns to single step mode after the subroutine

nP Sets a breakpoint at address n

Q Return to the command (asterisk) prompt

L Sets a breakpoint at the current instruction and then runs the program normally until the current instruction is reached again. This is very useful when finding errors in a loop. This command can only be used if the current instruction is three bytes long.

O Only the instructions are displayed

nN Runs the next n instructions in single step mode

Z Contents of any of the 8-bit registers can be changed by entering the name of the register followed by the required contents, for example, A1B puts the number 1B in the A register. The register changed can be any of A, B, C, D, E, F, H, L or M which is the memory byte pointed to by the HL register. Enter NEWLINE to return to single step.

nT Runs the program normally until one of the 16-bit registers contains the number n or a call is made to memory byte number n; the program then returns to single step mode.

2.7 PROGRAM STORAGE

A problem for the assembly language or machine code programmer is deciding where in the computers memory to store a program so that it can be run.

There are essentially three different areas of memory which can be used for storing machine code programs. The safest area is at the top of memory. The assembler program ZXAS uses memory from location 27648 to location 32767, the area immediately below this can be used for your machine code programs. By resetting the system variable RAMTOP, to the memory byte just below the start of your program, the program is protected from overwriting.

The second area is a spare area of memory in the middle of memory. When ZXDB is being used there is a free area of memory from $50C0 to $52FF (20672 to 21247). From BASIC the spare memory is pointed to by the system variable STKEND.

The main disadvantage of the areas discussed so far is that the machine code program cannot be SAVEd. Memory is only saved as far as the memory location contained in the system variable E-LINE. Saving the program will, of course, save the assembly listing which is in the BASIC program in REM statements; but you should remember that to run the program you will need to have the assembler program in your ZX81.

These areas of memory are used mainly for storing programs while they are being tested. When the program is error free it can then be transferred to an area of memory within a BASIC program so that it can be saved as a machine code program.

If the machine code program is to be saved it must be stored within the BASIC program area and this is most conveniently done by storing the program in a REM statement at the beginning of the BASIC program. The detailed procedure is to make the first line of the BASIC program a REM statement, which should contain as many characters as there are bytes in the machine code program. Usually it is easier to make the REM statement longer than will be needed. The assembly language program should then be assembled, starting at location 16514 which is the location of the first character after REM. The program may now be saved with the machine code inside the program.

3 Some simple instructions

3.1 THE BASIC OPERATIONS

In this chapter we shall look at the operations of loading registers, adding to the value in the accumulator, subtracting from the value in the accumulator, incrementing and decrementing the value in a register, and taking the negative of the value in the accumulator. These instructions allow us to carry out simple arithmetic with 8-bit numbers.

3.2 LOADING A REGISTER

Any of the single 8-bit registers can be loaded directly with a numeric value by using an instruction with the form

```
LD r.n
```

where r can be any of the registers A, B, C, D, E, H or L, and n is a positive whole number in the range 0 to 255. For example the instruction LD B.99 will put the value 99 into the B register. The computer can be programmed so that it recognises positive and negative values, in which case the range of values which can be loaded into a register is -128 to +127.

EXERCISE 3.1

Why must the value in a register be within the quoted ranges?

3.3 SIMPLE ADDITION AND SUBTRACTION

Numbers can be added to, and subtracted from, the value in the accumulator using the instructions

```
ADD A.n   and   SUB n
```

Although the accumulator is the only register which can be used for 8-bit arithmetic, 16-bit addition can be carried out using one of the 16-bit registers. This means that in the ADD instruction the use of the accumulator must be specified, whereas subtraction can only be from the accumulator and so it does not have to be specified in the SUB instruction.

The following sequence of instructions shows the use of these instructions:

```
LD A.15
ADD A.46
SUB 22
```

The effect of these instructions is to put the value 15 into the accumulator, add 46 to this - giving the value 61 in the accumulator, subtracting 22 from this - leaving the value 39 in the accumulator. This is equivalent to calculating 15 + 46 - 22.

EXERCISE 3.2

Write a program segment to calculate 73 - 21 + 55.

The value in any of the single 8-bit registers, including the accumulator, can be added to or subtracted from the value in the accumulator using the instructions

```
ADD A.r   and   SUB r
```

This allows values which have been stored temporarily in one of the registers to be added to, or subtracted from, the value in the accumulator.

3.4 MOVING BETWEEN REGISTERS

The value in a register may be loaded into another register with the instruction

```
LD r1.r2
```

which loads register rl with the value in register r2. The value in register r2 is not affected by this instruction which puts a copy of the value in r2 into register rl. This instruction is used mainly for saving the contents of the accumulator temporarily while the accumulator is used for other purposes.

EXERCISE 3.3

Write a program segment to compute 3 × (56 - 22). Do this by adding (56 - 22) to itself twice.

3.5 INCREMENT AND DECREMENT

The instructions

```
INC r   and   DEC r
```

are used to increase, or decrease, the value in a register by one. The main use for these instructions is to enable a count to be maintained of the number of times a particular group of instructions are carried out, as we shall explain later. However, they can also be used to add one to, or subtract one from, the value in a register during an arithmetic calculation. In particular INC A or DEC A execute faster and use less memory than ADD A.1 or SUB 1.

3.6 FINDING A NEGATIVE

The instruction

```
NEG
```

gives the negative of the value in the accumulator. For example, if the accumulator contains the value 78 and a NEG instruction is executed the accumulator will then contain -78.

It is essential that the value in the accumulator is treated as a number in the range -128 to +127. A NEG instruction must not be executed with a value which is defined as a number in the range 0 to 255 since the effects are difficult to define.

EXERCISE 3.4

What is the value in the accumulator after each of the instructions in the following program segment?

```
LD A.27
NEG
INC A
```

3.7 ADDRESSING MODES - IMMEDIATE AND EXTENDED

The way in which an instruction refers to the value to be used by the instruction, is called the addressing mode. Several of the instructions that we have used so far have the value to be used specified in the instruction, and this is known as immediate addressing. Some examples of immediate addressing are:

LD B.35, ADD A.15 and SUB 20.

Another addressing mode - extended addressing - is used when the value in a particular memory byte is required. For example:

```
LD A.($527A)
```

specifies that the value in memory byte $527A is to be loaded into the accumulator. Similarly, the value in the accumulator can be stored in a memory byte using the instruction

```
LD (n).A
```

where n is the address of the memory byte.

EXERCISE 3.5

If memory byte $35 contains 79, what will be the value in the accumulator after the instructions

(i) LD A.$35   and   (ii) LD A.($35)

have been executed?

Of the single 8-bit registers, only the accumulator may be loaded and stored using the extended addressing mode.

3.8 AN EXAMPLE PROGRAM

The example program is shown in figure 3.1. It is a small program which adds 10 to the value in a memory byte and stores the result in another memory byte.

Figure 3.1 shows the listing written in REM statements and the listing produced by the ZXAS assembler after the program has been assembled. The first column of the assembler listing is the line number of the BASIC program which contains the assembly language program, the second column is the address in hexadecimal of the first memory byte used to store the instruction, the third column is the machine code program in hexadecimal, and the fourth column is the assembly language instruction. This listing shows that some instructions use one memory byte, some use two memory bytes and some use three memory bytes to hold a single instruction.

All machine code programs on the Sinclair must be run from within a BASIC program and they should always end with a RET instruction as this returns control to the BASIC program.

This program can be tested after assembly into the locations shown either by running the ZXDB program or by running the following BASIC program. The program has been assembled in memory starting at location 25000 ($61A8).

```
10      INPUT A
20      POKE 25100,A
30      RAND USR 25000
40      PRINT PEEK 25101
```

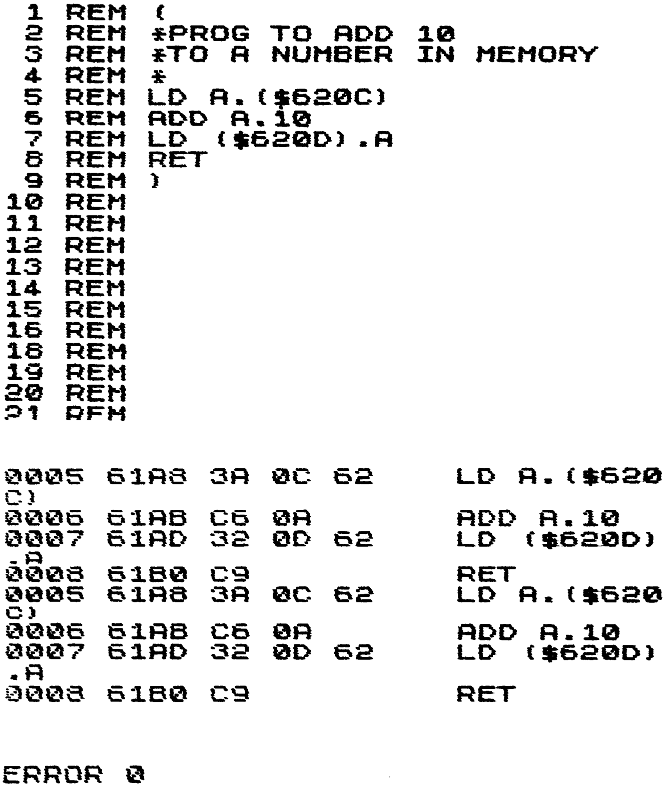
Line 10 inputs the number to which we wish to add 10; it should be an integer in the range 0 to 245.

Line 20 puts the input number into memory byte number 25100.

Line 30 transfers control to the program starting in memory byte number 25000, which is equivalent to $61A8. This is the line which tells the computer to run our machine code program.

Line 40 prints out the value in memory byte number 25101, which is where the machine code program stores its result.

Because assembly language programs are more difficult to follow and understand that programs in BASIC, it is a good idea to include plenty of comments in a program when it is being written. Comments included in the assembly language program can start anywhere on a line, following an asterisk.



```
 1 REM (
 2 REM *PROG TO ADD 10
 3 REM *TO A NUMBER IN MEMORY
 4 REM *
 5 REM LD A.($620C)
 6 REM ADD A.10
 7 REM LD ($620D).A
 8 REM RET
 9 REM )
10 REM
11 REM
12 REM
13 REM
14 REM
15 REM
16 REM
18 REM
19 REM
20 REM
21 REM

0005 61A8 3A 0C 62      LD A.($620
C)
0006 61AB C6 0A         ADD A.10
0007 61AD 32 0D 62      LD ($620D)
.A
0008 61B0 C9            RET
0005 61A8 3A 0C 62      LD A.($620
C)
0006 61AB C6 0A         ADD A.10
0007 61AD 32 0D 62      LD ($620D)
.A
0008 61B0 C9            RET


ERROR 0
```

Figure 3.1

3.9 PROGRAM

Write a program which calculates

$$C = A - 3 (B + 1) - 1$$

Assemble the program either by using an assembler program such as ZXAS or by hand assembly. Hand assembly is described in Appendix B.

Load the program into a suitable area of memory and test it in a similar way to the program described in the previous section. If you are running it from a BASIC program you will need to POKE values for A and B into memory bytes before the machine code program is run. Readers using the ZXDB program will find it instructive to single step through the program to see what happens during the execution of the program.

4 Subroutines and output to the screen

4.1 SUBROUTINES

Subroutines are a very important feature of programming, especially assembly language programming, which are usually left until near the end of any course on programming. Because of their importance, this chapter will show why they are used and how to use them. It will not look at the details of how they work; this will be covered in chapter 8.

First we take a look at the reasons for using subroutines. Suppose a program contains two or more groups of statements which carry out the same function, as indicated by the shaded areas on the left of figure 4.1. It is, obviously, wasteful to keep repeating the same group of statements within the program, so instead we write the group of statements once, as a separate block on their own. We call this block of statements a subroutine, as shown on the right of figure 4.1. The program now consists of a main program, followed by the subroutine. In the program run, when the group of statements in the subroutine are required to be run, a special jump instruction is used to jump to the start of the subroutine. After the subroutine has been run, another jump instruction is carried out which returns control to the main program and carries on running the main program from the instruc-

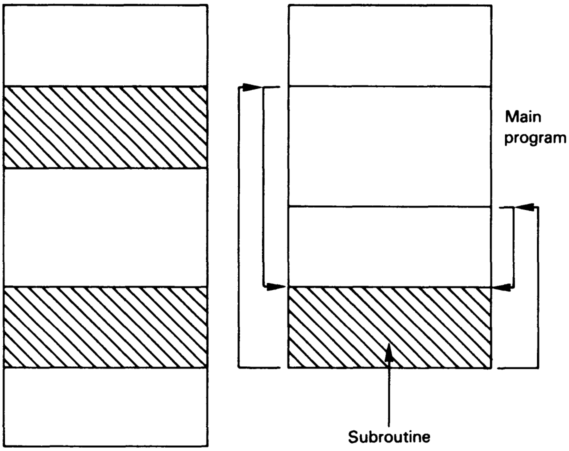


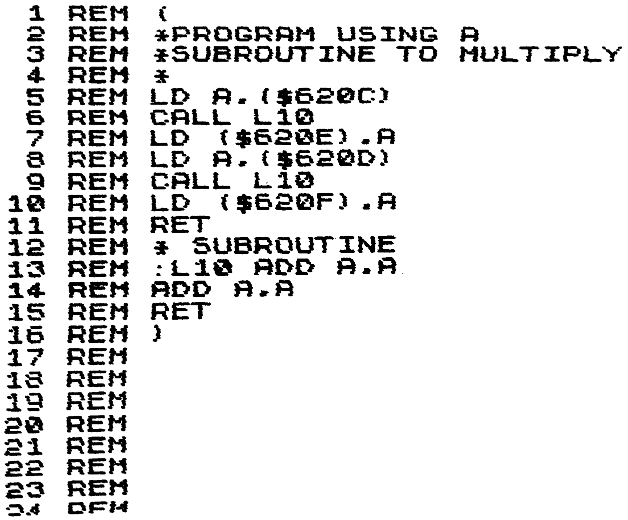
Figure 4.1

tion after the instruction which called for the jump to the subroutine.

It is important to notice that jumps from the main program to the subroutine can be made from anywhere in the main program and they will always jump to the start of the subroutine. When the subroutine has been run, the jump back to the main program will always be to the instruction after the instruction which called for the jump to the subroutine. This means that the subroutine can jump back to several different places in the main program depending upon where the jumps to the subroutine occurred. In chapter 8 we will look at the method used by the subroutine to find the correct instruction for the jump back to the main program. When a program makes a jump to a subroutine we say that it is calling the subroutine.

4.2 THE CALL AND RET INSTRUCTIONS

Subroutines are called in the Z80 microprocessor by using the instruction CALL. CALL must be followed by an indicator to the first instruction of the subroutine. In a machine code program this will be the address of the memory byte which contains the first instruction of the subroutine. In an assembly language program this address will be calculated by the assembler program, but in order to carry out this calculation it must be able to identify the first instruction in the subroutine. This is done by giving the instruction a label, similar to giving a house a name. The assembler can then refer to the memory byte holding the instruction by either its label or its numeric address. Figure 4.2 shows a program which calls a subroutine twice from the main program. The subroutine, which is called by its label L10, multiplies the number in the accumulator by four.



```
 1 REM (
 2 REM *PROGRAM USING A
 3 REM *SUBROUTINE TO MULTIPLY
 4 REM *
 5 REM LD A.($620C)
 6 REM CALL L10
 7 REM LD ($620E).A
 8 REM LD A.($620D)
 9 REM CALL L10
10 REM LD ($620F).A
11 REM RET
12 REM * SUBROUTINE
13 REM :L10 ADD A.A
14 REM ADD A.A
15 REM RET
16 REM )
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
```

Figure 4.2

Going through the program will show how it runs. First the accumulator is loaded with the value in memory byte $620C and then the subroutine starting at L10 is run. The subroutine first adds the value in the accumulator to itself; this is the same as multiplying the value by two. This instruction is then repeated, which multiplies the new value by two, so that the original value is multiplied by four. The last instruction in the subroutine is RET which returns control to the main program and it then loads the value in the accumulator into memory byte $620E.

The main program then continues with the next instruction which loads the value in memory byte $620D into the accumulator. The next instruction calls the subroutine for a second time to multiply the new value in the accumulator by four. At the end of the subroutine control is again returned to the main program which loads the new value in the accumulator into memory byte $620F. The final instruction in the main program is also a RET which is a return from a subroutine. This is used because the USR function from BASIC is also a jump to a subroutine, in this case your machine code program. The whole of your assembly language program must be written as a subroutine to a BASIC program, which could be similar to the program shown in chapter 3.

The accumulator is used by the subroutine just examined to take data from the main program to the subroutine and to return the result from the subroutine back to the main program. Any of the registers, memory bytes or the stack may be used to pass data between programs and subroutines, and to pass data back to a program at the end of a subroutine.

EXERCISE 4.1

Write a subroutine to add together the contents of registers B, C, D and E and leave the result in the accumulator.

A program may need to use more than one subroutine. All that is required is that the start of each subroutine is indicated by a different label and each subroutine ends with a RET instruction. When more than one subroutine is used it is usually easier to understand the program if all the subroutines are located at the end of the main program, one after the other.

4.3 ANOTHER SUBROUTINE INSTRUCTION

In addition to the CALL instruction, the Z80 microprocessor has an instruction which allows subroutines stored in the first few bytes of the memory to be called with a single byte instruction. The instruction to call these subroutines is RST; this is followed by one of eight allowed starting addresses. In the ZX81 all of these subroutines are used by the monitor ROM and only a few of them are useful to the assembly language programmer. For example, the instruction RST 18 calls a subroutine which is used by BASIC to collect the next character in a program line.

4.4 SOME USEFUL SUBROUTINES

One big advantage of subroutines is that a programmer can use someone else's subroutines without necessarily knowing the details of how the subroutine works. A fruitful source of subroutines for the ZX81 assembly language programmer is the Sinclair ROM.

Usually outputting to a display in an assembly language program needs a complex set of instructions, but by using a subroutine from the Sinclair ROM we can output a character by loading its character code into the accumulator and then calling a subroutine using the instruction RST 10. Incidentally, the character code is still in the accumulator after the return from the subroutine. The character codes are given in Appendix D.

EXERCISE 4.2

What is the character code for the letter G, and which character has the code of $15?

A major problem with assembly language programs is that under normal circumstances the break key has no effect, so that the only way of stopping a program which is not working properly is to turn off the computer. However, when running BASIC programs the ZX81 calls a subroutine to check for the break key and this subroutine can be used at strategic places in your program; for example, in any program loop, this can be used to enable the program to be stopped without having to turn off the computer. This subroutine is called by the instruction CALL $F43. Table 4.1 gives a list of a few of the useful subroutines in the ZX81 monitor.

TABLE 4.1 SUBROUTINES IN THE MONITOR

Calling instruction	*Description of the routine*
RST 0	This is equivalent to switching off and on again.
RST 10	Display the character in the accumulator in the next screen position.
CALL 02BB	Keyboard scan, check for key press.
CALL 07BD	Keyboard decode routine.
CALL 02F6	This is the SAVE routine.
CALL 0340	This is the LOAD routine.
CALL 0A2A	Clear screen.
CALL 0F20	FAST mode.
CALL 0F28	SLOW mode.
CALL 0CBA	Start BASIC.
CALL 0F43	Check for BREAK key.

4.5 LABELS

Labels are chosen by the programmer and for the ZXAS assembler; they consist of the letter L followed by a number between 0 and 255. When used at the beginning of an instruction the label must be preceded by a colon (:). The main use of labels is to identify instructions for jumps and subroutines, but they can be used for other purposes. An interesting use of labels is to set up data storage areas in memory.

One of the instructions available in the ZX81 is a do nothing instruction; this has the mnemonic NOP which stands for No Operation. The main use of the NOP instruction is to provide a delay in a program, by including it in a loop. Since the instruction occupies one byte in memory, one or more NOP instructions identified by a label can be used to save data. For example, the following program segment loads the value from HL into the two memory bytes pointed to by the label L30 and occupied by the two NOP instructions

```
LD (L30).HL

-------

-------

:L30NOP;NOP

--------
```

4.6 PROGRAM

Using the subroutine in the ROM write a program to output your initials followed by NEWLINE.

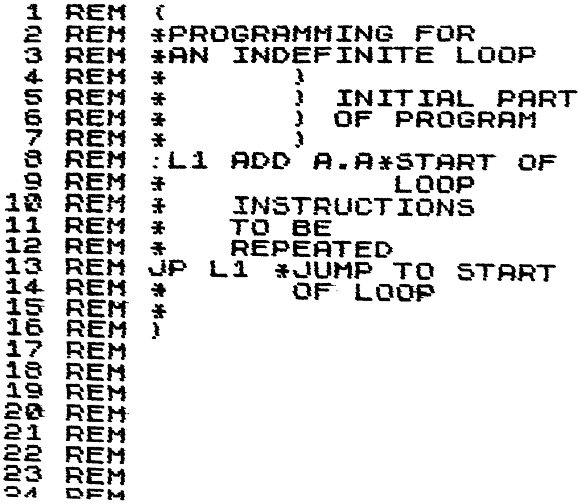
5 Unconditional jumps and keyboard input

5.1 UNCONDITIONAL JUMPS

You have so far only seen programs which carry out their instructions in the sequence in which they are written. However, as you know from your BASIC programs, it is usual to include in programs some instructions which allow the sequence to be changed. This is done by jumping from the present instruction in the program to some other instruction. There are two types of jump instructions - unconditional and conditional. This chapter will only look at unconditional jumps. An unconditional jump has the form

```
JP nn
```

where nn is the address of a memory byte. The instruction causes the computer to take the instruction at address nn as the next instruction. Because it can be quite difficult to calculate the address for the required jump, the address to be jumped to is usually specified to the assembler program by means of a label, as shown in figure 5.1.



```
 1 REM {
 2 REM *PROGRAMMING FOR
 3 REM *AN INDEFINITE LOOP
 4 REM *       }
 5 REM *       }  INITIAL PART
 6 REM *       }  OF PROGRAM
 7 REM *       }
 8 REM :L1 ADD A.A*START OF
 9 REM *           LOOP
10 REM *   INSTRUCTIONS
11 REM *   TO BE
12 REM *   REPEATED
13 REM JP L1 *JUMP TO START
14 REM *       OF LOOP
15 REM *
16 REM }
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
```

Figure 5.1

Without conditional jumps the use of the JP instruction is limited to providing indefinite loops; this is also shown in figure 5.1.

When the JP L1 instruction is reached the program jumps back to the instruction with the label L1.

The programming of indefinite loops is not recommended because the only way to stop the looping is to switch off the computer.

Of course, if the subroutine to check for the BREAK key is called in the loop then the BREAK key can be used to stop the program. However, as we shall see in the next chapter, the use of conditional jumps overcomes the need to program indefinite loops.

EXERCISE 5.1

Write a set of instructions to output repeatedly an asterisk character to the display until the BREAK key is pressed.

The JP instruction specifies the address, or a label which allows the assembler to calculate the address, of any byte in the memory of the computer. However, because most of the jumps found in programs are to instructions only a few memory bytes away from the memory byte holding the jump instruction, a faster jump instruction is also available. This is the JR instruction known as a relative jump, because the data part of the instruction specifies the number of bytes of memory from the end of the jump instruction to the beginning of the instruction which is to follow the jump.

The form of the JR instruction is

```
JR n
```

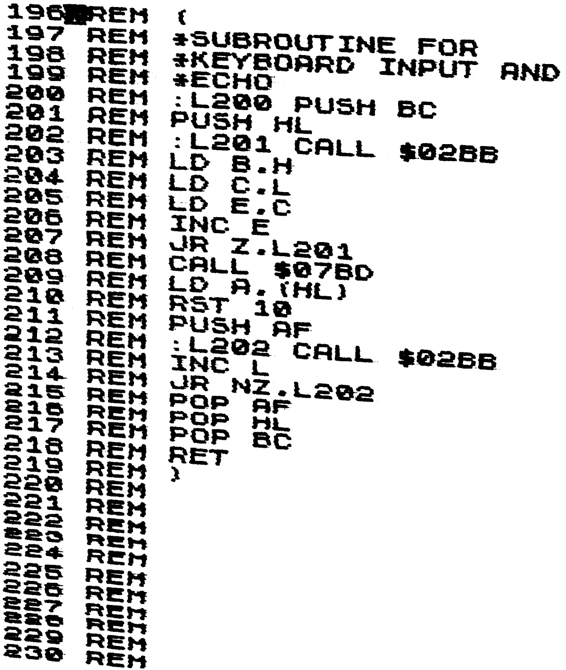
where n is the number of bytes and is a number in the range -128 to +127 as in JR -27. The instruction can specify the label of the instruction to follow the jump instead of the number of bytes. When specifying a jump to a label the assembler will determine the address of the labelled instruction and calculate the number of bytes to be jumped.

EXERCISE 5.2

Using the tables in Appendix E find the number of memory bytes occupied by a JP instruction and by a JR instruction.

5.2 KEYBOARD INPUT

In the same way that you can use a subroutine to output to the display, you can include the following subroutine in your programs to accept input from the keyboard. The subroutine is shown in figure 5.2.



```
196 REM (
197 REM *SUBROUTINE FOR
198 REM *KEYBOARD INPUT AND
199 REM *ECHO
200 REM :L200 PUSH BC
201 REM PUSH HL
202 REM :L201 CALL $02BB
203 REM LD B.H
204 REM LD C.L
205 REM LD E.C
206 REM INC E
207 REM JR Z.L201
208 REM CALL $07BD
209 REM LD A.(HL)
210 REM RST 10
211 REM PUSH AF
212 REM :L202 CALL $02BB
213 REM INC L
214 REM JR NZ.L202
215 REM POP AF
216 REM POP HL
217 REM POP BC
218 REM RET
219 REM )
220 REM
221 REM
222 REM
223 REM
224 REM
225 REM
226 REM
227 REM
228 REM
229 REM
230 REM
```

Figure 5.2

On return from the subroutine the accumulator contains the character code of the character which was input from the keyboard. In the figure the start of the subroutine is indicated by the label :L200 and it would be called by the instruction CALL L200.

When a character is input from the keyboard in an assembly language program it is not automatically displayed on the display unit. This can be a useful feature, since you can make the running of a program dependent on the input of a password which will not be displayed on the screen for other people to see. Normally you will wish to display the character which has been input on the screen; this is done by immediately calling the output subroutine. The output routine, RST 10, has been included in figure 5.2. This technique is known as echoing the input.

Groups of statements which are required more than once in a program, or which are likely to be used by more than one program, should always be written as subroutines.

5.3 CHARACTER CODES AND VALUES

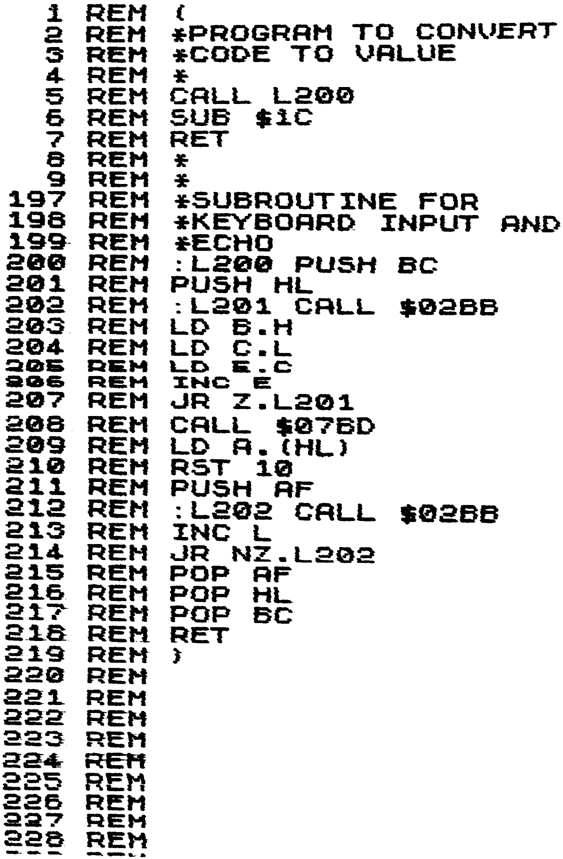
You must be careful when working with numbers that you do not confuse the character code for a digit with its value. The two are quite different.

The values of the decimal digits 0 to 9 are represented in the registers and memory bytes of the computer by the numbers $00 to $09.

The character codes of the digits 0 to 9 are represented in the computer by the numbers $1C to $25.

When carrying out arithmetic in the computer, obviously the values of the digits should be used, but for input and output of digits the character codes are used. The character code of a digit which is input must be converted to its value before it can be used for arithmetic, and after the arithmetic has been completed the value of the answer must be converted back into its character code before it can be output.

Figure 5.3 shows a subroutine which converts the character code of a digit input into the accumulator into its value.



```
  1 REM (
  2 REM *PROGRAM TO CONVERT
  3 REM *CODE TO VALUE
  4 REM *
  5 REM CALL L200
  6 REM SUB $1C
  7 REM RET
  8 REM *
  9 REM *
197 REM *SUBROUTINE FOR
198 REM *KEYBOARD INPUT AND
199 REM *ECHO
200 REM :L200 PUSH BC
201 REM PUSH HL
202 REM :L201 CALL $02BB
203 REM LD B.H
204 REM LD C.L
205 REM LD E.C
206 REM INC E
207 REM JR Z.L201
208 REM CALL $07BD
209 REM LD A.(HL)
210 REM RST 10
211 REM PUSH AF
212 REM :L202 CALL $02BB
213 REM INC L
214 REM JR NZ.L202
215 REM POP AF
216 REM POP HL
217 REM POP BC
218 REM RET
219 REM )
220 REM
221 REM
222 REM
223 REM
224 REM
225 REM
226 REM
227 REM
228 REM
```

Figure 5.3

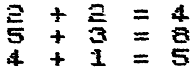
EXERCISE 5.3

The program shown in figure 5.3 is run and the key marked 8 is pressed. What will the contents of the accumulator be immediately after the input, and what will its contents be at the end of the program?

So far only the use of single digit numbers has been discussed. Numbers with more than one digit are input as a series of character codes, and it is a more lengthy process to convert them into their numeric value; this is considered in detail in chapter 7.

5.4 PROGRAM

Write a program which repeatedly inputs, and echoes, two decimal digits from the keyboard and outputs their sum. The sum of the two digits must not exceed 9. A typical run of the program would give the display shown in figure 5.4. To move the display to a new line before outputting the next sum, you should output the character code for NEWLINE.



```
2 + 2 = 4
5 + 3 = 8
4 + 1 = 5
```

Figure 5.4

It is the program's responsibility to output the + and = characters, the input to the program will be the decimal digits only.

6 Conditional jumps and comparisons

6.1 THE FLAG REGISTER

Inside the Z80 microprocessor there is a register, the F register, which is only used to hold information concerning the result produced by the last arithmetic or logic instruction. Of the eight bits in the register only six are used. Figure 6.1 shows the information held in these bits.

S	Z	X	H	X	P/V	N	C

S	- Sign flag
Z	- Zero flag
H	- Half carry flag
P/V	- Parity/overflow flag
N	- Add/subtract flag
C	- Carry flag
X	- These bits are not used

Figure 6.1 The flag register

The bits in this register, which are used, are known as flags, because they are used to signal the result of a previous instruction. The two main flags to consider at this stage are the SIGN FLAG and the ZERO FLAG.

The SIGN FLAG is set to 1 if the result produced by an instruction is negative; otherwise the flag is reset to 0. For example, after execution of the instructions

```
LD A.23
SUB 56
```

the accumulator will contain -33 and the sign flag will be set to 1.

The ZERO FLAG is set to 1 if the result produced by an instruction is zero; otherwise the flag is reset to 0. The zero flag would be reset to 0 after the execution of the two instructions given above.

The operation of the remainder of the flags will be dealt with later in the book.

Not all instructions affect the flags. For example, none of the LD instructions affects any of the flags, whereas the SUB instruction affects all of the flags. You can discover which instructions affect which flags by looking in the tables in Appendix E.

EXERCISE 6.1

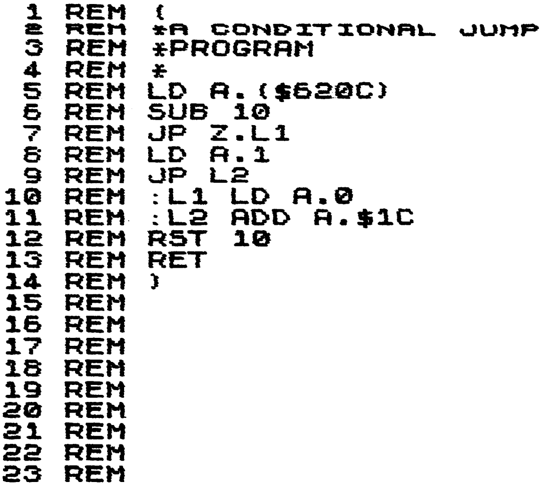
Give the contents of the accumulator and the S and Z flags after the execution of each of the following program instructions:

```
LD A.120

SUB 122

LD B.A

SUB B

ADD A.70

NEG
```

6.2 CONDITIONAL JUMP INSTRUCTIONS

One of the main uses of the flag bits is the control of conditional jump instructions.

Conditional jump instructions allow a program either to continue executing the instructions following the jump instruction, or to



```
 1 REM (
 2 REM *A CONDITIONAL JUMP
 3 REM *PROGRAM
 4 REM *
 5 REM LD A.($620C)
 6 REM SUB 10
 7 REM JP Z.L1
 8 REM LD A.1
 9 REM JP L2
10 REM :L1 LD A.0
11 REM :L2 ADD A.$1C
12 REM RST 10
13 REM RET
14 REM )
15 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
```

Figure 6.2

obey the jump and continue execution with a set of instructions elsewhere in the program. The program goes one way or another depending upon a condition, which is indicated by the state of a flag bit. The program segment in figure 6.2 shows the use of a conditional jump instruction.

The program resets the accumulator to 0 if the value in the memory byte is equal to 10, or sets it to 1 if the value is not equal to 10. The SUB instruction subtracts 10 from the value in memory and sets the flags, in particular the sign and zero flags, according to the result of the calculation. The JP Z.L1 instruction causes a jump to the instruction with the label :L1 if the result of the subtraction is zero, that is, if the zero flag is set to 1, or to execution of the following instructions, LD A.1, if the zero flag is 0. The LD A.1 instruction is followed by the unconditional jump, JP L2, to allow the program to leapfrog the instructions which are carried out when the zero flag is set to 1.

EXERCISE 6.2

Referring to the program in figure 6.2, in what order will the instructions be executed if the value in memory byte $620C is 10?

Four of the conditions which can be tested by a conditional jump are

Zero	- jump if zero flag is 1	e.g. JP Z.L1
Non-zero	- jump if zero flag is 0	e.g. JP NZ.L6
Minus	- jump if sign flag is 1	e.g. JP M.L4
Positive	- jump if sign flag is 0	e.g. JP P.L23

All of the arithmetic conditions used to control jumps in BASIC must be rewritten, when used in assembly language programs, as tests for the sign or zero. For example, the program segment shown in figure 6.2 is equivalent to the following BASIC program segment:

```
30    IF X = 10 THEN 60
40    LET A = 1
50    GOTO 70
60    LET A = 0
70    PRINT A
```

By carrying out a subtraction, and then testing for one of the four conditions shown above, you can carry out tests for =,<,>, or <>. Using more than one test, the assembly language programmer can carry out any test which can be programmed in BASIC.

EXERCISE 6.3

Write a program segment which outputs the letter N if the sum of two numbers, already stored in the B and C registers is negative, the letter Z if the sum is zero, and the letter P if the sum is positive, but not zero.

The state of the zero flag can also be tested by relative jump instructions. Thus the JP Z and JP NZ instructions have equivalent relative jump instructions, JR Z and JR NZ. There are no relative jump instructions equivalent to the JP M and JP P instructions.

6.3 THE COMPARE INSTRUCTION

The main problem with conditional jumps using subtraction is that the original value in the accumulator is lost by the subtraction; it could, of course, be restored by carrying out an addition, but this is often inconvenient. To overcome this problem the Z80 microprocessor includes a compare instruction with the form CP n. It allows the accumulator to be compared with another value without affecting the contents of the accumulator. The compare instruction subtracts the value of the operand from the accumulator and sets the flags, according to the result of the calculation. The result of the subtraction does *not* replace the value in the accumulator; when the flags are set the result of the calculation is discarded. The operand to the instruction can be either a register or a numeric value.

The following program segment shows the straightforward use of the CP instruction to determine if the accumulator contains a specific value:

```
CP 50
JR Z.L4
```

This causes the zero flag to be set to 1 if the accumulator contains 50 and reset to 0 if it contains anything else. It is followed by a relative jump to the instruction labelled L4 if, in fact, the accumulator does contain the value 50.

EXERCISE 6.4

Write a program segment which causes a jump to an instruction labelled :L1 if the value in register B is less than 100, to an instruction labelled :L2 if the value in B is 100, or to an instruction labelled :L3 if the value in B is greater than 100.

The instruction CP 0 is useful for setting the flag register, according to the value in the accumulator, after an instruction which does not affect the flags. For example, in the following sequence:

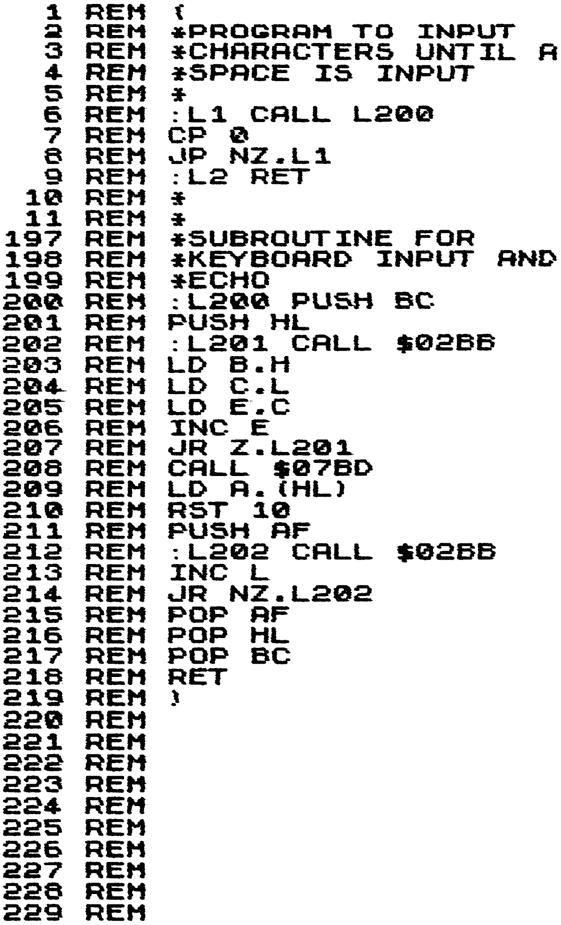
```
LD A.($527A)
CP 0
```

the LD instruction does not affect the flags but the instruction CP 0 sets the flags to give the status of the accumulator.

6.4 CONDITIONAL LOOP TERMINATION

The loops which were used in the previous chapter were referred to as indefinite loops, because there were no instructions in the programs to stop the looping.

There are several different methods of stopping loops and many of them will be detailed in subsequent chapters. At this stage we will consider the method which uses the occurrence of a specific condition, which is tested, to stop the loop. The program in figure 6.3 contains a loop to input characters, using the input and echo subroutine which you have already been given, until a space is input.



```
  1 REM {
  2 REM *PROGRAM TO INPUT
  3 REM *CHARACTERS UNTIL A
  4 REM *SPACE IS INPUT
  5 REM *
  6 REM :L1 CALL L200
  7 REM CP 0
  8 REM JP NZ.L1
  9 REM :L2 RET
 10 REM *
 11 REM *
197 REM *SUBROUTINE FOR
198 REM *KEYBOARD INPUT AND
199 REM *ECHO
200 REM :L200 PUSH BC
201 REM PUSH HL
202 REM :L201 CALL $02BB
203 REM LD B.H
204 REM LD C.L
205 REM LD E.C
206 REM INC E
207 REM JR Z.L201
208 REM CALL $07BD
209 REM LD A.(HL)
210 REM RST 10
211 REM PUSH AF
212 REM :L202 CALL $02BB
213 REM INC L
214 REM JR NZ.L202
215 REM POP AF
216 REM POP HL
217 REM POP BC
218 REM RET
219 REM }
220 REM
221 REM
222 REM
223 REM
224 REM
225 REM
226 REM
227 REM
228 REM
229 REM
```

Figure 6.3

The instructions in the program will be repeatedly executed until a space is input; this puts the character code for a space, $00, into the accumulator. When the CP instruction is executed a zero result will be produced, and then the conditional jump JP NZ.11 will be ignored and program execution will continue with the instruction labelled L2. Note that this program is not using or storing the input characters, it is only searching for an input of a space. The program could be easily modified so that it stored the input characters in successive memory bytes until a space was input.

6.5 PROGRAM

First write a subroutine to put characters into three different categories. On entry to the subroutine the accumulator contains the character code. The subroutine should end with the accumulator unchanged and register B containing:

	-1	if the character is a decimal digit
	0	if the character is a letter of the alphabet
or	1	if it is any other character

Use the subroutine to write a main program which repeatedly inputs and echoes a character from the keyboard, and then outputs a space followed by one of the letters D, A or N indicating that the character just input was a decimal digit, an alphabetic character or neither, respectively. After the letter you should output a NEWLINE.

The program should stop if the NEWLINE key is pressed.

7 Counting loops

7.1 COUNTING LOOPS

There are many different forms of loops used in programs. The indefinite loop and the conditional termination loop have already been used and several more forms will be used throughout the book. One of the simplest and most useful is the counting loop. The counting loop allows a sequence of instructions to be executed a set number of times. The program in figure 7.1 is an example of this; it is a program to input and sum four numbers.

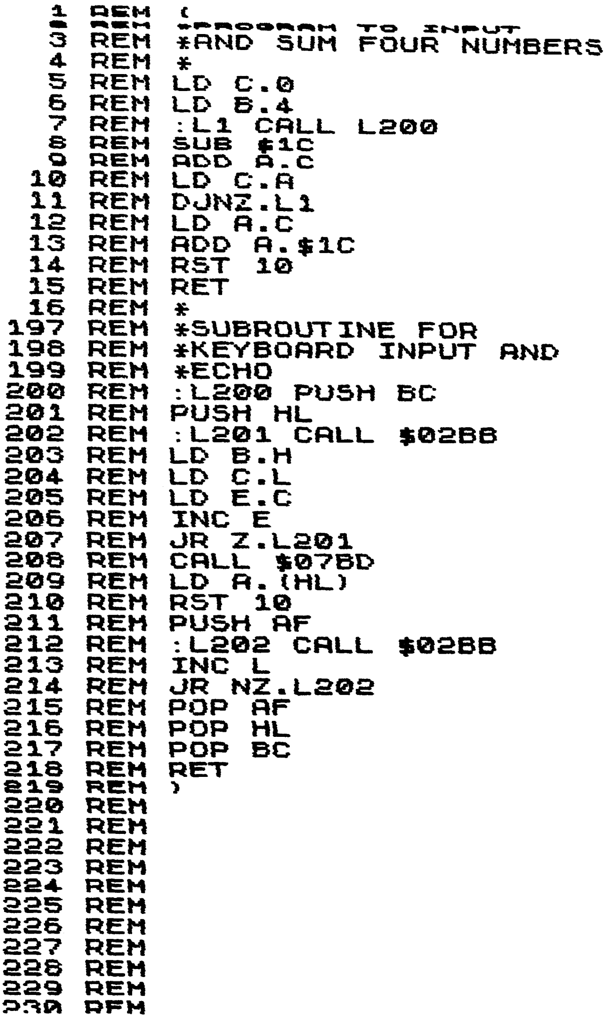
The sequence of instructions between, and including, the instructions CALL L200 and LD C.A will be executed four times. The loop counter, register B, is initially loaded with the number of times that the loop is to be executed using the instruction LD B.4. The instruction DJNZ causes the value in register B to be decreased by one and a jump made to the specified label if the value in B is not zero. When the value in B becomes zero the loop will have executed four times and execution will continue with the instruction following the DJNZ instruction.

Other registers, or memory bytes, can be used as loop counters as well as the B register. The B register is normally used as the loop counter because of the DJNZ instruction, which can only be used with the B register. The DJNZ instruction which stands for Decrement and Jump if Non Zero is equivalent to the two separate instructions, DEC B followed by JR NZ label.

EXERCISE 7.1

With the B register used to store the loop counter, what is the maximum loop count that can be used?

When writing programs it is good practice to make them as general as possible. The program in figure 7.1 would be better if instead of using a fixed value for the loop counter it had taken its initial value for the loop from a particular memory byte. The value in the memory byte could have been put there previously by a main program which uses this subroutine for summing numbers.



```
  1 REM (
  2 REM *PROGRAM TO INPUT
  3 REM *AND SUM FOUR NUMBERS
  4 REM *
  5 REM LD C.0
  6 REM LD B.4
  7 REM :L1 CALL L200
  8 REM SUB $1C
  9 REM ADD A.C
 10 REM LD C.A
 11 REM DJNZ.L1
 12 REM LD A.C
 13 REM ADD A.$1C
 14 REM RST 10
 15 REM RET
 16 REM *
197 REM *SUBROUTINE FOR
198 REM *KEYBOARD INPUT AND
199 REM *ECHO
200 REM :L200 PUSH BC
201 REM PUSH HL
202 REM :L201 CALL $02BB
203 REM LD B.H
204 REM LD C.L
205 REM LD E.C
206 REM INC E
207 REM JR Z.L201
208 REM CALL $07BD
209 REM LD A.(HL)
210 REM RST 10
211 REM PUSH AF
212 REM :L202 CALL $02BB
213 REM INC L
214 REM JR NZ.L202
215 REM POP AF
216 REM POP HL
217 REM POP BC
218 REM RET
219 REM )
220 REM
221 REM
222 REM
223 REM
224 REM
225 REM
226 REM
227 REM
228 REM
229 REM
230 REM
```

Figure 7.1

EXERCISE 7.2

Write a program which inputs a digit n from the keyboard and displays n asterisks.

Although the B register is the natural choice for the loop counters, other registers are used occasionally and an example of a counting loop using the accumulator for the loop counter is shown in figure 7.2.

```
 1 REM (
 2 REM *ACCUMULATOR USED FOR
 3 REM *A LOOP COUNTER
 4 REM LD A.10
 5 REM :L1 ADD HL.BC *BODY
 6 REM *               OF
 7 REM *               LOOP
 8 REM DEC A
 9 REM JR NZ.L1
10 REM RET
11 REM )
12 REM
13 REM
14 REM
15 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
```

Figure 7.2

It is sometimes convenient to use the value of the loop counter within the sequence of instructions controlled by the loop. Figure 7.3 shows a program to find the sum of the numbers from 1 to n. It does it by adding the value of the loop counter to a running total each time that it goes through the loop. The loop counter is being used to count the number of times the loop is executed, and also as the value to be added each time the loop is executed. After looping is complete, the accumulator will contain the final sum. Notice that the accumulator must be set to zero before starting, because unused registers and memory bytes are not automatically set to zero.

```
 1 REM (
 2 REM *SUM OF NUMBERS 1 TO N
 3 REM *
 4 REM LD A.0
 5 REM LD B.($620C) *VALUE OF N
 6 REM :L1 ADD A.B
 7 REM DJNZ.L1
 8 REM LD ($620D).A*FINAL SUM
 9 REM RET
10 REM )
11 REM
12 REM
13 REM
14 REM
15 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
```

Figure 7.3

When the loop counter is being used as a value within the loop it is often inconvenient to count down to 0. If the lowest value taken by the loop counter is not zero, then a compare instruction must be used to detect the end of looping. As the compare instruction can only be used with the accumulator, this means that the accumulator must be used as the loop counter in these circumstances.

EXERCISE 7.3

Write a program which outputs the decimal digits 9 to 0, in descending order.

7.2 NUMBER INPUT

So far we have only considered the input of a single digit number from the keyboard. Inputting a number with more than one digit is not so straightforward, even if the numbers to be input are restricted to positive whole numbers.

If, for example, you wanted to input the number 123 you would have to press the three keys '1', '2' and '3'. This means it would be input as three separate digits. It would be done by three calls to the input and echo subroutine and loading the character codes for the three digits 1, 2 and 3 into successive memory bytes. Then these three character codes held in three separate memory bytes must be converted into the value 123 held in a single memory byte or register.

First the character code for each digit must be converted to the value of the digit. Then the first digit's value is multiplied by 100, the second digit's value is multiplied by 10 and these two products are added to the value of the third digit to give the total value of the number. Thus for the number 123 the calculation will be:

$$1 \times 100 + 2 \times 10 + 3.$$

A more efficient method of calculating the value of a three-digit number, especially on the computer, is first to convert all of the character codes to the values of the digits. Then multiply the value of the first digit by 10 and add to this product the value of the second digit; multiply this sum by 10 and add to this product the value of the third digit. For the number 123 the calculation is:

$$(1 \times 10 + 2) \times 10 + 3.$$

EXERCISE 7.4

Repeated addition can be used as a crude method of multiplying two numbers together since, for example, 3 × 4 is equivalent to

3 + 3 + 3 + 3. So to multiply p by 10, p is added to itself 9 times. Write a subroutine to multiply the value in the accumulator by ten and leave the result in the accumulator.

Although this seems to be a complex process, the method can be extended to give a method which can be used to input a number consisting of any number of digits. The only problem with extending the method is that, at the moment, the numeric value which has been calculated can only be stored in a single memory byte or register, which means in eight bits. Since consideration has not yet been given to storing negative numbers, a memory byte or a register can only hold numbers in the range 0 to 255.

A further problem which arises when inputting numbers with more than one digit is how to indicate to the program that all of the digits have been input. The normal method is to use the input of any non-numeric character as marking the end of the number. Before you store the character code as part of the number, it is checked to see that it is the character code of a numeric digit.

7.3 STACKS

A stack is a method of storing data in memory bytes such that the data items form a list. Data can only be added to the end of the list and only the last item in the list can be used. A stack is often referred to as a Last-in-First-Out list.

Every stack has a top and a bottom. The bottom of the stack is the memory byte which contains the first data item on the stack and the top of the stack is the memory byte which contains the last data item added. A special register, called the SP or stack pointer, is used to hold the address of the memory byte at the top of the stack. Initially, when the stack is empty, the stack pointer contains the address of the memory byte just below the bottom of the stack.

On the ZX81, each data item which is put onto, or removed from, the stack is two bytes long. This means that two memory bytes are allocated for each data item and when data is added to, or removed from, the stack the address in the stack pointer register is automatically increased or decreased by two.

Figure 7.4 illustrates the working of a stack in the ZX81. As you can see the stack is actually stored in memory upside down, so that the stack bottom is at a higher address than the stack top. The stack is usually located at, or close to, the top of memory so that as items are added to it and the size of the stack increases it is not until memory is completely full that the stack can overlap into the program and other data areas.

Stacks are used mainly for the temporary storage of data and addresses. One use of stacks will be discussed in the next chapter when looking at the way in which subroutines are controlled; the following sections will show another common use of the stack.

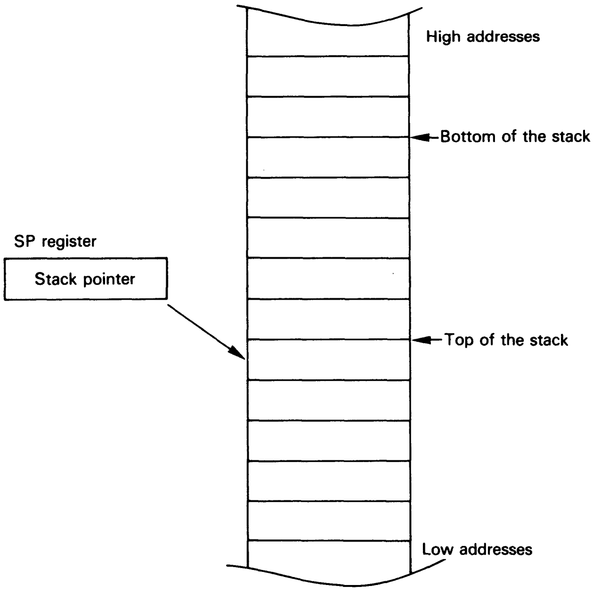


Figure 7.4

7.4 STACK INSTRUCTIONS

To enable stacks to be used easily by the programmer there is a set of special stack instructions. Adding data to a stack is known as pushing the stack, and the instruction to do this is PUSH rp; removing data from the stack is known as popping the stack, and the instruction to do this is POP rp. The rp in the above instructions stands for any of the register pairs AF, BC, DE, HL, IX and IY. The instruction, PUSH AF, means put the contents of the register pair AF onto the stack and decrease the value in the SP register by two. Similarly the instruction, POP BC, means take the data item at the top of the stack and load it into the register pair BC and increase the value in the SP register by two.

Any operation on the stack increases or decreases its size by two bytes.

You can start a stack by loading the SP register with a value at the top of an area of unused memory. So, for example, LD SP.$7000 will set the bottom of the stack to the byte with address $7000.

EXERCISE 7.5

What will be the contents of the registers A, B, C, D, E and SP after the execution of the following program segment?

```
LD A.$0A
LD B.$0B
LD C.$0C
LD D.$0D
LD E.$0E
LD SP.$7000
PUSH AF
PUSH BC
PUSH DE
POP BC
POP DE
```

Another useful stack instruction is EX (SP).HL which exchanges the data item at the top of the stack with the contents of the HL register pair. Exchanges may also take place with the IX and IY registers using the instructions

EX (SP).IX   and   EX (SP).IY

7.5 SAVING AND RESTORING REGISTERS

A subroutine should normally leave all registers with the same contents, when it returns, as they had when the subroutine is entered, except when the registers are being used to pass values back to the main program. This technique is known as saving and restoring registers. On entry to the subroutine, the subroutine will save in memory the present contents of any registers that it is going to use. Then, just before returning, it will reload or restore those values into the correct registers. The simplest method of saving and restoring registers is to use the stack. For example, if a subroutine uses registers B, C and D then the start and end of the subroutine would look like:

```
:L1 PUSH BC
    PUSH DE
    _______
    _______
    _______
    POP DE
    POP BC
    RET
```

When using the stack in this way care must be taken to ensure that registers placed on the stack are removed before the return to the main program. Registers that are saved on the stack must be removed in reverse order so that the last one saved is the first one removed. This method of saving registers is used in the input subroutine. A subroutine which does not save registers should have a comment statement at the beginning of the subroutine indicating which registers are affected by the subroutine.

7.6 PROGRAM

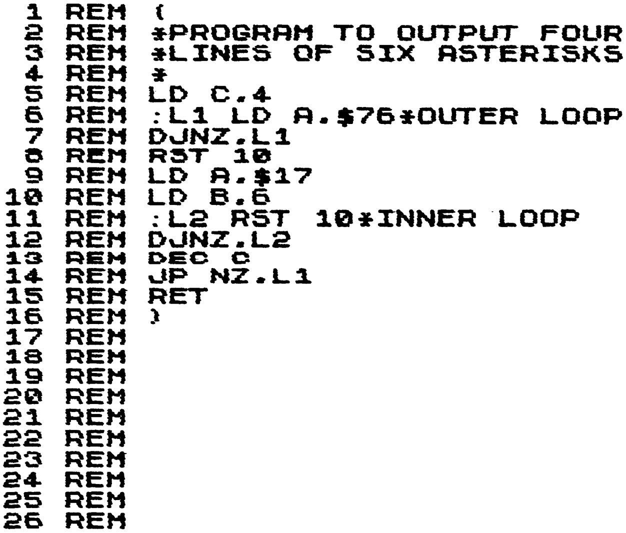
Write a subroutine which inputs an unsigned decimal number consisting of any number of digits. The number is terminated by any non-digit character and may have leading spaces which are ignored by the subroutine. On return from the subroutine, the accumulator should contain the value of the number, and the B register, the number of digits in the number.

Using the above subroutine, and any previous subroutines, write a main program which repeatedly inputs, and echoes, a decimal number terminated by a comma and outputs either the message OK, if the number is less than 50, or the number of digits. The program should return to BASIC when the number 99 is input.

8 Loops within loops

8.1 NESTED LOOPS

A natural follow-on to the loops which have been used so far is to put a loop inside another loop. This type of construction is known as a nested loop. The terms outer loop and inner loop are then used to describe the loop forming the outside of the nest and the loop which is on the inside of the nest. Figure 8.1 shows a program which has two loops, one inside the other. This program outputs four lines of six asterisks. The outer loop uses the C register to count the number of lines and the inner loop uses the B register to count the number of asterisks.



```
 1 REM {
 2 REM *PROGRAM TO OUTPUT FOUR
 3 REM *LINES OF SIX ASTERISKS
 4 REM *
 5 REM LD C.4
 6 REM :L1 LD A.$76*OUTER LOOP
 7 REM DJNZ.L1
 8 REM RST 10
 9 REM LD A.$17
10 REM LD B.6
11 REM :L2 RST 10*INNER LOOP
12 REM DJNZ.L2
13 REM DEC C
14 REM JP NZ.L1
15 REM RET
16 REM }
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
25 REM
26 REM
```

Figure 8.1

EXERCISE 8.1

During a complete run of the program in figure 8.1, how many times will each of the instructions LD A.$17, :L2 RST 10 and DEC C be executed?

Although the example program shows two standard counting loops, there may be more than two loops nested and each of the nested loops may be any type of loop. The way in which a loop is used, and the way in which it ends, does not in any way affect the ability of the loop to be nested inside another loop, or to have a loop nested inside it.

8.2 MORE ADDRESSING MODES

The Z80 microprocessor has ten different addressing modes, that is, ten different ways of stating the operand to be used by an instruction.

Immediate and extended addressing have already been discussed in chapter 3, but three other modes have also been used without any special mention. These are register, implied and relative addressing.

Immediate addressing mode has the value of the operand included in the instruction, as the second byte of the instruction.

Extended addressing mode has the address of the memory byte, which contains the operand value, included in the instruction. In assembly language, extended addressing is indicated by brackets round the operand. Extended addressing is also known as indirect addressing.

Register addressing mode has one of the registers as the operand and the value in the register is the value used by the instruction.

Implied addressing mode has the operand implied, that is, no operand is actually stated because the operand to be used is included in the instruction.

Relative addressing mode is only used for jump instructions which contain a displacement between a jump instruction and the instruction to which the jump is to be made. This is used for JR instructions with or without conditions.

EXERCISE 8.2

Which addressing modes are used by the following instructions?

(i) NEG

(ii) INC D

(iii) CP 50

(iv) LD A.($6352)

Two more addressing modes - register indirect and immediate extended - will be considered now.

In register indirect addressing mode the address of the memory byte, which contains the value to be used by the instruction, is held in a register pair and the name of the register pair is used as the operand. For example, the instruction LD A.(HL) causes the accumulator to be loaded with the value which is in the memory byte whose address is in the HL register pair. When loading the accumulator any of the register pairs may be used to point to the memory byte, but for loading any other single 8-bit register only the HL register pair may be used as a pointer to memory. The register indirect addressing mode can also be used in arithmetic instructions, for example:

```
ADD A.(HL)
SUB (HL)
```

Only the HL register pair can be used to point to the memory byte when using arithmetic instructions. This addressing mode can

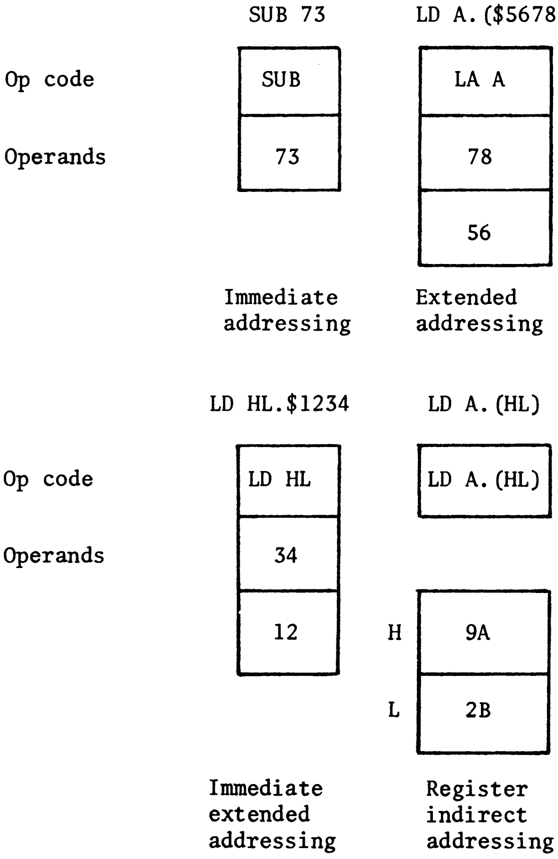
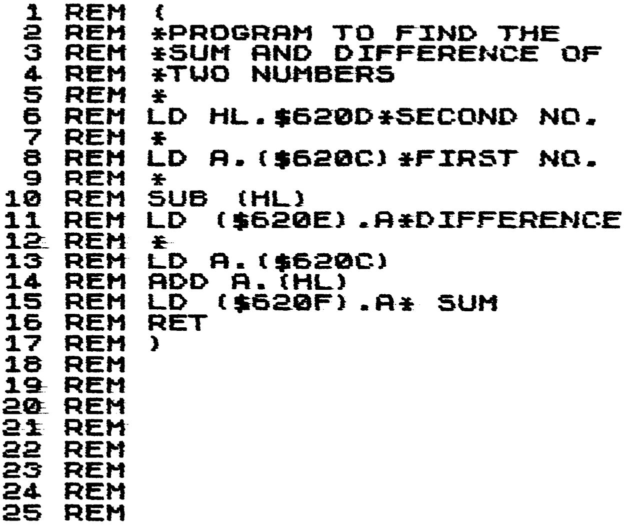


Figure 8.2 Addressing modes

also be used to put a value into memory, so that, for example, the instruction LD (BC).A causes the memory byte, whose address is in the BC register pair, to be loaded with the value in the accumulator. Any of the register pairs BC, DE and HL can be used to point to a memory byte to be loaded from the accumulator, but only the memory byte pointed to by HL can be loaded from the other single registers.

Immediate extended addressing mode is, as its name suggests, an extension of the immediate addressing mode. Immediate addressing refers to 8-bit values, whereas immediate extended addressing refers to 16-bit values.

The operand in the immediate extended addressing mode is the 16-bit value given in the last two bytes of the instruction. For example, the instruction LD BC.$4985 would cause the register pair BC to be loaded with the hexadecimal value $4985, which when converted to binary will give a 16-bit or two-byte value. Any of the register pairs BC, DE, HL, SP, IX and IY may be loaded with a value; the exception is the AF register pair, which should never be used as a register pair for loading. You will find that the HL register pair is most used in this way. Figure 8.2 shows the main addressing modes.



```
 1 REM (
 2 REM *PROGRAM TO FIND THE
 3 REM *SUM AND DIFFERENCE OF
 4 REM *TWO NUMBERS
 5 REM *
 6 REM LD HL.$620D*SECOND NO.
 7 REM *
 8 REM LD A.($620C)*FIRST NO.
 9 REM *
10 REM SUB (HL)
11 REM LD ($620E).A*DIFFERENCE
12 REM *
13 REM LD A.($620C)
14 REM ADD A.(HL)
15 REM LD ($620F).A* SUM
16 REM RET
17 REM )
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
25 REM
```

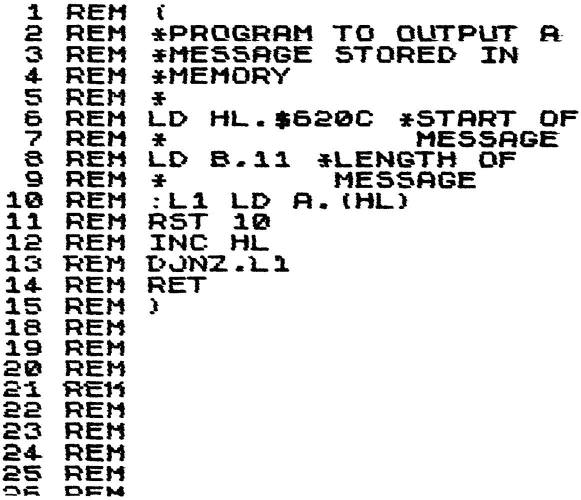
Figure 8.3

The program in figure 8.3 shows the use of immediate extended, and register indirect addressing modes. The program computes the sum and difference of two numbers. The first instruction in the program loads HL with the address of the memory byte which holds the second of the two numbers, using immediate extended addressing. Then the first number is loaded into the accumulator, using extended addressing, and the subtraction is carried out using register indirect addressing. The addition is carried out in a similar way.

8.3 TEXT OUTPUT

When only one, two or three characters are being output to the screen it is usual to output them with separate sets of statements, each of which loads the accumulator with a character and calls the output subroutine. However, for more than three characters this is a very inefficient way to output them to the screen.

A more efficient method of displaying a message on the screen is to output it using a loop. The message must be loaded, as character codes, into memory in consecutive locations and then a program, such as figure 8.4, can be used to output the message.



```
 1 REM (
 2 REM *PROGRAM TO OUTPUT A
 3 REM *MESSAGE STORED IN
 4 REM *MEMORY
 5 REM *
 6 REM LD HL.$620C *START OF
 7 REM *            MESSAGE
 8 REM LD B.11 *LENGTH OF
 9 REM *        MESSAGE
10 REM :L1 LD A.(HL)
11 REM RST 10
12 REM INC HL
13 REM DJNZ.L1
14 REM RET
15 REM )
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
25 REM
26 REM
```

Figure 8.4

When the program is run, the first instruction loads the HL register pair with the address of the memory byte containing the first character, and then the B register is loaded with the number of characters to be output. The next few instructions form a loop which loads each character in turn into the accumulator, calls the output subroutine, increments the HL register pair to point to the next character and finally tests the B register to check whether the end of the message has been reached. This method can be used to produce animated graphics on the ZX81. The required picture can be stored in an area of unused memory and then output when required.

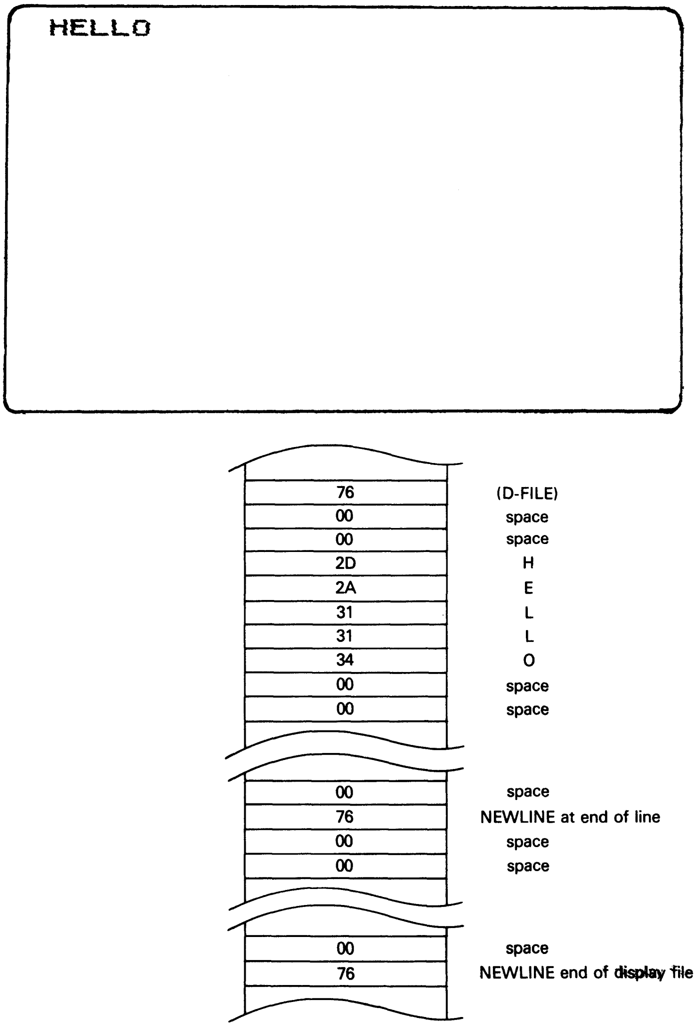
EXERCISE 8.5

The program in figure 8.4 would be more generally useful if instead of counting the characters, characters were output until a memory byte containing an unused value, for example, the value $50, was encountered. Rewrite the program to do this.

The INC HL instruction in the program in figure 8.4 is a new instruction for you to note. All of the register pairs BC, DE, HL, SP, IX and IY can be increased by one using an INC rp instruction, and also decreased by one using a DEC rp instruction. One important difference between incrementing and decrementing register pairs, and incrementing or decrementing single registers, is that none of the flags is affected by the register pair instructions.

8.4 THE DISPLAY FILE

When information is to be output to the screen, it is first of all stored in an area of memory known as the display file. It is

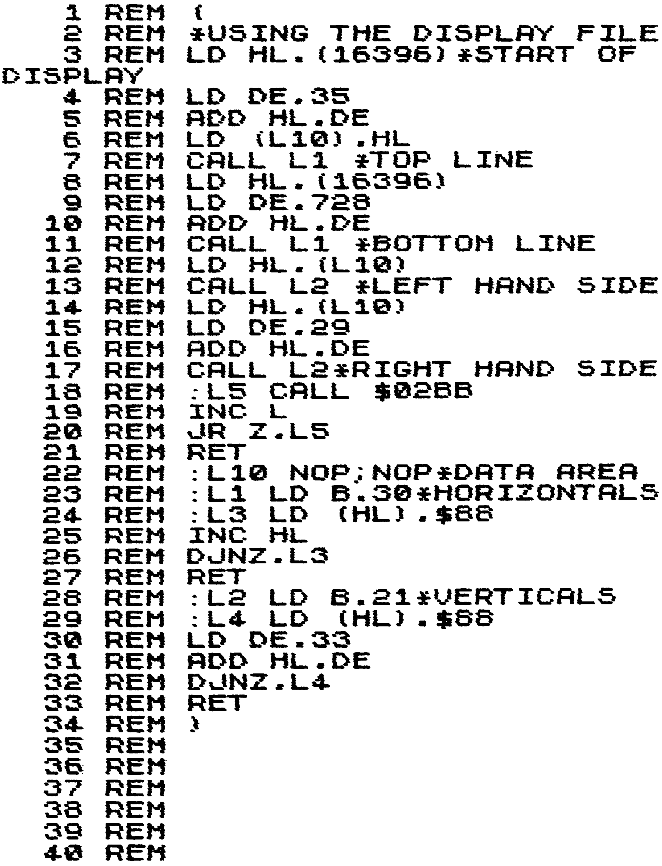


76	(D-FILE)
00	space
00	space
2D	H
2A	E
31	L
31	L
34	O
00	space
00	space
00	space
76	NEWLINE at end of line
00	space
00	space
00	space
76	NEWLINE end of display file

Figure 8.5

then copied from the display file to the screen. The display file contains the character codes of the characters to be displayed on the screen, in the format that will be used on the screen. In a ZX81 with less than 4K of memory the display file initially holds 25 NEWLINE character codes only, but with a computer with more memory the display file initially holds a NEWLINE character, followed by 24 lines each consisting of 32 spaces and a NEWLINE character. This is the display file for a clear screen.

Data can be displayed on the screen by loading it, as character codes, into the display file in memory. Figure 8.5 shows a screen display and the layout of the corresponding display file. When loading data into the display file it is important to see that each line ends with a NEWLINE character code. This is the best method of producing animated graphics on the ZX81. However, when producing animated graphics in assembly language, it is usually necessary to include a delay in the program to slow it down to a viewable speed.



```
   1 REM (
   2 REM *USING THE DISPLAY FILE
   3 REM LD HL.(16396)*START OF
DISPLAY
   4 REM LD DE.35
   5 REM ADD HL.DE
   6 REM LD (L10).HL
   7 REM CALL L1 *TOP LINE
   8 REM LD HL.(16396)
   9 REM LD DE.728
  10 REM ADD HL.DE
  11 REM CALL L1 *BOTTOM LINE
  12 REM LD HL.(L10)
  13 REM CALL L2 *LEFT HAND SIDE
  14 REM LD HL.(L10)
  15 REM LD DE.29
  16 REM ADD HL.DE
  17 REM CALL L2*RIGHT HAND SIDE
  18 REM :L5 CALL $02BB
  19 REM INC L
  20 REM JR Z.L5
  21 REM RET
  22 REM :L10 NOP;NOP*DATA AREA
  23 REM :L1 LD B.30*HORIZONTALS
  24 REM :L3 LD (HL).$88
  25 REM INC HL
  26 REM DJNZ.L3
  27 REM RET
  28 REM :L2 LD B.21*VERTICALS
  29 REM :L4 LD (HL).$68
  30 REM LD DE.33
  31 REM ADD HL.DE
  32 REM DJNZ.L4
  33 REM RET
  34 REM )
  35 REM
  36 REM
  37 REM
  38 REM
  39 REM
  40 REM
```

Figure 8.6

The display file can also be used to check whether a character is in a particular position on the screen by loading the character code from that position in the display file into the accumulator and comparing it with the code of the character.

Since the display file is not static in memory, its starting position is determined by looking at the system variable.

Figure 8.6 is a program to draw a square on the screen using the whole of the screen and keep it there until any key is pressed.

8.5 THE SUBROUTINE MECHANISM

Having used subroutines we are now going to see how they work, that is, what happens inside the computer when a call is made from a program to a subroutine and a return is made to the program from the subroutine. Reference should be made to figure 8.7 when reading the following description. The return address is saved by the computer on the stack. When the program reaches a CALL instruction, the address of the instruction following the call (in this case labelled L2) is automatically pushed on to the top of the stack.

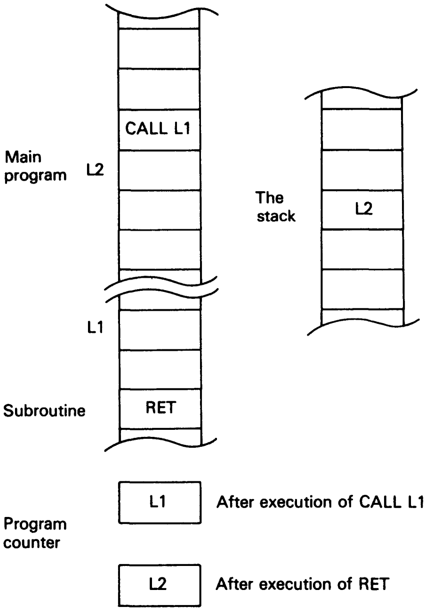


Figure 8.7

Execution of the subroutine is then started by loading the program counter register with the address of the start of the subroutine. This address is specified by the CALL instruction, in this example it is labelled L1. When the subroutine is completed the RET instruction is executed. The effect of the RET instruction is to POP the return address off the stack and load it into the program counter register so that it is the next instruction to be executed.

If a subroutine uses the stack for any reason, then it must ensure that it has POPped off all that it has PUSHed on, so that when the RET instruction is executed the top of the stack contains the return address.

8.6 PROGRAM

Write a program which clears the whole screen by loading the character code for space into the display file. Using nested loops write 32 spaces and a NEWLINE character to one line and then write that line 24 times. After clearing the screen output your name and address to the centre of the screen by loading the correct character codes into the display file in memory. The start address of the display file is stored in the system variable D-FILE.

9 Carry and overflow

9.1 ARITHMETIC CONDITIONS

Two conditions, which are of particular importance when performing addition and subtraction, are carry and overflow. They are conditions which affect bits in the flag register so that they can then be tested by conditional jump instructions. The carry and overflow conditions are used as an aid in determining whether an arithmetic operation has been carried out correctly.

9.2 CARRY

Carry refers to the extra digit produced by the most significant bits, that is the digits on the extreme left, during the addition of two numbers. For example, the sum

```
          00110011      (+51)
        + 00011100      (+28)
          --------
          01001111      (+79)
```

does not produce a carry because the result is within the range of numbers which can be held in an 8-bit binary number. However the sum

```
          11100010      (+226)
        + 10100001      (+161)
          --------
    (1)   10000011      (+131)
```

does produce a carry and gives the wrong result because the answer, which should of course be +387, is outside the range of numbers which can be held in an 8-bit number.

This last sum is only incorrect if it is the addition of two unsigned integers. If the 8-bit numbers are signed integers, in two's complement form, then the sum

```
        11100010     (-30)
      + 10100001     (-95)
        --------
    (1) 10000011     (-125)
```

gives the correct result and the carry can be ignored.

EXERCISE 9.1

Does 11000000 + 01000000 produce a carry? Does it produce the correct result?

9.3 THE CARRY FLAG

When an ADD instruction is executed, the CARRY FLAG is set to 1 if carry occurs, that is, if an extra digit is produced; otherwise it is reset to 0. The carry flag can then be tested by using one of the conditional jump instructions

```
JP  C.label
JP NC.label
JR  C.label
JR NC.label
```

where C stands for carry, that is, the carry flag set to 1, and NC stands for no carry, that is, the carry flag reset to 0.

The carry flag is also used to indicate that a SUB instruction needed to borrow a 1 during subtraction of the two most significant bits.

EXERCISE 9.2

Will the instruction labelled L4 or L5 be executed after the JP instruction in the following sequence?

```
LD A.7
SUB 8
JP NC.L4
:L5 ----
```

The carry flag is also involved in the execution of shift, rotate and decimal adjust instructions, all of which are dealt with later in the book.

There are two instructions which may be used to change the value of the carry flag directly. The instructions are SCF which sets the carry flag to 1 and CCF which complements the carry flag, that is, makes it the opposite to its present value.

EXERCISE 9.3

Write a sequence of instructions to reset the carry flag to zero directly.

9.4 OVERFLOW

Overflow occurs when the result of an operation is outside the range of numbers allowed. For the 8-bit registers and memory bytes the ZX81 expects this range to be -128 to +127.

The addition of a positive and a negative number will never cause overflow because the result will always be within the expected range. However, when adding two positive numbers or two negative numbers overflow may or may not occur. For example the sum

```
          01100100      (+100)
        + 00110001      (+49)
          --------
          10010101      (-107)
```

does not produce the correct arithmetic result because the real sum of the two numbers, +149, is greater than +127; it is therefore outside the expected range and will produce overflow. When two negative numbers are added and the result is outside the expected range, then carry will occur as well as overflow.

Overflow can only occur on subtraction if the two numbers have different signs. For example the subtraction

```
          01111110      (+126)
        - 11000000    - (-64)
          --------
    (1)   10111110      (-66)
```

does not produce the correct arithmetic result because the real result is outside the range. This particular example produces carry as well as overflow.

9.5 THE OVERFLOW FLAG

The parity/overflow flag is used to indicate overflow or parity depending on the instruction. When used to indicate overflow for arithmetic operations the flag is referred to as the overflow flag.

The flag is set to 1 if overflow occurs and reset to 0 if overflow does not occur. The overflow flag is affected by the ADD, SUB, INC, DEC, NEG and CP instructions. The flag can be tested using one of the conditional jump instructions

```
JP PO.label

JP PE.label
```

Unfortunately, because the flag is also used to indicate parity, which will be discussed later, the PO and PE refer to this use and are rather misleading when referring to overflow.

PO stands for overflow, that is, the overflow flag is set to 1, and PE stands for no overflow, that is, the overflow flag is reset to 0. Notice that there are no JR, relative jump, instructions which test for overflow.

9.6 CONDITIONAL CALLS AND RETURNS

In addition to the simple CALL instructions for subroutines, CALL instructions can also be dependent upon a condition being met, similar to the conditional JP instructions. For example, the program segment

```
JP NZ.L10

CALL L20

:L10 ---
```

can be replaced by the single instruction

```
CALL Z.L20
```

thereby saving one instruction.

The conditions which can be tested by the conditional CALL instructions are the same as those which can be tested by the conditional JP instructions, that is, Z and NZ, M and P, C and NC, and PE and PO.

There are also conditional RET instructions, which allow the return from a subroutine to be dependent upon the result of a condition. However, these should be used with great caution because a subroutine should only have one return to the calling program, to ensure that all that has to be done before returning has been completed; in addition, the subroutine can easily be modified without the possibility of forgetting that there are other return instructions, other than the normal one at the end of the subroutine. The end of the subroutine will usually consist of several instructions, not just a RET instruction; for example, normally registers have to be restored before the subroutine returns to the main program.

Conditional RET instructions should only be used if the return from the subroutine is only made when the condition is met.

9.7 PROGRAM

In chapter 7 a subroutine was produced for inputting decimal numbers; amend this subroutine so that it will input signed two-digit decimal numbers such as -21, +84 and 53. This last number is positive but does not include a positive sign. If the first character input is a minus sign, then after the number has been input it should be put into two's complement form using the NEG instruction.

Write a program, which uses the subroutine, and requests two numbers to be input with the prompts

```
INPUT FIRST NUMBER      XX
INPUT SECOND NUMBER     XX
```

and then outputs

```
XX + XX IS word
```

where word is ZERO, NEGATIVE or POSITIVE, followed if appropriate on the next line by

```
PRODUCED OVERFLOW
```

and on the next line if appropriate by

```
PRODUCED CARRY
```

The program should repeatedly input pairs of numbers until END is input in place of the first number, when it should RET to BASIC.

10 Multiplication and division

10.1 SHIFTS

Shift instructions allow the bits of a register, or memory byte, to be moved one bit place to the left, or to the right. There are two types of shift instructions, logical and arithmetic. Logical shifts consider the contents of the register, or memory byte, as a pattern of binary digits when the shift is made. Arithmetic shifts treat the contents of the register, or memory byte, as a signed number, so that the effect of a left shift is equivalent to multiplication by two and a right shift is equivalent to division by two. The Z80 microprocessor in the ZX81 has one logical and two arithmetic shift instructions.

10.2 THE SRL INSTRUCTION

The SRL, or Shift Right Logical, instruction moves the bits in a register, or memory byte, one place to the right. The most significant bit, bit 7, of the register, or memory byte, is reset to 0 and the least significant bit, bit 0, is moved into the carry bit. Figure 10.1 shows the operation of this instruction.

The form of the instruction is:

```
SRL m
```

where m is any of the single 8-bit registers, or a memory byte, which must be pointed to by one of the register pairs HL, IX or IY.

EXERCISE 10.1

If the contents of the accumulator and carry flag are $A7 and 0, respectively, what will be their contents after the execution of a SRL A instruction?

In all shift instructions the bit which is moved out of the register, or memory byte, is placed in the carry flag. This is useful, because the value of the bit which has just been moved out can be checked by any of the conditional jump instructions which test the carry flag, such as JP C.label and JR NC.label.

10.3 THE SRA INSTRUCTION

The SRA, or Shift Right Arithmetic, instruction is the same as the SRL instruction, except that the most significant bit, bit 7, is set to what it was before the shift. Figure 10.1 illustrates the operation of the instruction. Since the value of bit 7 is unchanged by this instruction, it means that if the value in the register or memory byte is a signed number the value of the sign bit is unchanged by the shift. In other words, a positive value will remain positive since bit 7 will stay 0, and a negative value will remain negative since bit 7 will stay 1. The effect of the SRA instruction is to divide the value in the register or memory byte by two and leave the remainder in the carry flag.

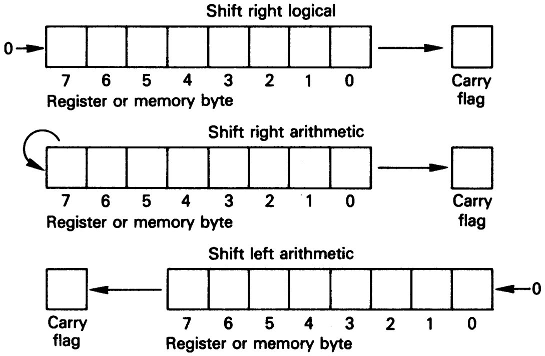


Figure 10.1

The form of the instruction is:

```
SRA m
```

where m is any of the single 8-bit registers, or a memory byte, pointed to by the register pairs HL, IX or IY.

EXERCISE 10.2

Give the value in the B register, the C register and the carry flag, in binary and decimal, after the execution of each of the instructions in the following program segment:

```
LD B.11
SRA B
LD C.F8
SRA C
```

10.4 THE SLA INSTRUCTION

The SLA, or shift left arithmetic, instruction moves the bits in a register, or memory byte, one place to the left. In doing so, the least significant bit, bit 0, of the register, or memory byte, is reset to 0 and the most significant bit, bit 7, is moved into the carry flag. Figure 10.1 illustrates the operation of this instruction.

The effect of the instruction is to multiply the value in the register, or memory byte, by two.

The form of the instruction is:

```
SLA m
```

where m is any of the single 8-bit registers, or a memory byte, which is pointed to by one of the register pairs HL, IX or IY.

As there is no difference between the shift left arithmetic and the shift left logical, there is not a separate instruction for the logical left shift.

EXERCISE 10.3

Using the SLA and ADD instructions, write a program segment which multiplies the value in the accumulator by ten, using the fact that 10 × N is equivalent to N × 2 + N × 2 × 2 × 2.

10.5 EIGHT-BIT MULTIPLICATION AND DIVISION

The repeated addition method of multiplication, which we have been using so far, is very inefficient for multipliers greater than five. A more efficient method for larger multipliers is called 'shift and add'. This method is based on the normal method of long multiplication, and the reason for using this method is illustrated by working through the following multiplication:

```
      10111     multiplicand
    × 01010     multiplier
      -----
      00000       × 0
     10111        × 10
    00000         × 000
   10111          × 1000
  00000           × 00000
  ---------
  011100110         product
```

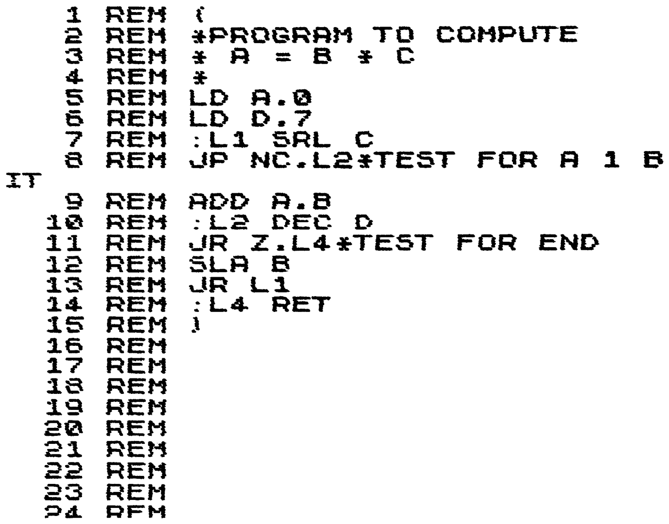
Long multiplication in binary involves multiplying by either one or zero.

Each bit in the multiplier causes the multiplicand to be shifted one bit to the left; if the bit in the multiplier has the value 1, then the shifted multiplicand should be added to the product.

This can be stated as a method for multiplication by 'shift and add' as follows:

> For each bit of the multiplier, working from right to left, add the multiplicand to the product if the multiplier bit is one, otherwise do nothing. Shift the multiplicand one bit to the left, and repeat for the next bit of the multiplier, until the end of the multiplier is reached.

Fig.10.2 shows a program segment which multiplies the contents of the B and C registers, using the 'shift and add' method. The product is left in the accumulator.



```
   1 REM {
   2 REM *PROGRAM TO COMPUTE
   3 REM * A = B * C
   4 REM *
   5 REM LD A.0
   6 REM LD D.7
   7 REM :L1 SRL C
   8 REM JP NC.L2*TEST FOR A 1 B
IT
   9 REM ADD A.B
  10 REM :L2 DEC D
  11 REM JR Z.L4*TEST FOR END
  12 REM SLA B
  13 REM JR L1
  14 REM :L4 RET
  15 REM }
  16 REM
  17 REM
  18 REM
  19 REM
  20 REM
  21 REM
  22 REM
  23 REM
  24 REM
```

Figure 10.2

The program segment only deals with positive signed numbers. It could easily be extended to deal with negative signed numbers. One method of doing this is first to multiply the positive values of the two numbers and then, using the signs of the two numbers, to compute the sign of the product. The sign of the product is simply determined from the rule that equal (or the same) signs give a positive product and unequal (or different) signs give a negative value.

Division can be performed by 'repeated subtraction', which is similar to multiplication by repeated addition, or by a method similar to long division, in which a check is made to see if the divisor goes into (is smaller than) the remaining dividend. This method is similar to multiplication by 'shift and add' and it can be referred to as the 'shift and subtract' method.

10.6 PROGRAM

Write a new multiplication subroutine which uses the 'shift and add' method and deals with all 8-bit signed numbers.

Write a subroutine which divides a signed number in register B by a signed number in register C, and leaves the answer in the accumulator and the remainder in register D.

Using the division subroutine to divide by ten, write a subroutine which will output the contents of the accumulator as a signed number in the range -128 to +127. Leading zeros should not be output and positive numbers should be output without a + character.

Using the above subroutine and any previous subroutines, write a program which inputs two signed numbers separated by either a * character or a / character followed by a = character. The program should then output either the product of the numbers or the quotient and remainder, depending upon which symbol was input. A typical display would look like:

```
-11*5 = -55
125/10 = 12 R5
```

11 The logical bits

11.1 BIT INSTRUCTIONS

The Z80 microprocessor has an extensive range of instructions which use data bit by bit. There is a series of bit instructions which allows individual bits of a register, or memory byte, to be tested, set to 1 or reset to 0. The number of the bit used in the operation is specified in the instruction; for this purpose the bits are numbered right to left as in figure 11.1. All bit instructions operate on any of the 8-bit registers, or a memory byte pointed to by HL, IX or IY.

7	6	5	4	3	2	1	0

Figure 11.1 Bit positions

11.2 THE BIT TEST INSTRUCTION

The BIT instruction tests a specified bit of a register or memory byte and sets the zero flag accordingly. Pixel graphics are produced by setting bits in the memory bytes which form the display file, so by testing bits in the memory bytes in the display file area you can find out if a particular pixel on the screen is lit or not. The zero flag is set to 1 if the bit is zero and to 0 if the bit is not zero. For example, the instruction

```
BIT 6.E
```

tests bit number six of register E, and if register E contained 01000100B the zero flag would be set to 0, because the bit is a 1.

A BIT instruction is usually followed by a 'jump on zero', or 'jump on non-zero' instruction, in which case, the zero and non-zero conditions refer to the value of the specified bit. So there is normally no need to remember how the zero flag is set by a bit instruction.

EXERCISE 11.1

What will be the value of the zero flag if a BIT 1.A instruction is executed when the accumulator contains $FD?

11.3 THE SET AND RES INSTRUCTIONS

The SET instruction allows a specified bit of a register, or memory byte, to be given the value 1. For example, the instruction

```
SET 2.C
```

gives bit 2 of register C the value 1. The other bits of C remain unchanged.

The RES instruction allows a specified bit of a register, or memory byte, to be reset to 0. For example, the instruction

```
RES 5.(HL)
```

gives bit 5 of the memory byte pointed to by HL the value 0. The other bits of the memory byte remain unchanged. When using the SET or RES instructions to give a specific value to a bit in a memory byte, the memory byte must be pointed to by the HL, IX or IY registers.

EXERCISE 11.2

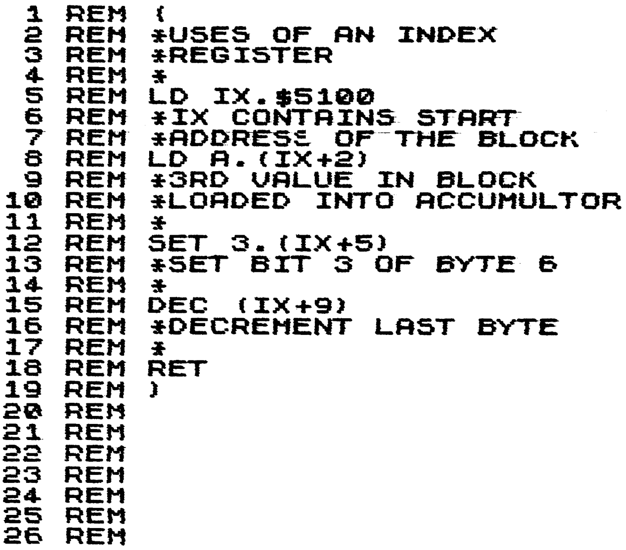
Write a program segment which checks if the number in the accumulator is odd or even, and then sets bit 7 of the B register to 1 if the number is odd, or to 0 if the number is even, without affecting the other bits in the B register.

11.4 THE INDEX REGISTERS

The two 16-bit registers, IX and IY, are called index registers. They are normally used to store the address of the first byte in a block of memory bytes; other bytes in the block can be used by giving their displacement from the start of the block.

The program segment in figure 11.2 shows how an index register can be used.

This program uses the block of ten bytes starting at $5100. Before any bytes in the block can be accessed, the index register is set to point to the first byte, using the LD IX.$5100 instruction. Any particular byte in the block can now be referenced by specifying the index register, plus the displacement from the start of the block. For example, the fourth byte of the block is referenced by (IX+3).



```
 1 REM {
 2 REM *USES OF AN INDEX
 3 REM *REGISTER
 4 REM *
 5 REM LD IX.$5100
 6 REM *IX CONTAINS START
 7 REM *ADDRESS OF THE BLOCK
 8 REM LD A.(IX+2)
 9 REM *3RD VALUE IN BLOCK
10 REM *LOADED INTO ACCUMULTOR
11 REM *
12 REM SET 3.(IX+5)
13 REM *SET BIT 3 OF BYTE 6
14 REM *
15 REM DEC (IX+9)
16 REM *DECREMENT LAST BYTE
17 REM *
18 REM RET
19 REM }
20 REM
21 REM
22 REM
23 REM
24 REM
25 REM
26 REM
```

Figure 11.2

The IY index register can be used in exactly the same way as the IX index register.

The index registers are useful for referring to blocks of memory in which the data in each byte is distinct, but related to the other bytes in the block. For example, the block of memory could hold a table of values.

In programs with no requirement to refer to blocks of memory the index registers can still be used usefully. When used without a displacement they can be used as pointers to particular memory bytes, in much the same way as the HL register. Often it is necessary to specify a zero displacement when using the index registers as pointers to single memory bytes.

11.5 LOGIC OPERATORS

There are several instructions in the Z80 microprocessor which allow logical operations to be performed between corresponding bits in the accumulator and an 8-bit operand.

To understand the operation of the logical instructions you need to know the rules of the basic logical operators. The basic logical operators are AND, OR, XOR and NOT. The rules are normally displayed in tabular form as follows:

0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

These show how corresponding bits are combined by the logical operators.

Apart from the NOT operator, the logical operators use two one-bit values for data and produce a one-bit result. For example, the result of the AND operator is one, if and only if, the two data bits are one; otherwise the result is zero.

EXERCISE 11.3

The logical operator XOR is known as the exclusive OR operator. Express the operation of this operator in words.

11.6 LOGICAL INSTRUCTIONS

All of the logical instructions in the Z80 perform a logical operation between the bits in the accumulator and their corresponding bits in the operand, putting the result in the accumulator. For example, if the accumulator contains 00101010B and register B contains 11001111B, then the instruction AND B would produce a result of 00001010B, as follows:

Contents of A	00101010
Contents of B	11001111
Contents of A after AND B	00001010

Only where the bits of both A and B are 1 is the bit set to 1 in the result. Corresponding bits in the accumulator and the operand are logically operated on, in isolation from the other bits.

The operand of a logical instruction may be a register, an 8-bit value or a memory byte pointed to by the HL, IX or IY registers.

The logical instructions may be specified together as:

AND	r
	n
OR	(HL)
	(IX)
XOR	(IY)

Since logical instructions can only be performed using the value in the accumulator, the use of the accumulator is not specified in the instruction.

Performing a logical instruction will set the sign and zero flags in the same way as arithmetic instructions.

The NOT logical operation is performed by the CPL instruction. The mnemonic CPL stands for the word 'complement'. The effect of the instruction is to change all the 0's in the accumulator to 1's and all the 1's to 0's.

EXERCISE 11.4

Give the contents of the accumulator and the S and Z flags, in binary, after the execution of each instruction in the following sequence:

```
LD A.$B5
LD C.$F0
AND $1F
OR C
XOR $CC
CPL
XOR A
```

The instruction XOR A is commonly used to give a zero value in the accumulator.

11.7 PROGRAM

Two other common logical operations are NOR and NAND, which are short for NOT OR and NOT AND respectively. The rules of these two logical operations are:

0 NOR 0 = 1	0 NAND 0 = 1
0 NOR 1 = 0	0 NAND 1 = 1
1 NOR 0 = 0	1 NAND 0 = 1
1 NOR 1 = 0	1 NAND 1 = 0

from which it can be seen that the result of the NOR operation is the complement, or NOT, of the OR result and the result of the NAND operation is the complement of the AND result.

Write a Logic Operation Trainer program which repeatedly allows input of the form:

a lop b = c

where a, b and c are 0 or 1 and lop is one of:

OR (with a space input after the R)

AND

XOR

```
        NOR
    or  NAN   (short for NAND)
```

Output should be on the same line as the input and will be TRUE if the input is correct, otherwise FALSE.

A training session is ended by inputting a statement with END in place of a lop.

12 Rotate instructions and parity

12.1 ROTATE INSTRUCTIONS

Rotate instructions are similar to shift instructions, except that the bit which is moved out of one end is carried round, and put into the other end, hence the name rotate.

There are several rotate instructions available to the ZX81 assembly language programmer. Four of the rotate instructions use the accumulator, and the remainder use one of the single 8-bit registers, or a memory byte.

Some of the rotate instructions include the carry flag in the rotation; while others, referred to as rotate circular instructions, do not include the carry flag in the rotation, although it is still affected by the instruction.

Rotates may be to the left or right and they are movements of one bit only. All rotations are similar to logical shifts.

12.2 ACCUMULATOR ROTATE INSTRUCTIONS

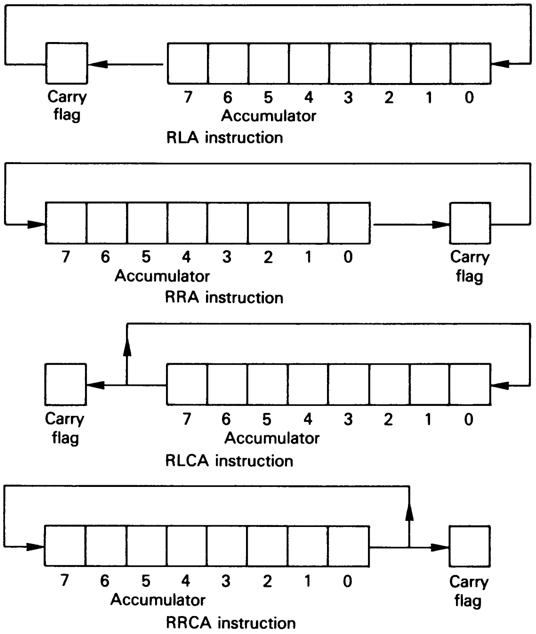
The accumulator is the register which is generally used for arithmetic operations, so there are four rotate instructions which use the accumulator without it having to be named as an operand. These instructions are one-byte instructions, whereas the rotate instructions using any other register are two-byte instructions.

Rotations of the accumulator can be to the left or to the right, and for each direction the carry flag may, or may not, be included within the rotation. This gives the four instructions:

RLA	rotate left accumulator
RRA	rotate right accumulator
RLCA	rotate left circular accumulator
RRCA	rotate right circular accumulator

Figure 12.1 shows the operation of these four instructions.

The rotate left accumulator instruction includes the carry flag in the rotation. The contents of the accumulator move to the left one bit position; in doing so bit 7 is moved into the carry flag and the bit in the carry flag is moved into bit 0.



RLA instruction

RRA instruction

RLCA instruction

RRCA instruction

Figure 12.1

The rotate right accumulator instruction operates in a similar fashion to the rotate left accumulator, except that the contents of the accumulator move to the right. Bit 0 is moved into the carry flag and the bit in the carry is moved into bit 7.

The rotate left circular accumulator instruction does not include the carry flag in the rotation; however, the carry flag is still involved. The contents of the accumulator are moved left one bit position; in doing so bit 7 is moved into the carry flag and also round into bit 0. So after a left circular rotation the value of bit 0 and the carry flag will be the same.

The rotate right circular accumulator instruction operates in a similar fashion to the rotate left circular accumulator instruction, except that all the bits in the accumulator are moved one place to the right and bit 0 is moved into the carry flag and also into bit 7.

EXERCISE 12.1

Assuming that the accumulator contains 10101011B and the carry flag contains 0, what will be the contents of the accumulator and the carry flag, in binary, after the execution of each of the instructions in the following sequence?

```
RLA
RLCA
RRA
RRCA
```

12.3 REGISTER AND MEMORY BYTE ROTATIONS

The four types of rotate that we have just seen for the accumulator can also be applied to any of the single 8-bit registers, or a memory byte, using the instructions:

RL m	rotate left register or memory byte
RR m	rotate right register or memory byte
RLC m	rotate left circular register or memory byte
RRC m	rotate right circular register or memory byte

where m is any of the single registers, or a memory byte, pointed to by one of the register pairs HL, IX and IY.

The operation of these instructions is exactly the same as the corresponding accumulator rotate instructions.

Because a rotate accumulator instruction occupies only one byte, whereas a shift accumulator instruction occupies two bytes, the rotate accumulator instruction is often used to perform a shift operation instead of a shift accumulator instruction. Although this gives a saving in memory space, and in time of execution, it should be used with care because it makes the program harder to understand.

12.4 PACKING AND UNPACKING

The term packing refers to the use of a register, or memory byte, to hold two or more distinct data items. We say that the items are packed into the memory byte or register. For example, a memory byte could be packed with the sex and the age of a person. Bit 7 of the memory byte could be used to indicate the sex of the person, say 0 for female and 1 for male. Bits 0 to 6 of the memory byte could be used to hold the person's age; this gives an age range from 0 to 127. Figure 12.2 shows an example of a memory byte packed in this fashion. The example is for a male aged 51.

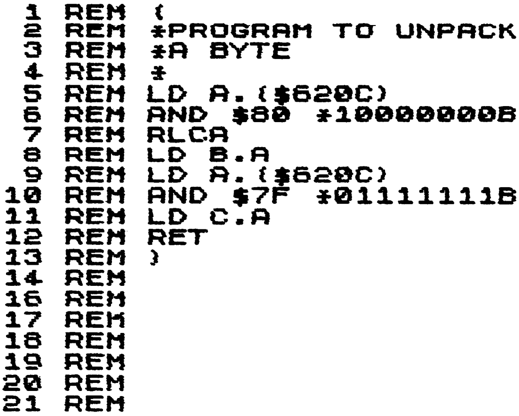
1	0	1	1	0	0	1	1
7	6	5	4	3	2	1	0

Figure 12.2 A packed byte

Assuming that the sex and age of one thousand people are to be held in memory, then packing the data into one byte means that one thousand bytes will be used. If the sex and age were held in separate memory bytes, then two thousand bytes would be needed to hold the same information.

To be able to use packed data it is usually necessary to unpack the memory bytes into separate data items. For example, to use the sex and age packed as shown above, it would be necessary to unpack the sex bit into one register and the age bits into another register.

In general, packing is carried out using the OR and shift instructions, while unpacking is carried out using the AND and shift instructions. Figure 12.3 gives a program segment for unpacking a memory byte holding sex and age. After unpacking, the sex is in the B register and the age is in the C register. There are many ways of unpacking data from memory bytes; some use fewer instructions than others, while some are easier to understand. The method used for unpacking also depends on the number, length and position of the data items that have been packed.



```
 1 REM (
 2 REM *PROGRAM TO UNPACK
 3 REM *A BYTE
 4 REM *
 5 REM LD A.($620C)
 6 REM AND $80 *10000000B
 7 REM RLCA
 8 REM LD B.A
 9 REM LD A.($620C)
10 REM AND $7F *01111111B
11 REM LD C.A
12 REM RET
13 REM )
14 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
```

Figure 12.3

EXERCISE 12.2

Write a program segment which packs into memory byte $620C, the sex, from bit 0 of the B register, and the age, from the C register.

12.5 PARITY

Parity refers to the number of 1's in a binary number. A binary number is said to have even parity if the number of 1's is even and odd parity if the number of 1's is odd. For example:

01101000 has odd parity

11110011 has even parity

and 10110011 has odd parity.

Parity is often used in computers when transferring information from one part of the computer to another, or when transferring information between computers. A parity bit is added to the information, before it is transferred, to make the number of 1 bits either even or odd. At the destination, the parity of the information is checked to make sure that it is still correct. It is not a foolproof check that the information has been transferred correctly, since if an even number of bits have changed their value during the transfer, the parity would remain correct. However, a parity check will find the majority of errors.

12.6 THE PARITY FLAG

The parity/overflow flag is used to indicate the parity of a result after most of the rotate instructions and the shift and logical instructions.

If the number of 1's in the register, or memory byte, is even after any of these instructions has been executed, the parity flag will be set to 1, but if the number of 1's is odd the parity flag will be reset to 0.

EXERCISE 12.3

Assuming that the accumulator contains $B9, what will be the value of the parity flag after each of the instructions in the following program segment?

```
AND $FE
SLA A
RLA
```

The parity condition can be tested by any of the instructions:

```
    JP PE.label
    JP PO.label
    CALL PE.label
    CALL PO.label
    RET PE
and RET PO
```

The initials PE and PO stand for Parity Even and Parity Odd respectively.

EXERCISE 12.4

Write a subroutine which checks the parity of the accumulator. On entry to the subroutine, register B contains either 0, to indicate a check for even parity, or 1, to indicate a check for odd parity. On exit from the subroutine, register C should contain 0 if the parity was correct, and 1 if there is a parity error.

12.7 PROGRAM

Write a subroutine which outputs onto the screen the contents of the accumulator as eight binary digits. The subroutine will have to extract each bit, separately, from the accumulator and output the character code for 1 or 0 depending on the value of a bit.

Write a subroutine which packs the accumulator from four consecutive memory bytes, as follows:

bits 0 and 1 of M to bits 6 and 7 of the accumulator

bits 0 and 1 of M + 1 to bits 4 and 5 of the accumulator

bit 0 of M + 2 to bit 3 of the accumulator

and bits 0 to 2 of M + 3 to bits 0 to 2 of the accumulator.

M is the first of the four memory bytes.

The address of M is contained in the HL register pair on entry to the subroutine.

Write a subroutine to unpack the accumulator by reversing the previous subroutine.

Using these subroutines, and any previous subroutines, write a main program which inputs ten sets of four numbers, and packs each set of four numbers into the accumulator, as specified above. The four numbers are input as decimal numbers in the ranges 0 to 3, 0 to 3, 0 or 1, 0 to 7 respectively. The ten sets of packed numbers should be stored in memory in ten consecutive memory bytes. After the ten sets have been input, they should be output in the packed form, in binary, for checking purposes.

13 Multiple byte arithmetic

13.1 WHY MULTIPLE BYTE?

So far we have only been concerned with single byte arithmetic, that is, arithmetic using 8-bit operands and giving 8-bit results. The range of numeric values which can be manipulated by 8-bit arithmetic is small, so we often need to use more than one byte to hold our numbers; we then need to carry out arithmetic using sixteen or more bits. The microprocessor inside the ZX81 has assembly language instructions which allow 16-bit arithmetic to be performed directly. These instructions can also be used to provide 32-bit arithmetic, 48-bit arithmetic and so on. The 16-bit arithmetic instructions also allow additional loop facilities.

All the 16-bit arithmetic instructions are detailed in Appendix E.

13.2 THE 16-BIT ARITHMETIC INSTRUCTIONS

The main 16-bit arithmetic instructions use the HL register pair as an accumulator. For example the 16-bit addition instruction

```
ADD HL.DE
```

adds the value in register pair DE to the value in the register pair HL, leaving the result in HL. The general form of the instruction is:

```
ADD HL.ss
```

where ss is any one of the register pairs BC, DE, HL or SP.

The program segment

```
LD BC.2054
LD HL.1362
ADD HL.BC
```

shows two 16-bit numbers added together with the result left in the HL register pair. After execution of the ADD HL.BC instruction, the HL register pair will contain the value 3416.

EXERCISE 13.1

What is the range, in decimal, of unsigned and signed numbers which can be dealt with by 16-bit arithmetic?

There is another 16-bit add instruction; this includes in the addition any carry produced by a previous operation. The general form of the 16-bit Add with Carry instruction is:

```
ADC HL.ss
```

where ss is the same range of register pairs as for the 16-bit ADD instruction. The value in the register pair ss is added to the value in HL, along with the value in the carry flag, and the result is placed in HL.

The ADC HL.ss instruction can be used to provide a simple 32-bit arithmetic facility. Figure 13.1 is a program segment which adds two 32-bit signed numbers. Since we are using 32-bit numbers, each number to be added, and the result, is stored in four consecutive memory bytes. When the program is run, the least significant sixteen bits of the two 32-bit numbers are added using the ADD HL.DE instruction and stored in the last two bytes of the result. During this addition any carry will be recorded in the carry flag. Then the most significant sixteen bits of the numbers are added using the ADC HL.DE instruction, which will also include any carry from the previous instruction in the addition. The result from this addition is then stored in the first two bytes of the result, to form a complete 32-bit, or four-byte, number. Overflow is detected by checking the overflow flag after execution of the ADC instruction.

EXERCISE 13.2

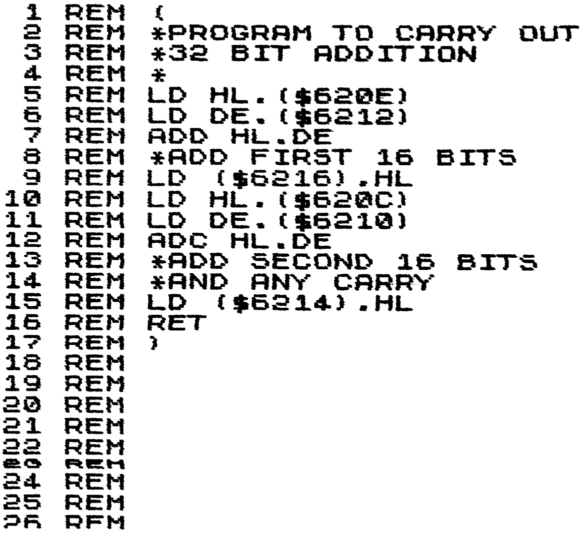
If, before running the program in figure 13.1, the four consecutive memory bytes, starting at $620C, hold the numbers $05, $A1, $63, $82, and the four consecutive memory bytes, from $6210, hold the numbers $00, $C6, $A5, $7E, what will be the contents of the four memory bytes from $6214?

The above technique can be extended to add multiple 16-bit numbers.

There is only one 16-bit subtract instruction, a subtract with carry instruction. The general form of the 16-bit subtract instruction is:

```
SBC HL.ss
```

where ss is any one of the register pairs BC, DE, HL and SP, the same as for the 16-bit add instructions. The instruction causes both the value in the register pair ss and the value in the carry flag to be subtracted from the value in HL, with the result being left in the HL register pair.



```
 1 REM {
 2 REM *PROGRAM TO CARRY OUT
 3 REM *32 BIT ADDITION
 4 REM *
 5 REM LD HL.($620E)
 6 REM LD DE.($6212)
 7 REM ADD HL.DE
 8 REM *ADD FIRST 16 BITS
 9 REM LD ($6216).HL
10 REM LD HL.($620C)
11 REM LD DE.($6210)
12 REM ADC HL.DE
13 REM *ADD SECOND 16 BITS
14 REM *AND ANY CARRY
15 REM LD ($6214).HL
16 REM RET
17 REM }
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
24 REM
25 REM
26 REM
```

Figure 13.1

The SBC HL.ss instruction can be used, in a similar way to the ADC HL.ss instruction, to perform subtractions with numbers which are multiples of sixteen bits.

When subtracting single 16-bit numbers, it is necessary to set the carry flag to 0, using the SCF and CCF instructions, before executing the SBC HL.ss instruction to ensure that nothing, other than zero, is subtracted from the true result.

EXERCISE 13.3

Write a program segment which subtracts the value in the BC register pair from the value in HL, assuming single 16-bit arithmetic.

One important point to note about the 16-bit ADD, ADC and SBC instructions is their effect on the flags. The 16-bit ADC and SBC instructions set the carry, zero, overflow and sign flags as you would expect, but the 16-bit ADD instruction only causes the carry flag to be set. A 16-bit ADC instruction, preceded by the setting of the carry flag to zero, can be used if the setting of the zero, overflow and sign flags is required during a single 16-bit addition.

13.3 EXTENDED LOOPS

The 16-bit load, increment/decrement and arithmetic instructions can be used for loops in which the loop counter has a range of 0 to 65535. However, there are special 16-bit instructions, involving the IX and IY index register pairs, which can be used for such loops. The basic loop structure using one of the index registers looks like:

```
LD IX.nn
:L5 ---
------
-------
INC IX
JP L5
```

The IX index register pair is first loaded with an initial value, either directly as shown, or indirectly using a LD IX.(nn) instruction. At the end of the set of instructions to be repeated, the index register is incremented by one, using an INC IX instruction, or decremented by one, using a DEC IX instruction. A jump is then made back to the first of the instructions to be repeated.

The loop must be terminated, either by a condition occurring within the loop or by IX becoming a specific value. However, it must be remembered that the INC IX and DEC IX instructions do not affect any of the flags.

The IY index register can, of course, be used wherever the IX index register can be used.

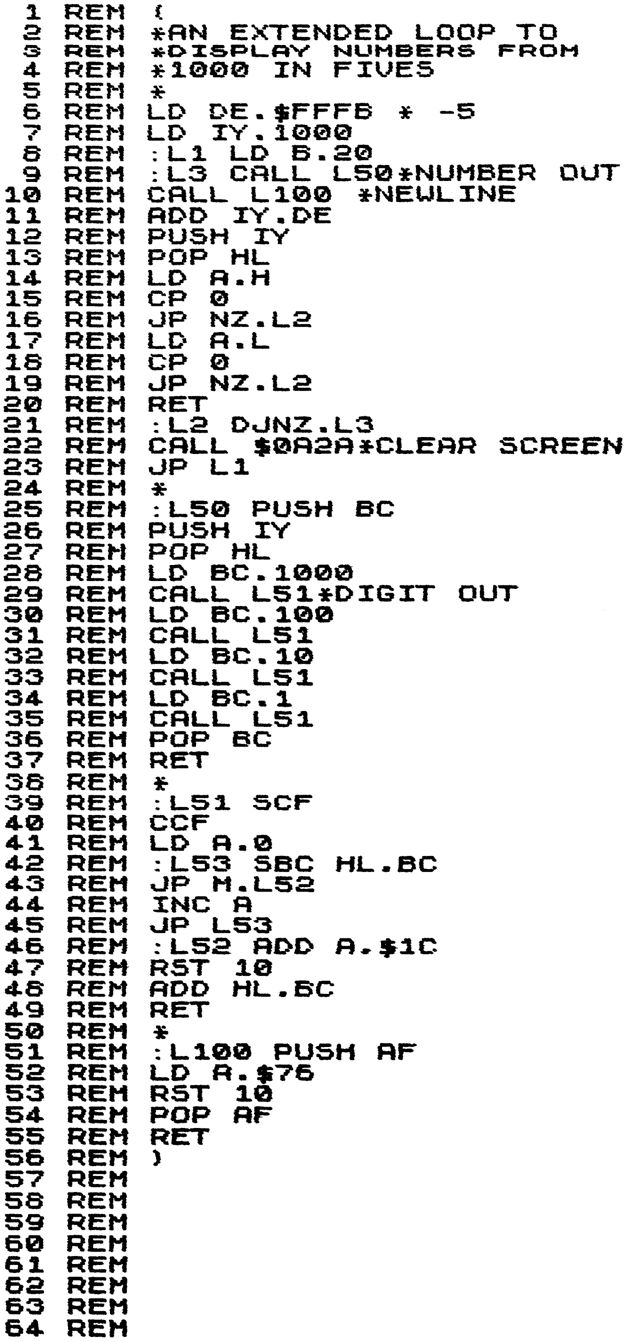
There is a special ADD instruction, relating to the index registers, which allows, among other things, these registers to be incremented and decremented by a value other than one, when used as the loop index register. The form of the instruction is:

```
ADD IX.pp
ADD IY.rr
```

where pp is any of the register pairs BC, DE, IX and SP, and rr is any of the register pairs BC, DE, IY and SP.

The use of the ADD IY.rr instruction is shown in the program in figure 13.2, which outputs the numbers 1000 to 0 in decrements of 5.

The index register, IY, which is used as the loop counter is given the starting value of 1000, and the DE register pair is set to the decrement value. Each time through the loop a subroutine is called which outputs the value in IY as an unsigned number in the range 0 to 65535. After the repeated instructions, the value in IY is decremented by 5, by adding -5, the value in register pair DE, to IY. IY is then checked to see if it is zero. This check is not so straightforward as you might think. There are several methods of performing the check, none of them is particularly neat. In the program, the check is made by splitting the sixteen bits in register IY into the two 8-bit registers B and C, each of which is then checked for zero.



```
 1 REM (
 2 REM *AN EXTENDED LOOP TO
 3 REM *DISPLAY NUMBERS FROM
 4 REM *1000 IN FIVES
 5 REM *
 6 REM LD DE.$FFFB * -5
 7 REM LD IY.1000
 8 REM :L1 LD B.20
 9 REM :L3 CALL L50*NUMBER OUT
10 REM CALL L100 *NEWLINE
11 REM ADD IY.DE
12 REM PUSH IY
13 REM POP HL
14 REM LD A.H
15 REM CP 0
16 REM JP NZ.L2
17 REM LD A.L
18 REM CP 0
19 REM JP NZ.L2
20 REM RET
21 REM :L2 DJNZ.L3
22 REM CALL $0A2A*CLEAR SCREEN
23 REM JP L1
24 REM *
25 REM :L50 PUSH BC
26 REM PUSH IY
27 REM POP HL
28 REM LD BC.1000
29 REM CALL L51*DIGIT OUT
30 REM LD BC.100
31 REM CALL L51
32 REM LD BC.10
33 REM CALL L51
34 REM LD BC.1
35 REM CALL L51
36 REM POP BC
37 REM RET
38 REM *
39 REM :L51 SCF
40 REM CCF
41 REM LD A.0
42 REM :L53 SBC HL.BC
43 REM JP M.L52
44 REM INC A
45 REM JP L53
46 REM :L52 ADD A.$1C
47 REM RST 10
48 REM ADD HL.BC
49 REM RET
50 REM *
51 REM :L100 PUSH AF
52 REM LD A.$76
53 REM RST 10
54 REM POP AF
55 REM RET
56 REM )
57 REM
58 REM
59 REM
60 REM
61 REM
62 REM
63 REM
64 REM
```

Figure 13.2

13.4 MULTIPLE BYTE ARITHMETIC

There are two instructions which can be used for multiple byte arithmetic, in the same way that the ADC HL.ss and SBC HL.ss instructions could be used for multiple 16-bit arithmetic.

The equivalent 8-bit add instruction is:

ADC A.s

where s is either a value, a single 8-bit register, or a memory byte pointed to by HL, IX or IY. The instruction causes the value from s to be added to the accumulator, along with the value in the carry flag.

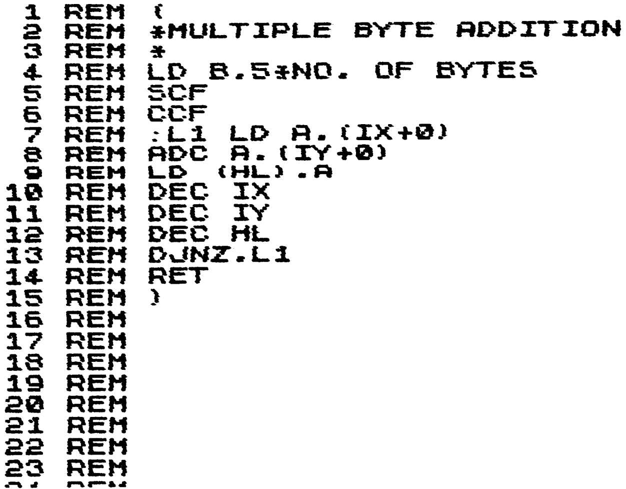
The equivalent subtract instruction is:

SBC A.s

where s is the same as for the ADC A.s instruction. This instruction causes the value from s and the value of the carry flag to be subtracted from the accumulator.

The principle of multibyte arithmetic is that the two least significant bytes, that is, the bytes on the right-hand side of the numbers, are added or subtracted using the ADD or SUB instructions and the remaining pairs of bytes, working from right to left, are added or subtracted using the ADC or SBC instructions.

Depending on the number of bytes to be added, there are many combinations of 8-bit and 16-bit arithmetic instructions which can be used. However for a general routine, for numbers with any number of bytes, an appropriate number of 8-bit arithmetic instructions is most suitable.



```
 1 REM {
 2 REM *MULTIPLE BYTE ADDITION
 3 REM *
 4 REM LD B.5*NO. OF BYTES
 5 REM SCF
 6 REM CCF
 7 REM :L1 LD A.(IX+0)
 8 REM ADC A.(IY+0)
 9 REM LD (HL).A
10 REM DEC IX
11 REM DEC IY
12 REM DEC HL
13 REM DJNZ.L1
14 REM RET
15 REM }
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
23 REM
```

Figure 13.3

Figure 13.3 shows a program segment, for a general routine, for the addition of multiple byte numbers. Initially, the register pairs IX, IY and HL point to the least significant bytes of the first number, the second number and the result, respectively, and the B register contains the number of bytes in each number. The numbers must be of equal length. The carry flag is initially set to zero, before the loop is entered, so that the first ADC instruction is equivalent to an ADD instruction.

EXERCISE 13.4

What modifications need to be made to the program segment in figure 13.3 to make it subtract the second number from the first number?

13.5 PROGRAM

Write a program which outputs to the screen the numbers from m to n in steps of k.

The numbers m, n and k are unsigned hexadecimal numbers which are input to the program from the keyboard.

The level of difficulty of this program may be varied by restricting the values of m, n and k to fit into:

8 bits (8-bit arithmetic)

16 bits (16-bit arithmetic)

24 bits (16-bit and 8-bit arithmetic)

32 bits (double 16-bit arithmetic)

n × 8 bits (multiple byte arithmetic)

Additionally, the program could be made to input and output in decimal, rather than hexadecimal, numbers.

14 Block transfer and search

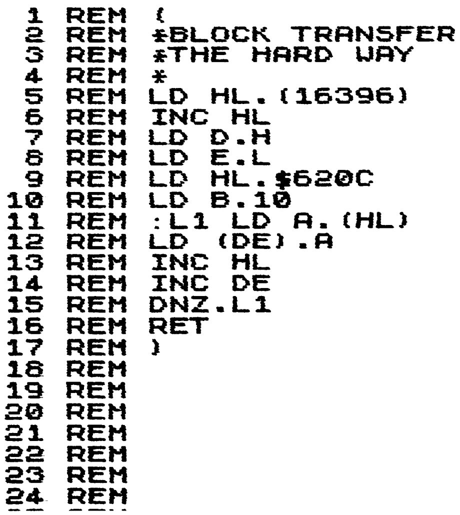
14.1 BLOCK INSTRUCTIONS

The microprocessor in the ZX81 has eight very powerful block instructions, which allow operations on blocks of consecutive memory bytes. Four of the instructions are block transfer instructions, which allow the contents of one block of memory bytes to be transferred to another block of memory bytes. One interesting use of these instructions is to enable a graphics display to be stored in memory and then transferred to the display file when it is to be displayed on the screen.

The other four instructions are block search instructions which allow a block of memory bytes to be searched for a specified value in one of the bytes.

14.2 BLOCK TRANSFER

Suppose you wish to move the contents of a block of ten memory bytes to the top line of the display; the program in figure 14.1 could be used. The program moves a block of ten memory bytes,



```
  1 REM {
  2 REM *BLOCK TRANSFER
  3 REM *THE HARD WAY
  4 REM *
  5 REM LD HL.(16396)
  6 REM INC HL
  7 REM LD D.H
  8 REM LD E.L
  9 REM LD HL.$620C
 10 REM LD B.10
 11 REM :L1 LD A.(HL)
 12 REM LD (DE).A
 13 REM INC HL
 14 REM INC DE
 15 REM DNZ.L1
 16 REM RET
 17 REM }
 18 REM
 19 REM
 20 REM
 21 REM
 22 REM
 23 REM
 24 REM
```

Figure 14.1

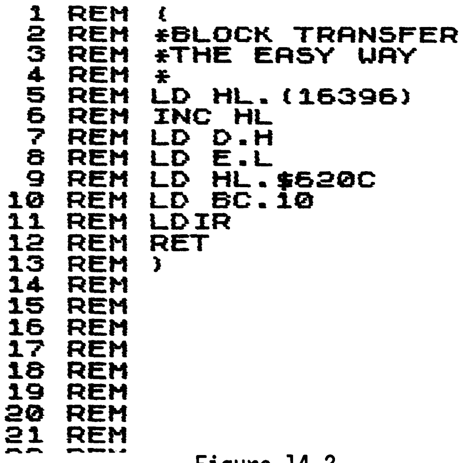
starting at address $620C, to the block of ten bytes at the start of the display file. The register pairs, HL and DE, are set to these two addresses and the B register is used to count the number of bytes moved. The B register is initially loaded with the value ten.

A loop is used to transfer a byte from one memory block to the other, then increment the values in the HL and DE register pairs to point to the next bytes in the original and new blocks of memory bytes. When the loop has finished a copy of the original block of memory bytes will have been moved to the new block of memory bytes. This will overwrite any data which may have been held in the new block of memory bytes.

EXERCISE 14.1

Using the BC register pair as a counter, what changes must be made to the program, in figure 14.1, to enable blocks of thousands of memory bytes to be moved instead of tens of memory bytes?

The microprocessor in the ZX81 has an instruction which can replace all the instructions in the loop in figure 14.1. This is the LDIR instruction, the name standing for Load, Increment and Repeat. Prior to execution of the LDIR instruction, the HL register pair must contain the address of the first of the memory bytes to be moved, the DE register pair must contain the address of the first memory byte to which they are to be moved, and the BC register pair must contain the number of bytes to be moved. Figure 14.2 shows a version of the program in figure 14.1 using the LDIR instruction. The program in figure 14.2 is functionally the same as the program in figure 14.1, except that blocks of up to 65535 bytes can be moved, since it uses a register pair to hold the value of the counter instead of a single 8-bit register. For each byte which is moved, register pairs HL and DE are incremented



```
 1 REM (
 2 REM *BLOCK TRANSFER
 3 REM *THE EASY WAY
 4 REM *
 5 REM LD HL.(16396)
 6 REM INC HL
 7 REM LD D.H
 8 REM LD E.L
 9 REM LD HL.$620C
10 REM LD BC.10
11 REM LDIR
12 REM RET
13 REM )
14 REM
15 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
```

Figure 14.2

by one, and the BC register pair is decremented by one; the move is repeated until the value in the BC register pair is zero.

The LDDR instruction, which stands for Load, Decrement and Repeat, is the same as the LDIR instruction except that the HL and DE register pairs are decremented by one instead of incremented. The value in the BC register pair is, of course, still decremented.

EXERCISE 14.2

Rewrite the program in figure 14.2 using the LDDR instruction instead of the LDIR instruction.

Two other block transfer instructions LDI and LDD are similar to the LDIR and LDDR except that they do not automatically test the value in the BC register pair and move to the next byte of the block.

The LDI instruction, which means Load and Increment, increments by one from the beginning of the block of bytes to be moved, whereas the LDD instruction decrements by one from the end of the block of bytes to be moved. The LDI instruction moves a byte from one block to the other and increments both the HL and DE register pairs by one, whereas the LDD instruction after moving a byte decrements the HL and DE register pairs by one. Both instructions decrement the BC register pair by one.

It is important to know that the BC register pair becoming zero is shown by the P/V, parity/overflow, flag not the zero flag. The P/V flag is set to zero; test for PO, if BC is zero, otherwise it is set to one.

EXERCISE 14.3

Rewrite the program in figure 14.2 using an LDI instruction instead of the LDIR instruction.

The LDIR and LDDR instructions can only be used when the number of bytes to be moved is known in advance. When the number of bytes to be moved is not known in advance, the LDI and LDD instructions must be used, and the program can contain tests for the end of the block.

EXERCISE 14.4

Write a program segment which moves a block of memory bytes from one place to another. A maximum of 400 bytes in the block should be allowed for, although the move should stop when the value in the next memory byte is zero. The memory byte containing the zero should not be moved.

Care must be taken, during block transfers, when the two blocks of memory bytes overlap. Take as an example the following program segment to move a block of memory bytes further up memory:

```
LD HL.20000
LD DE.20100
LD BC.500
LDIR
```

The first one hundred bytes of the original block of memory bytes will be copied into the first one hundred bytes of the new block of memory bytes, but this also happens to be the second one hundred bytes of the original block of memory bytes. So the last four hundred bytes of the original block are overwritten, and therefore lost, before they can be moved to the new block of memory bytes.

EXERCISE 14.5

What changes can be made to the program segment above, to move the block of bytes up memory correctly?

14.3 BLOCK SEARCH INSTRUCTIONS

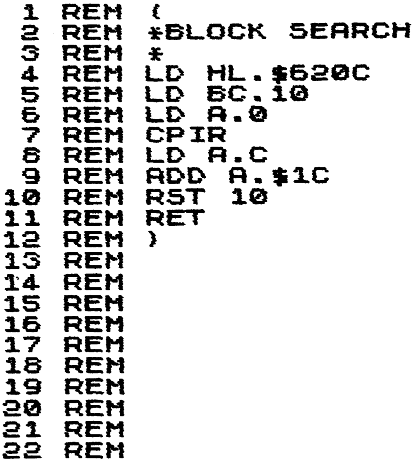
There are four block search instructions which allow a block of memory bytes to be searched to see if one of them contains the same value as the accumulator. The operation of all four instructions requires that the accumulator should contain the value for which the computer is searching, the HL register pair should contain the address of the first memory byte of the block of bytes to be searched, and the BC register pair should contain a count of the maximum number of memory bytes to be searched.

In a similar fashion to the block transfer instructions, two of the block search instructions include an automatic repeat to search through a block of memory bytes. The other two block search instructions only look at one memory byte, and require extra program instructions to move onto the next byte of the block.

The two automatic block search instructions are CPIR, which means Compare, Increment and Repeat, and CPDR, which means Compare, Decrement and Repeat. The program in figure 14.3 shows the use of the CPIR instruction to search for a memory byte, in a block of memory, containing the value zero.

First of all the register pair HL is loaded with the address of the first memory byte, the register pair BC is loaded with the number of memory bytes in the block, and the accumulator is loaded with the value to be found; in this example it is zero.

The CPIR instruction then searches through the block of memory bytes until either a match with the contents of the accumulator is



```
 1 REM (
 2 REM *BLOCK SEARCH
 3 REM *
 4 REM LD HL.$620C
 5 REM LD BC,10
 6 REM LD A.0
 7 REM CPIR
 8 REM LD A.C
 9 REM ADD A.$1C
10 REM RST 10
11 REM RET
12 REM )
13 REM
14 REM
15 REM
16 REM
17 REM
18 REM
19 REM
20 REM
21 REM
22 REM
```

Figure 14.3

found, or the end of the block is reached, that is, the value in the BC register pair becomes zero. For each byte, the value in the accumulator is compared with the value in the memory byte; if they are equal the zero flag is set to one, the HL register pair is incremented by one and the BC register pair is decremented by one. Finally, if the zero flag is set to one or the value in the BC register pair is zero, the instruction is finished; otherwise the next memory byte in the block is considered, and so on.

The CPDR instruction is used to search a block of memory bytes starting from the highest address and working back to the lowest address. In this case, the HL register pair is initially set to point to the address of the last byte of the block of memory bytes and during execution of the instruction the value in the HL register pair is decremented by one.

The CPI, which stands for Compare and Increment, and CPD, Compare and Decrement, instructions are similar to the CPIR and CPDR except that they do not automatically go on to the next memory byte. Extra instructions have to be used to test whether a match between the accumulator and the memory byte has been found and to detect if the whole of the block has been searched. If a match is found the zero flag is set and can be tested; the end of the block is reached when the BC register pair is zero and the P/V flag is set to zero. These two instructions are used in place of the CPIR and CPDR instructions whenever intermediate processing is required as, for example, when more than one occurrence of the value in the accumulator needs to be detected.

In the same way that the P/V flag is used to indicate that the value in the BC register pair is zero after the execution of the LDI and LDD block transfer instructions, the P/V flag is also used to indicate that the value in BC is zero after the block search instructions.

EXERCISE 14.6

What changes must be made to the program in figure 14.3 to make it output the value of the counter at every occurrence of zero?

It is sometimes useful to know the value in the HL and DE registers when the block instructions LDIR, LDDR, CPIR and CPDR have finished execution.

After the LDIR instruction, HL and DE will be pointing to the memory bytes immediately following the ends of the blocks and after the LDDR instruction, HL and DE will be pointing to the memory bytes immediately preceding the beginning of the blocks.

After the CPIR instruction, HL will be pointing to the memory byte immediately following the end of the block, and after the CPDR instruction, HL will be pointing to the memory byte immediately preceding the beginning of the block.

14.4 PROGRAM

This program is required to provide an internal filing system which can hold up to nine records. Each record contains 20 characters.

The file will consist of a block of 180 consecutive memory bytes. The program identifies each record by a record number, which is its numerical position in the file, so the file has records numbered 1 to 9.

A user of the program should be able to input any of the following:

D n — To delete the record numbered n, all the following records move up one record position.

I n 20-character-record — To insert the input record, after the record numbered n (n can be 0 to 8), any following records move down one record position.

R n 20-character-record — To replace the record number n by the input record.

L — To list the file on the display, for each record display record number and the contents of the record.

F character-string — To find and display the first record in the file containing the specified character-string, which may be 1, 2 or 3 characters.

15 Decimal arithmetic

15.1 BINARY CODED DECIMAL

Up to now we have only considered the binary representation of numbers, and arithmetic has involved signed and unsigned binary numbers. However, the microprocessor in the ZX81 also caters for another representation of numbers, called Binary Coded Decimal, or BCD for short.

The Binary Coded Decimal representation of numbers requires that each decimal digit be expressed as a 4-digit binary number, so that, for example, nine would be represented by the binary number 1001.

A 4-bit group is called a nibble, so a nibble is half a byte and a byte can hold two digits of a BCD number. Figure 15.1 shows a representation of a byte which holds a 2-digit BCD number. The left nibble, bits 4 to 7 inclusive, holds the binary representation of the number 7 and the right nibble, bits 0 to 3 inclusive, holds the binary representation of the number 4. The byte as a whole contains the BCD number 74.

0	1	1	1	0	1	0	0
7	6	5	4	3	2	1	0
Left nibble				Right nibble			

Figure 15.1 A two-digit BCD number

EXERCISE 15.1

Give the binary contents of a byte which holds the BCD number 57.

A nibble, when used for BCD numbers, contains the binary representation of a single decimal digit, that is, a number in the range 0 to 9, but when used to hold a pure unsigned binary number a nibble can be used to represent numbers in the range 0 to 15. So you can see that the BCD representation is wasteful compared to the unsigned binary representation, but, as will be explained, it does have other advantages.

EXERCISE 15.2

Compare the range of BCD and unsigned binary numbers which can be held in a byte.

Numbers are normally input from the keyboard and output to the display as decimal digits. To enable arithmetic to be performed on the numbers, using the instructions we have seen so far, they must first be converted to binary, and then the final results of the arithmetic must be converted back from binary to decimal digits, before being displayed on the screen.

When the computer has suitable instructions to allow arithmetic to be carried out with numbers in their decimal digit form, then the conversion from decimal to binary and vice versa are unnecessary. The ZX81 does have suitable instructions for decimal arithmetic, so sometimes it is more convenient to use the BCD representation of numbers rather than their binary representation. The instructions required are a Decimal Adjust instruction, to allow for the fact that each digit only uses one nibble and there can be a carry from one nibble to the next, and rotates to allow nibbles to be moved around.

15.2 BCD ARITHMETIC

As you would expect, BCD arithmetic is not so straightforward as binary arithmetic on a computer which is expecting binary arithmetic. The normal arithmetic instructions cannot be used because they are meant to add binary, and they are not suitable for BCD arithmetic. For example, if we take the BCD representation of 28 and the BCD representation of 39 and add them together using binary arithmetic and then interpret the result as a BCD number, we get the incorrect result of 61, as follows:

```
BCD 28 = 00101000
BCD 39 = 00111001
------------------
         01100001 = BCD 61
```

The error occurs because in BCD arithmetic if the sum of the right nibbles is greater than 9 then a carry into the left nibble is required. Binary arithmetic can also produce nibbles containing binary numbers in the range 1010B to 1111B, for which there is no valid BCD digit.

Having performed a binary add operation on two BCD numbers, it is possible to correct the erroneous result to give a correct BCD result. The correction, after the addition of two BCD digits, is as follows: if the nibble contains a value between 1010B and

1111B, or a carry occurs from the most significant bit of the nibble, then 0110B must be added to the nibble, otherwise nothing needs to be done. The following examples show the effect of the correction:

```
    0110      BCD 6
  + 0111      BCD 7
  ------
    1101      incorrect, not a valid BCD digit
  + 0110
--------
00010011      BCD 13 correct BCD result
    1000      BCD 8
  + 1001      BCD 9
--------
00010001      BCD 11, incorrect result
  + 0110
--------
00010111      BCD 17, correct result
```

EXERCISE 15.3

Show the addition of BCD 17 and BCD 69 as above.

A carry occurring from the left nibble of a two-digit BCD addition would indicate overflow, that is, a value greater than 99.

When subtracting BCD numbers, the same correction factor 0110B must be subtracted from the result if either a borrow occurs into the nibble or the nibble contains a value in the range 1010B to 1111B.

EXERCISE 15.4

Subtract BCD 56 from BCD 82.

A borrow occurring into the left nibble of a two-digit BCD subtraction would indicate overflow.

15.3 THE DAA INSTRUCTION

In order to provide decimal arithmetic, a computer can either provide a completely separate set of decimal arithmetic instructions, such as a decimal add, or it can provide a means of changing a binary arithmetic result into a BCD arithmetic result, using the methods shown in the previous section. The microprocessor in the ZX81 uses the latter method and includes the instruction DAA, which stands for Decimal Adjust the Accumulator.

Whenever an arithmetic instruction is executed, two flags, which we have not considered before, in the flag register, are affected. The two flags are the half carry and subtract flags. Although they are affected by every arithmetic instruction, they are only used by the DAA instruction. Neither of these flags may be used by a programmer, because there are no instructions to set or test them directly.

Using the setting of the half carry and subtract flags, the DAA instruction corrects the contents of the accumulator, if necessary, to give the correct result in the accumulator for BCD arithmetic. For example, the program segment

```
LD A.$43

LD B.$28

ADD A.B

DAA
```

loads the B register with the BCD representation of 28 and the accumulator with the BCD representation of 43. It then adds them together using the normal binary ADD instruction and then adjusts the result to BCD representation using the DAA instruction. Notice that for digits from 0 to 9, the hexadecimal representation of a number is the same as the BCD representation; this is very useful for writing BCD constant values in programs.

EXERCISE 15.5

What will be the contents of the accumulator in hexadecimal after the execution of each of the instructions in the above program segment?

The DAA instruction is used to give correct BCD arithmetic after the ADD A, SUB, INC A, DEC A, CP, NEG, ADC A, SBC and the four block search instructions. Notice that the DAA instruction only operates on the accumulator.

The two flags, of most direct importance to the programmer, which are affected by the DAA instruction are the carry flag, indicating BCD arithmetic overflow, and the zero flag, indicating a zero BCD value. The carry flag setting can also be used in multiple digit BCD arithmetic, as we shall see later.

15.4 THE DIGIT ROTATE INSTRUCTIONS

Two instructions are available to give a direct rotation of nibbles, or BCD digits; one to rotate to the left and the other to rotate to the right.

Both rotations involve the right-hand nibble of the accumulator and the two nibbles in a memory byte pointed to by the HL register pair. Figure 15.2 shows the operation of the two instructions.

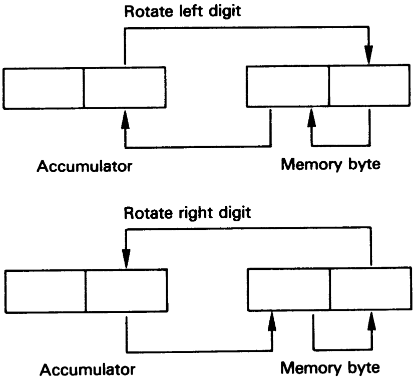
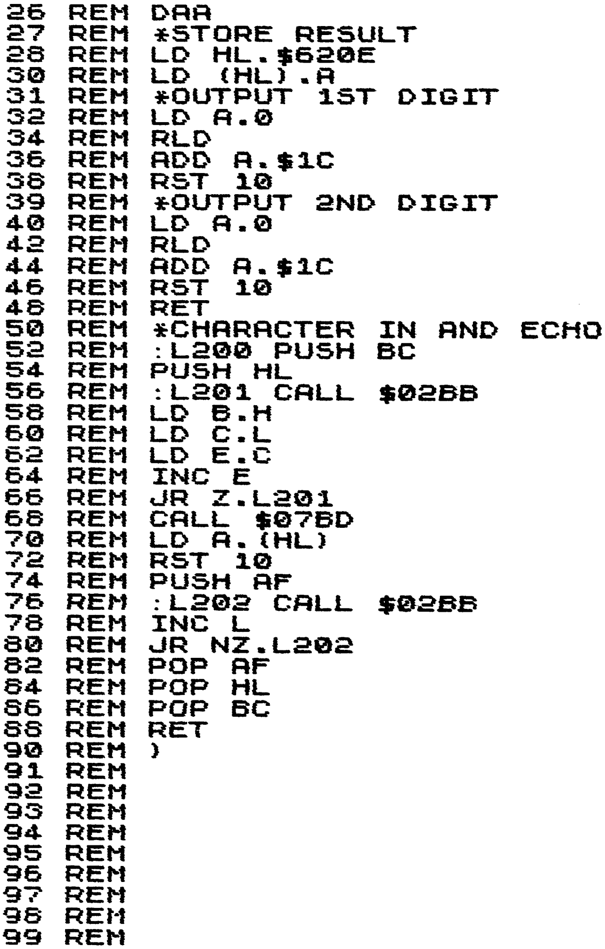


Figure 15.2

The Rotate Left Digit instruction, whose mnemonic is RLD, operates by moving the right-hand nibble of the accumulator into the right-hand nibble of the memory byte. The right-hand nibble of the memory byte is moved to the left-hand nibble of the memory byte, and the left-hand nibble of the memory byte is moved to the right-hand nibble of the accumulator. The left-hand nibble of the accumulator is unaffected by the rotation. The RRD instruction, which means Rotate Right Digit, works in a similar fashion, using the same nibbles, except that the rotation is in the opposite direction. This is illustrated in the figure.

The RLD and RRD instructions are very useful for manipulating BCD numbers. The program in figure 15.3 inputs two two-digit BCD numbers, adds them and outputs the result.

```
 1 REM (
 2 REM *BCD ADDITION
 3 REM *
 4 REM *INPUT FIRST NUMBER
 5 REM LD HL.$620C
 6 REM CALL L200
 7 REM SUB $1C
 8 REM LD (HL).A
 9 REM CALL L200
10 REM SUB $1C
11 REM RLD
12 REM *INPUT SECOND NUMBER
13 REM INC HL
14 REM CALL L200
15 REM SUB $1C
16 REM LD (HL).A
18 REM CALL L200
19 REM SUB $1C
20 REM RLD
21 REM *CARRY OUT ADDITION
22 REM LD HL.$620C
23 REM LD A.(HL)
24 REM INC HL
25 REM ADD A.(HL)
```



```
26 REM DAA
27 REM *STORE RESULT
28 REM LD HL.$620E
30 REM LD (HL).A
31 REM *OUTPUT 1ST DIGIT
32 REM LD A.0
34 REM RLD
36 REM ADD A.$1C
38 REM RST 10
39 REM *OUTPUT 2ND DIGIT
40 REM LD A.0
42 REM RLD
44 REM ADD A.$1C
46 REM RST 10
48 REM RET
50 REM *CHARACTER IN AND ECHO
52 REM :L200 PUSH BC
54 REM PUSH HL
56 REM :L201 CALL $02BB
58 REM LD B.H
60 REM LD C.L
62 REM LD E.C
64 REM INC E
66 REM JR Z.L201
68 REM CALL $07BD
70 REM LD A.(HL)
72 REM RST 10
74 REM PUSH AF
76 REM :L202 CALL $02BB
78 REM INC L
80 REM JR NZ.L202
82 REM POP AF
84 REM POP HL
86 REM POP BC
88 REM RET
90 REM )
91 REM
92 REM
93 REM
94 REM
95 REM
96 REM
97 REM
98 REM
99 REM
```

Figure 15.3

The only point that needs explanation in this program is the instruction that sets the accumulator to zero before the digits can be output. This is to ensure that the left nibble of the accumulator is zero; although the left nibble is not involved, you must ensure that the contents of the left nibble are zero before outputting the value in the accumulator.

15.5 PROGRAM

A program is required which inputs, adds, subtracts, and outputs multiple digit BCD numbers.

A BCD number is held in a number of successive memory bytes. The first byte specifies the sign of the number and the number of BCD digits in the number; bit 7 is zero for a positive number, or one for a negative number, and bits 0 to 6 hold the number of digits. Subsequent bytes contain the BCD digits packed two to a byte.

Write a subroutine which inputs a signed BCD number from the keyboard; no sign before the number implies a positive number. On entry to the subroutine the HL register pair points to the first byte to be used for storing the number.

Write a subroutine which outputs a signed BCD number to the display; on entry to the subroutine the HL register pair points to the first byte of the number to be output.

Write a subroutine which adds two signed BCD numbers and a similar subroutine which subtracts the second number from the first number.

Use the subroutines to write a main program which repeatedly inputs two signed BCD numbers separated by either a + character or a - character and followed by an = character and outputs the result. The program should cater for BCD numbers containing up to twenty BCD digits.

16 Miscellaneous instructions

16.1 OTHER INSTRUCTIONS

There are several instructions which have not yet been considered because they are only used rarely. For completeness, they are discussed briefly in this final chapter.

16.2 THE AUXILIARY REGISTERS

Inside the microprocessor in the ZX81 there is another set of eight registers called the auxiliary registers which are denoted by A', F', B', C', D', E', H' and L'. These auxiliary registers can be used in exactly the same way as the main registers, but not at the same time.

To change over from using the standard registers to using the auxiliary registers, the instruction EXX, which stands for Exchange Standard and Auxiliary Registers, is used. After this instruction has been executed subsequent instructions will refer to the registers which were the auxiliary registers. The effect of the instruction is to make the auxiliary registers become the standard registers. To revert back to using the original standard registers another EXX instruction must be executed.

Most programs need to use only the standard registers and it is not usually necessary to exchange all eight registers. A more frequent requirement is to be able to use the second accumulator. The instruction

```
EX AF.AF'
```

exchanges the standard accumulator and flag register with the auxiliary accumulator and flag register.

16.3 EXCHANGE INSTRUCTIONS

The EX instruction can also be used to exchange the values in some of the standard registers. When used for this purpose it is available in two forms:

```
EX DE.HL
```

In this form the instruction exchanges the values in the DE and HL register pairs. The other form of the instruction is:

```
EX (SP).rr
```

where rr is one of the register pairs HL, IX or IY. The effect of this instruction is to exchange the value in the register pair with the value in the pair of memory bytes pointed to by the SP register.

16.4 INPUT AND OUTPUT INSTRUCTIONS

There are twelve input and output instructions altogether and they are specified in Appendix E. On the ZX81 they are only used when inputting and outputting data using the expansion port on the back.

Input and output of data can be specified to be to, and from, any of the single 8-bit registers using the

```
IN r.(C)     and     OUT (C).r
```

instructions, in which case the value in the C register identifies the port to be used and r is the register used for input or output.

The remaining input and output instructions allow the input and output of blocks of data. These instructions are similar to the block search instructions except that, instead of comparing the value in a memory byte, the value is either input to or output from a memory byte.

These block input and output instructions appear very useful at first sight, but they are only of very limited usefulness since the repeat instructions can only be used with devices which operate at the same high speed as the instructions; this does not include keyboards, printers or most mechanical control devices.

16.5 INTERRUPT INSTRUCTIONS

An interrupt facility allows signals from outside to stop the sequence of instructions in the central processing unit. The interrupt instructions are given in Appendix E. The microprocessor in the ZX81 has three different modes of interrupt, which can be set by an

```
IM n
```

instruction where n is 0, 1 or 2. The other two instructions used with the interrupts are EI, which stands for enable interrupts and turns on the interrupt facility, and DI, which stands for disable interrupts and turns off the interrupt facility.

When an interrupt is received, and enabled, it causes the normal program flow to stop and jump to an interrupt service routine. The position of the service routine depends upon the interrupt mode in operation. The last instruction of an interrupt service routine is either a RETI instruction, which stands for return from interrupt, or a RETN instruction, which stands for return from non-maskable interrupt. The effect of both instructions is to return control to the program which was running before the interrupt.

16.6 THE HALT INSTRUCTION

This instruction is equivalent to the STOP instruction in BASIC. Its effect is to make the computer repeatedly execute NOP instructions until it receives an interrupt. Since an interrupt signal cannot be produced on the standard ZX81 computer, this instruction should not be used. The only means of regaining control of the ZX81 after the execution of a HALT instruction is to switch off the power.

Appendix A Number systems

A.1 OTHER NUMBER SYSTEMS

In order to use the computer with assembly language programs you must understand how information is stored in the registers and memory bytes. The numbers we use in everyday life are decimal numbers. However, all the information stored inside the computer is stored as binary numbers. Binary numbers are very awkward to use, so normally we use hexadecimal numbers, which are a compact way of representing binary, to represent the numbers stored in the computer.

Put simply, decimal is counting in tens, binary is counting in twos, and hexadecimal is counting in sixteens.

A.2 BINARY AND HEXADECIMAL NUMBERS

A decimal number, such as 453, may be broken down into its constituent parts as

$$453 = 4 \times 100 + 5 \times 10 + 3 \times 1$$

where 100 = 10 tens

Similarly, a hexadecimal number, such as 974, may be broken down into

$$\$974 = 9 \times 256 + 7 \times 16 + 4 \times 1$$

where 256 = 16 sixteens

and the binary number 101 is expressed as

$$101B = 1 \times 4 + 0 \times 2 + 1 \times 1$$

where 4 = 2 twos

The $ in front of the hexadecimal number is there to show that the number is hexadecimal and not decimal or binary.

Hexadecimal numbers can also be indicated by an H following the number, in the same way that a binary number is indicated by a B following the number.

Looking at the numbers above you can see that decimal numbers work in powers of ten, which means that each digit position in the number has a value of ten times the value of the digit position to its right. Hexadecimal numbers work in powers of sixteen, so that each digit position has a value sixteen times the value of the digit position to its right and binary numbers work in powers of two, so that each digit position has a value twice the value of the next digit position to its right.

Ten, sixteen and two are said to be the bases of the numbers. Decimal numbers have a base of ten, hexadecimal numbers have a base of sixteen, and binary numbers have a base of two. Any number can be used as a base, but nearly all computers work in binary, and hexadecimal is a compact way of showing the equivalent to a binary number; therefore base two and base sixteen are the most common number systems for computers.

EXERCISE A.1

By working out the expressions given above, what are the decimal numbers equivalent to $974 and 101B?

You already know that decimal numbers use the ten digits 0 to 9, that is, zero to one less than the value of the base. Numbers in any base use digits from zero to one less than the value of the base.

EXERCISE A.2

What digits are used for binary numbers?

For hexadecimal numbers we need sixteen digits, that is, 0 to something to represent the number 15. We can use the same digits as are used for decimal numbers up to 9, but we need some new symbols for the remaining six digits. The chosen symbols are the letters A, B, C, D, E and F, so that hexadecimal A is equivalent to decimal 10, and hexadecimal F is equivalent to decimal 15.

Figure A.1 shows the hexadecimal and binary equivalents of the decimal numbers 0 to 15.

A.3 BINARY-HEXADECIMAL CONVERSION

The main reason for using hexadecimal numbers is that conversion to and from binary is very easy. The basis of the conversion is that every hexadecimal digit can be replaced directly by a four-digit binary number and vice versa.

So to convert a hexadecimal number, say $6B, to binary each hexadecimal digit is replaced by its four-digit binary equivalent, as shown in figure A.1. Hence

$6B = 01101011B

Decimal	*Hexadecimal*	*Binary*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Figure A.1

To convert a binary number to hexadecimal, the binary number is separated into groups of four binary digits starting from the right; if necessary, extra zeros can be added on to the left to make the last group of four digits. For example, the binary number 1111100111 would become

```
0011  1110  0111
```

with two zeros added to make the last group into four digits. Each group is then converted to its equivalent hexadecimal digit, using figure A.1, so that the above binary number would become

```
 3     E     7
```

so that 1111100111B is equivalent to $3E7.

EXERCISE A.3

Convert $9AB3 to binary and 110011101111B to hexadecimal.

A.4 DECIMAL TO BINARY CONVERSION

To convert a decimal number to a binary number, repeatedly divide the decimal number by 2 until you can divide it no more. The remainders from the division make the equivalent binary number; the last remainder is the first digit of the number. Figure A.2 shows the method by converting 155 to binary.

Decimal to Binary Conversion

```
2 ) 155 ( 1
2 )  77 ( 1
2 )  38 ( 0
2 )  19 ( 1
2 )   9 ( 1
2 )   4 ( 0
2 )   2 ( 0
2 )   1 ( 1
      0
```

155 is equivalent to 10011011B

Figure A.2

A.5 BINARY TO DECIMAL CONVERSION

To convert a binary number to a decimal number it is best to expand the binary number in powers of two and add up the terms. For example, the conversion of 1001101B to decimal would be carried out as follows:

$$\begin{aligned} 1001101 &= 1 \times 64 + 0 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + \\ &\quad 0 \times 2 + 1 \times 1 \\ &= 64 + 8 + 4 + 1 \\ &= 77 \end{aligned}$$

the equivalent decimal number is 77.

A.6 DECIMAL-HEXADECIMAL CONVERSION

Conversions between decimal and hexadecimal numbers can be carried out in the same way as decimal and binary conversions, except that 16 is used instead of 2. For example, the conversion of 745 to hexadecimal is

```
16 ) 745
16 ) 46 r 9
16 ) 2 r 14(E)
     0 r 2
```

and the equivalent hexadecimal number is \$2E9, and the conversion of \$3AB2 to decimal becomes:

$$\begin{aligned}
\$3AB2 &= 3 \times 4096 + A \times 256 + B \times 16 + 2 \times 1 \\
&= 3 \times 4096 + 10 \times 256 + 11 \times 16 + 2 \times 1 \\
&= 12288 + 2560 + 176 + 2 \\
&= 15026
\end{aligned}$$

and the equivalent decimal number is 15026.

A.7 BINARY AND HEXADECIMAL ARITHMETIC

Addition and subtraction can be done using any base. The methods of working are exactly the same as for decimal numbers, except that instead of working in tens all working is done in terms of the base of the numbers being used. For example, when adding two hexadecimal numbers, a carry is only produced when the addition of two of the digits results in a number greater than \$F (or decimal 15).

Figure A.3 shows examples of addition and subtraction using hexadecimal and binary numbers.

```
  $3A7F          00110110B
+ $10BB        + 00011010B
  ----           --------

  $4B3A          01010000B


  $3A7F          00110110B
- $10BB        - 00011010B
  ----           --------

  $29C4          00011100B
```

Figure A.3

EXERCISE A.4

Do the following arithmetic:

```
  $C7BA          01101101B

- $9FF8        + 01011110B
  -----          ---------
```

A.8 BYTES

The basic unit of data in the Z80 microprocessor used in the ZX81 is a byte, which is another name for a binary number with eight binary digits (bits for short). It can also be represented by two hexadecimal digits. The value of a byte can be used to represent any one of several things, such as

a character

a positive whole number

or a signed whole number

The use of a byte to represent a character is dealt with in chapter 4.

If the value in the byte is considered simply as a number, this gives a representation for positive whole numbers only. For example, a byte containing 01100110B represents the number 1100110B which is $66 or 102. The range of numbers which can be contained in a byte is 0 to 11111111B ($FF or 255 decimal). This representation is often referred to as the unsigned number representation, to distinguish it from the representation discussed in the next section.

EXERCISE A.5

What range of unsigned numbers can be represented in two bytes (that is, 16 bits)?

A.9 SIGNED NUMBERS

Numbers which may have negative values as well as positive values are held in the ZX81 in what is called 'Two's complement form'. This form of representation depends upon all the numbers having the same number of digits; in the ZX81 this will normally be eight binary digits.

A negative number is represented by taking the two's complement of the equivalent positive value; this is done by converting all 0's to 1's and all 1's to 0's and then adding 1. For example:

```
   + 5  is  00000101B
so - 5  is  11111010B
                  + 1
            ---------
            11111011B
```

To convert back to a positive value from a negative number, the above process is repeated. This means that inside the computer, in a single 8-bit register or memory byte, the decimal number -5 will be stored as the binary number 11111011B.

When performing arithmetic with numbers using the two's complement system, the numbers are added together digit by digit as normal but any carry at the end is ignored. For example, adding + 5 and - 5 gives

```
           00000101        + 5
         + 11111011    +   - 5
           --------        ---
 carry(1)  00000000          0
```

The one carried out at the end of the addition is ignored and the result is given by the eight bits which are held in the register.

EXERCISE A.6

Calculate in binary using two's complement

```
(a)     - 60     (b)      - 23     (c)    85
      + + 70            + - 46          - 96
        ----              ----          ----
```

Two's Complement Numbers

127	01111111
126	01111110
	-
	-
2	00000000
1	00000001
0	00000000
- 1	11111111
- 2	11111110
	-
	-
	-
-127	10000001
-128	10000000

Values of the Bits in a Byte

Bit position	7	6	5	4	3	2	1	0
Unsigned numbers	128	64	32	16	8	4	2	1
Signed numbers	-128	64	32	16	8	4	2	1

Figure A.4

The range of numbers which can be held in a byte, using two's complement, is from -128 to +127. Figure A.4 shows the binary equivalent of some decimal numbers, and to help you in your understanding of signed and unsigned numbers the figure shows the value of each bit in a byte for both representations.

There are other ways which are used for representing negative numbers in computers, but the two's complement method is the most common and it is the one used by the ZX81. From now on instead of always referring to 'two's complement numbers' we shall refer to 'signed numbers'.

Appendix B Hand assembly

B.1 GENERAL METHOD

After a program has been written in assembly language it must be translated into a machine code program, before it can be run on a computer. The translation from assembly language program to machine code program is normally carried out by a special computer program, called an assembler. This book describes how to write assembly language programs for the Sinclair ZX81 computer and how to use the ZXAS program to translate them into machine code. It is not, however, essential to use an assembler program to carry out the translation process; it can be carried out by the programmer.

A program that has been written in assembly language can be translated into machine code using the tables given in Appendix E. These give the machine-code instructions in binary and it is possible to translate into binary numbers and then enter the binary numbers into the computer. However, when dealing with a list of binary numbers it is very easy to make errors; because of this it is normal to translate the machine code instructions into hexadecimal numbers and then enter the program into the computer as a list of hexadecimal numbers. For example, the instruction LD B.D when translated into binary becomes 01000010B, which is $42 when written in hexadecimal. Figure B.1 shows the input subroutine from chapter 5 translated into machine code in binary and hexadecimal numbers.

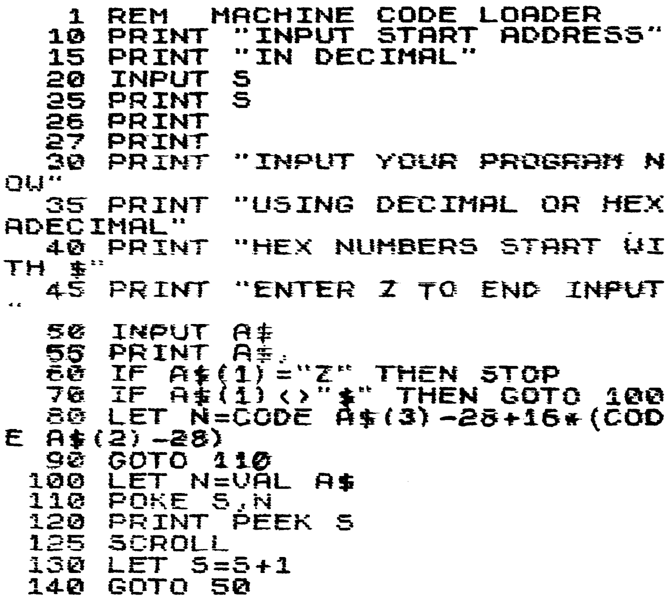
The final stage in hand assembly is to load the machine code version of the program into the computer, so that it can be run. This can be done using the facilities in the debugging program ZXDB, or by writing a simple memory loading program such as the one shown in figure B.2. This is a simple BASIC program which will load into memory a machine code program in the form of a list of hexadecimal numbers. The first line of the program is a REM statement which you should make long enough to take all of your machine code program. When the program is run the machine code program will be loaded into memory starting with the memory byte at address 16514.

B.2 NUMBERS AND ADDRESSES

All of the numbers in your machine code program must be in the same base that you are using for your instructions. For example, if you are loading your program into memory as a list of hexadecimal numbers all of the data in your program must also be given as hexadecimal numbers, so that the instruction LD A.12 becomes $3D, $0C, where $3D is the instruction code and $0C is the hexadecimal equivalent of the decimal number 12.

Assembly Language	*Machine Code*	
	Hex	Binary
:L200 PUSH BC	C5	11000101
PUSH HL	E5	11100101
:L201 CALL $02BB	CD	11001101
	BB	10111011
	02	00000010
LD B.H	44	01000100
LD C.L	4D	01001101
LD E.C	59	01011001
INC E	1C	00011100
JR Z.L201	28	00101000
	F7	11110111
CALL $07BD	CD	11001101
	BD	10111101
	07	00000111
LD A.(HL)	7D	01111101
RST 10	D7	11010111
PUSH AF	F5	11110101
:L202 CALL $02BB	CD	11001101
	BB	10111011
	02	00000010
INC L	2C	00101100
JR NZ.L202	20	00100000
	FA	11111010
POP AF	F1	11110001
POP HL	E1	11100001
POP BC	C1	11000001
RET	C9	11001001

Figure B.1



```
   1 REM  MACHINE CODE LOADER
  10 PRINT "INPUT START ADDRESS"
  15 PRINT "IN DECIMAL"
  20 INPUT S
  25 PRINT S
  26 PRINT
  27 PRINT
  30 PRINT "INPUT YOUR PROGRAM N
OW"
  35 PRINT "USING DECIMAL OR HEX
ADECIMAL"
  40 PRINT "HEX NUMBERS START WI
TH $"
  45 PRINT "ENTER Z TO END INPUT
"
  50 INPUT A$
  55 PRINT A$;
  60 IF A$(1)="Z" THEN STOP
  70 IF A$(1)<>"$" THEN GOTO 100
  80 LET N=CODE A$(3)-28+16*(COD
E A$(2)-28)
  90 GOTO 110
 100 LET N=VAL A$
 110 POKE S,N
 120 PRINT PEEK S
 125 SCROLL
 130 LET S=S+1
 140 GOTO 50
```

Figure B.2

Whenever the program uses a number which occupies two bytes, such as the address of a memory byte or a data value to be put into a register pair, the number must be written as a four-digit hexadecimal number and it will then be loaded into memory as two 2-digit numbers. Most importantly it must be loaded into memory with the two right-hand digits first, followed by the left-hand digits. For example, the instruction CALL $02BB must be entered into memory as $CD $BB $02.

B.3 JUMP INSTRUCTIONS

Jump instructions can be considered in two groups: absolute jumps which have the assembly language instruction JP, and relative jumps which have the assembly language instruction JR. Although conditional and unconditional jumps have different machine code instructions, the method of translating the instruction does not depend upon whether the instruction is conditional or unconditional.

Absolute jumps are always followed by the actual address of the memory byte which holds the instruction to be carried out after the jump has been made. Remember that this address will be held in two consecutive memory bytes, with the bytes in reverse order, as explained in the last section.

Relative jumps are more difficult to evaluate manually, and if a hand-assembled program fails to run it is often worth re-evaluating the relative jump instructions. The operand of a relative jump instruction is the number of bytes of program between the jump instruction and the instruction to be executed if the jump is made, minus two bytes. The relative jump works by adding the operand of the instruction to the value in the program counter, to give the address of the next instruction to be executed. Because the program counter always holds the address of the next instruction to be executed, when the jump instruction is executed the program counter points to the next instruction - this is why two bytes must be subtracted. For a jump forward, that is, to an instruction later in the program, the operand will be positive, and for a jump back to a previous instruction, the operand will be negative. A negative operand is entered as a two's complement number. Figure B.3 illustrates two relative jumps in assembly language and machine code. Note in figure B.3 that the operand for a jump backwards is the hexadecimal equivalent of the two's complement of the jump.

Assembly	*Machine Code*	*Comment*
:L1 LD A.(HL)	7E	
CP 0	FE	
	00	
JR Z.L2	28	Jump forward to label L2
	03	Jump is 3 bytes
INC HL	23	
JR L1	18	Jump back to label L1
	F8	Jump is -8 bytes
:L2 RET	C9	

Figure B.3

B.4 BIT INSTRUCTIONS

The machine code instructions for testing, setting and resetting depend upon both the register used and the particular bit of the register. In the machine code for these instructions there are three bits which are used to specify the register and another three bits which specify the bit position. Extra care needs to be taken when translating these assembly instructions into hexadecimal machine code. For example, the instruction RES 5.D translates to 11001011B, 10101010B in binary or $CB, $AA in hexadecimal.

B.5 INDEX REGISTER INSTRUCTIONS

Although index register instructions look more complex than most other instructions they can be greatly simplified. The instructions which operate on the value in the index register have exactly the same machine code as the equivalent instructions using the HL register pair, except that instructions for the IX index register are preceded by $DD and instructions for the IY index register are preceded by $FD.

When the index registers are used as pointers to data in memory bytes then not only must the equivalent HL instructions be preceded by $DD or $FD, depending upon which index register is used, but they must also be followed by a byte giving the displacement from the address in the index register. This displacement is added to the value in the index register to give the address of the memory byte which holds the data.

Appendix C Hexadecimal–decimal conversion tables

The table below provides for direct conversion between hexadecimal numbers in the range 0 to FF and decimal numbers in the range 0 to 255.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
10	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
20	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
30	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
40	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
50	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
60	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	161	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

For the conversion of larger numbers, use the following table in conjunction with the preceding table.

Hexadecimal	*Decimal*	*Hexadecimal*	*Decimal*
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

Appendix D Character codes

Only the normal keyboard characters are shown here. A full list of the Sinclair ZX81 character codes is given in the Sinclair manual; many of these may be used in your assembly language programs.

Character	*Code*		*Character*	*Code*	
	Decimal	*Hex*		*Decimal*	*Hex*
Space	0	00	1	29	1D
"	11	0B	2	30	1E
£	12	0C	3	31	1F
$	13	0D	4	32	20
:	14	0E	5	33	21
?	15	0F	6	34	22
(	16	10	7	35	23
)	17	11	8	36	24
>	18	12	9	37	25
<	19	13	A	38	26
=	20	14	B	39	27
+	21	15	C	40	28
-	22	16	D	41	29
*	23	17	E	42	2A
/	24	18	F	43	2B
;	25	19	G	44	2C
,	26	1A	H	45	2D
.	27	1B	I	46	2E
0	28	1C	J	47	2F
K	48	30	V	59	3B
L	49	31	W	60	3C
M	50	32	X	61	3D
N	51	33	Y	62	3E
O	52	34	Z	63	3F
P	53	35	cursor up	112	70
Q	54	36	cursor down	113	71
R	55	37	cursor left	114	72
S	56	38	cursor right	115	73
T	57	39	NEWLINE	118	76
U	58	3A			

Appendix E Summary of Z80 instructions

This appendix contains a summary of the complete Z80 instructions.

The first table, E.1, gives a summary of the operation of the flags.

In tables E.2 to E.12, the instructions are arranged in groups by function. Each table shows the assembly language instruction, a symbolic description of the instruction, the flags affected by the operation, the machine code instruction in binary, the number of bytes used by the instruction.

The following tables have been reproduced by permission of Zilog Inc. 1977. This material shall not be reproduced without the written consent of Zilog Inc.

TABLE E.1 SUMMARY OF FLAG OPERATION

Instruction	C	Z	P/V	S	N	H	Comments
ADD A, s; ADC A,s	↕	↕	V	↕	0	↕	8-bit add or add with carry
SUB s; SBC A, s, CP s, NEG	↕	↕	V	↕	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	↕	P	↕	0	1	Logical operations
OR s; XOR s	0	↕	P	↕	0	0	And set's different flags
INC s	●	↕	V	↕	0	↕	8-bit increment
DEC m	●	↕	V	↕	1	↕	8-bit decrement
ADD DD, ss	↕	●	●	●	0	X	16-bit add
ADC HL, ss	↕	↕	V	↕	0	X	16-bit add with carry
SBC HL, ss	↕	↕	V	↕	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	↕	●	●	●	0	0	Rotate accumulator
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	↕	↕	P	↕	0	0	Rotate and shift location m
RLD, RRD	●	↕	P	↕	0	0	Rotate digit left and right
DAA	↕	↕	P	↕	●	↕	Decimal adjust accumulator
CPL	●	●	●	●	1	1	Complement accumulator
SCF	1	●	●	●	0	0	Set carry
CCF	↕	●	●	●	0	X	Complement carry
IN r, (C)	●	↕	P	↕	0	0	Input register indirect .
INI; IND; OUTI; OUTD	●	↕	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	●	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	●	X	↕	X	0	0	Block transfer instructions
LDIR, LDDR	●	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	●	↕	↕	↕	1	X	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	●	↕	IFF	↕	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	●	↕	X	X	0	1	The state of bit b of location s is copied into the Z flag
NEG	↕	↕	V	↕	1	↕	Negate accumulator

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
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 into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	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.
↕	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>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

TABLE E.2 8-BIT LOAD GROUP

Mnemonic	Symbolic Operation	Flags						OP-Code	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments	
		C	Z	P/V	S	N	H	76 543 210					
LD r, r'	r ← r'	●	●	●	●	●	●	01 r r'	1	1	4	r, r'	Reg.
LD r, n	r ← n	●	●	●	●	●	●	00 r 110	2	2	7	000	B
								← n →				001	C
LD r, (HL)	r ← (HL)	●	●	●	●	●	●	01 r 110	1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	●	●	●	●	●	●	11 011 101	3	5	19	011	E
								01 r 110				100	H
								← d →				101	L
LD r, (IY+d)	r ← (IY+d)	●	●	●	●	●	●	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL), r	(HL) ← r	●	●	●	●	●	●	01 110 r	1	2	7		
LD (IX+d), r	(IX+d) ← r	●	●	●	●	●	●	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d), r	(IY+d) ← r	●	●	●	●	●	●	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL), n	(HL) ← n	●	●	●	●	●	●	00 110 110	2	3	10		
								← n →					
LD (IX+d), n	(IX+d) ← n	●	●	●	●	●	●	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD (IY+d), n	(IY+d) ← n	●	●	●	●	●	●	11 111 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD A, (BC)	A ← (BC)	●	●	●	●	●	●	00 001 010	1	2	7		
LD A, (DE)	A ← (DE)	●	●	●	●	●	●	00 011 010	1	2	7		
LD A, (nn)	A ← (nn)	●	●	●	●	●	●	00 111 010	3	4	13		
								← n →					
								← n →					
LD (BC), A	(BC) ← A	●	●	●	●	●	●	00 000 010	1	2	7		
LD (DE), A	(DE) ← A	●	●	●	●	●	●	00 010 010	1	2	7		
LD (nn), A	(nn) ← A	●	●	●	●	●	●	00 110 010	3	4	13		
								← n →					
								← n →					
LD A, I	A ← I	●	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 010 111					
LD A, R	A ← R	●	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 011 111					
LD I, A	I ← A	●	●	●	●	●	●	11 101 101	2	2	9		
								01 000 111					
LD R, A	R ← A	●	●	●	●	●	●	11 101 101	2	2	9		
								01 001 111					

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

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

TABLE E.3 16-BIT LOAD GROUP

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments	
		C	Z	P/V	S	N	H	76 543 210					
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001	3	3	10	dd	Pair
								← n →				00	BC
								← n →				01	DE
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101	4	4	14	10	HL
								00 100 001				11	SP
								← n →					
								← n →					
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101	4	4	14		
								00 100 001					
								← n →					
								← n →					
LD HL, (nn)	H ← (nn+1)	•	•	•	•	•	•	00 101 010	3	5	16		
	L ← (nn)							← n →					
								← n →					
LD dd, (nn)	dd_H ← (nn+1)	•	•	•	•	•	•	11 101 101	4	6	20		
	dd_L ← (nn)							01 dd1 011					
								← n →					
								← n →					
LD IX, (nn)	IX_H ← (nn+1)	•	•	•	•	•	•	11 011 101	4	6	20		
	IX_L ← (nn)							00 101 010					
								← n →					
								← n →					
LD IY, (nn)	IY_H ← (nn+1)	•	•	•	•	•	•	11 111 101	4	6	20		
	IY_L ← (nn)							00 101 010					
								← n →					
								← n →					
LD (nn), HL	(nn+1) ← H	•	•	•	•	•	•	00 100 010	3	5	16		
	(nn) ← L							← n →					
								← n →					
LD (nn), dd	(nn+1) ← dd_H	•	•	•	•	•	•	11 101 101	4	6	20		
	(nn) ← dd_L							01 dd0 011					
								← n →					
								← n →					
LD (nn), IX	(nn+1) ← IX_H	•	•	•	•	•	•	11 011 101	4	6	20		
	(nn) ← IX_L							00 100 010					
								← n →					
								← n →					
LD (nn), IY	(nn+1) ← IY_H	•	•	•	•	•	•	11 111 101	4	6	20		
	(nn) ← IY_L							00 100 010					
								← n →					
								← n →					
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6		
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101	2	2	10		
								11 111 001					
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101	2	2	10		
								11 111 001				qq	Pair
PUSH qq	(SP-2) ← qq_L	•	•	•	•	•	•	11 qq0 101	1	3	11	00	BC
	(SP-1) ← qq_H											01	DE
PUSH IX	(SP-2) ← IX_L	•	•	•	•	•	•	11 011 101	2	4	15	10	HL
	(SP-1) ← IX_H							11 100 101				11	AF
PUSH IY	(SP-2) ← IY_L	•	•	•	•	•	•	11 111 101	2	4	15		
	(SP-1) ← IY_H							11 100 101					
POP qq	qq_H ← (SP+1)	•	•	•	•	•	•	11 qq0 001	1	3	10		
	qq_L ← (SP)												
POP IX	IX_H ← (SP+1)	•	•	•	•	•	•	11 011 101	2	4	14		
	IX_L ← (SP)							11 100 001					
POP IY	IY_H ← (SP+1)	•	•	•	•	•	•	11 111 101	2	4	14		
	IY_L ← (SP)							11 100 001					

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

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

TABLE E.4 EXCHANGE GROUP AND BLOCK TRANSFER AND SEARCH GROUP

		Flags						Op-Code				
Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	$\begin{pmatrix}BC\\DE\\HL\end{pmatrix}\leftrightarrow\begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 100 000	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	•	•	0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC - 1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 110 001	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	•	② ↕	① ↕	↕	1	↕	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 111 001	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 Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

TABLE E.5 8-BIT ARITHMETIC AND LOGICAL GROUP

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
ADD A, r	A ← A + r	↕	↕	V	↕	0	↕	10 $\boxed{000}$ r	1	1	4	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
ADD A, n	A ← A + n	↕	↕	V	↕	0	↕	11 $\boxed{000}$ 110 ← n →	2	2	7	
ADD A, (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 $\boxed{000}$ 110	1	2	7	
ADD A, (IX+d)	A←A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101 10 $\boxed{000}$ 110 ← d →	3	5	19	
ADD A, (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101 10 $\boxed{000}$ 110 ← d →	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	V	↕	0	↕	$\boxed{001}$				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction
SUB s	A ← A − s	↕	↕	V	↕	1	↕	$\boxed{010}$				
SBC A, s	A ← A − s − CY	↕	↕	V	↕	1	↕	$\boxed{011}$				
AND s	A ← A ∧ s	0	↕	P	↕	0	1	$\boxed{100}$				
OR s	A ← A ∨ s	0	↕	P	↕	0	0	$\boxed{110}$				The indicated bits replace the 000 in the ADD set above.
XOR s	A ← A ● s	0	↕	P	↕	0	0	$\boxed{101}$				
CP s	A − s	↕	↕	V	↕	1	↕	$\boxed{111}$				
INC r	r ← r + 1	●	↕	V	↕	0	↕	00 r $\boxed{100}$	1	1	4	
INC (HL)	(HL) ← (HL)+1	●	↕	V	↕	0	↕	00 110 $\boxed{100}$	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	●	↕	V	↕	0	↕	11 011 101 00 110 $\boxed{100}$ ← d →	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d) + 1	●	↕	V	↕	0	↕	11 111 101 00 110 $\boxed{100}$ ← d →	3	6	23	
DEC m	m←m−1	●	↕	V	↕	1	↕	$\boxed{101}$				m is any of r, (HL), (IX+d), (IY+d) as shown for INC. Same format and states as INC. Replace 100 with 101 in OP code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

TABLE E.6 GENERAL PURPOSE ARITHMETIC AND CPU CONTROL GROUPS

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	↕	↕	P	↕	●	↕	00 100 111	1	1	4	Decimal adjust accumulator
CPL	A ← $\bar{A}$	●	●	●	●	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	A ← 0 − A	↕	↕	V	↕	1	↕	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	CY ← $\overline{CY}$	↕	●	●	●	0	X	00 111 111	1	1	4	Complement carry flag
SCF	CY ← 1	1	●	●	●	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	●	●	●	●	●	●	00 000 000	1	1	4	
HALT	CPU halted	●	●	●	●	●	●	01 110 110	1	1	4	
DI	IFF ← 0	●	●	●	●	●	●	11 110 011	1	1	4	
EI	IFF ← 1	●	●	●	●	●	●	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	●	●	●	●	●	●	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	●	●	●	●	●	●	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	●	●	●	●	●	●	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

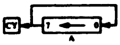
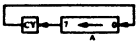
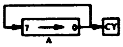
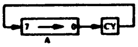
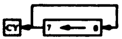
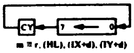
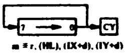
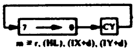
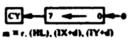
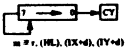
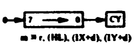
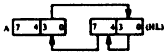
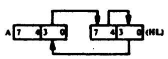
TABLE E.7 16-BIT ARITHMETIC GROUP

	Symbolic	Flags						Op-Code	No.	No.	No.		
Mnemonic	Operation	C	Z	P/V	S	N	H	76 543 210	of Bytes	of M Cycles	of T States	Comments	
ADD HL, ss	HL ← HL+ss	↕	•	•	•	0	X	00 ss1 001	1	3	11	ss	Reg.
												00	BC
ADC HL, ss	HL←HL+ss+CY	↕	↕	V	↕	0	X	11 101 101	2	4	15	01	DE
								01 ss1 010				10	HL
												11	SP
SBC HL, ss	HL←HL-ss-CY	↕	↕	V	↕	1	X	11 101 101	2	4	15		
								01 ss0 010					
ADD IX, pp	IX ← IX + pp	↕	•	•	•	0	X	11 011 101	2	4	15	pp	Reg.
								00 pp1 001				00	BC
												01	DE
												10	IX
												11	SP
ADD IY, rr	IY←IY+rr	↕	•	•	•	0	X	11 111 101	2	4	15	rr	Reg.
								00 rr1 001				00	BC
												01	DE
												10	IY
												11	SP
INC ss	ss ← ss + 1	•	•	•	•	•	•	00 ss0 011	1	1	6		
INC IX	IX ← IX + 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 100 011					
INC IY	IY ← IY + 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 100 011					
DEC ss	ss ← ss - 1	•	•	•	•	•	•	00 ss1 011	1	1	6		
DEC IX	IX ← IX - 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 101 011					
DEC IY	IY ← IY - 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 101 011					

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.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

TABLE E.8 ROTATE AND SHIFT GROUP

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
RLCA		↕	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		↕	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		↕	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		↕	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		↕	↕	P	↕	0	0	11 001 011 00 000 r	2	2	8	Rotate left circular register r
RLC (HL)		↕	↕	P	↕	0	0	11 001 011 00 000 110	2	4	15	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX+d)		↕	↕	P	↕	0	0	11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		↕	↕	P	↕	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	
RL m		↕	↕	P	↕	0	0	010				Instruction format and states are as shown for RLC,m. To form new OP-code replace 000 of RLC,m with shown code
RRC m		↕	↕	P	↕	0	0	001				
RR m		↕	↕	P	↕	0	0	011				
SLA m		↕	↕	P	↕	0	0	100				
SRA m		↕	↕	P	↕	0	0	101				
SRL m		↕	↕	P	↕	0	0	111				
RLD		•	↕	P	↕	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL).
RRD		•	↕	P	↕	0	0	11 101 101 01 100 111	2	5	18	The content of the upper half of the accumulator is unaffected

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

TABLE E. 9 BIT SET, RESET AND TEST GROUP

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
BIT b, r	Z ← $\overline{r}_b$	●	↕	X	X	0	1	11 001 011 01 b r	2	2	8	
BIT b, (HL)	Z ← $\overline{(HL)}_b$	●	↕	X	X	0	1	11 001 011 01 b 110	2	3	12	
BIT b, (IX+d)	Z ← $\overline{(IX+d)}_b$	●	↕	X	X	0	1	11 011 101 11 001 011 ← d → 01 b 110	4	5	20	
BIT b, (IY+d)	Z ← $\overline{(IY+d)}_b$	●	↕	X	X	0	1	11 111 101 11 001 011 ← d → 01 b 110	4	5	20	
SET b, r	r_b ← 1	●	●	●	●	●	●	11 001 011 $\boxed{11}$ b r	2	2	8	
SET b, (HL)	$(HL)_b$ ← 1	●	●	●	●	●	●	11 001 011 $\boxed{11}$ b 110	2	4	15	
SET b, (IX+d)	$(IX+d)_b$ ← 1	●	●	●	●	●	●	11 011 101 11 001 011 ← d → $\boxed{11}$ b 110	4	6	23	
SET b, (IY+d)	$(IY+d)_b$ ← 1	●	●	●	●	●	●	11 111 101 11 001 011 ← d → $\boxed{11}$ b 110	4	6	23	
RES b, m	s_b ← 0 m≡r, (HL), (IX+d), (IY+d)							$\boxed{10}$				To form new OP-code replace $\boxed{11}$ of SET b,m with $\boxed{10}$. Flags and time states for SET instruction

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

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

TABLE E. 10 JUMP GROUP

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
JP nn	PC ← nn	●	●	●	●	●	●	11 000 011 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ←nn, otherwise continue	●	●	●	●	●	●	11 cc 010 ← n → ← n →	3	3	10	
JR e	PC ← PC + e	●	●	●	●	●	●	00 011 000 ← e-2 →	2	3	12	
JR C, e	If C = 0, continue	●	●	●	●	●	●	00 111 000 ← e-2 →	2	2	7	If condition not met
	If C = 1, PC ← PC+e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	●	●	●	●	●	●	00 110 000 ← e-2 →	2	2	7	If condition not met
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	●	●	●	●	●	●	00 101 000 ← 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	●	●	●	●	●	●	00 100 000 ← e-2 →	2	2	7	If condition not mc
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	●	●	●	●	●	●	11 101 001	1	1	4	
JP (IX)	PC ← IX	●	●	●	●	●	●	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	●	●	●	●	●	●	11 111 101 11 101 001	2	2	8	
DJNZ,e	B ← B-1 If B = 0, continue	●	●	●	●	●	●	00 010 000 ← e-2 →	2	2	8	If B = 0
	If B ≠ 0, PC ← PC + e								2	3	13	IF B ≠ 0

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

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 op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

TABLE E.11 CALL AND RETURN GROUP

		Flags						Op-Code						
Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	(SP-1)←PC_H	●	●	●	●	●	●	11	001	101	3	5	17	
	(SP-2)←PC_L							←	n	→				
	PC←nn							←	n	→				
CALL cc, nn	If condition	●	●	●	●	●	●	11	cc	100	3	3	10	If cc is false
	cc is false							←	n	→				
	continue,													
	otherwise							←	n	→	3	5	17	If cc is true
	same as													
	CALL nn													
RET	PC_L←(SP)	●	●	●	●	●	●	11	001	001	1	3	10	
	PC_H←(SP+1)													
RET cc	If condition	●	●	●	●	●	●	11	cc	000	1	1	5	If cc is false
	cc is false													
	continue,													
	otherwise										1	3	11	If cc is true
	same as													
	RET													
RETI	Return from interrupt	●	●	●	●	●	●	11	101	101	2	4	14	
								01	001	101				
RETN	Return from non maskable interrupt	●	●	●	●	●	●	11	101	101	2	4	14	
								01	000	101				
RST p	(SP-1)←PC_H	●	●	●	●	●	●	11	t	111	1	3	11	
	(SP-2)←PC_L													
	PC_H←0													
	PC_L←P													

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

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
↕ = flag is affected according to the result of the operation.

TABLE E.12 INPUT AND OUTPUT GROUP

		Flags						Op-Code			
Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	•	•	•	•	•	•	11 011 011 ← n →	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	•	↕	P	↕	0	↕	11 101 101 01 r 000	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	1	X	11 101 101 10 100 010	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	1	X	11 101 101 10 101 010	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL -1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT (n), A	(n) ← A	•	•	•	•	•	•	11 010 011 ⟵ n ⟶	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101 01 r 001	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	① ↕	X	X	1	X	11 101 101 10 100 011	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	① ↕	X	X	1	X	11 101 101 10 101 011	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL -1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Appendix F Sample programs

F.1 THE PROGRAMS

This appendix contains two relatively simple, but complete programs. The programs are different in nature and the reasons why they are both programmed in assembly language are also different. These programs are intended to give a further insight into the use of assembly language programming.

The first program is the game of LIFE which was developed by John Conway of Cambridge University. The game simulates the growth and decay of colonies of organisms, called cells. A pattern for an initial colony is entered into the computer by the user. The colony then grows or decays according to the simple rules of the game. Each generation of the colony is displayed on the screen.

The next generation of the colony is produced by applying the rules of the game to each cell of the existing generation. This involves a large amount of processing and the main routines of the program were written in assembly language in order to give a reasonable speed of display.

The second program in this appendix is a very simple MAILING LIST program. While this program includes all the main features required by a mailing list program, it is relatively unsophisticated with little in the way of error-trapping routines.

The main problem with this type of program in BASIC on the ZX81 is that the size of the program only leaves a limited amount of space for the data file. The program was therefore written in assembly language so that the minimum amount of memory is used by the program. In practice the program when assembled only occupies 700 bytes.

F.2 LIFE

This is a very simple version of the game and it is played on a 24 by 15 grid of cells. Each cell can be either dead or alive; a dead cell is indicated by a space and a live cell is indicated by the letter 'O'.

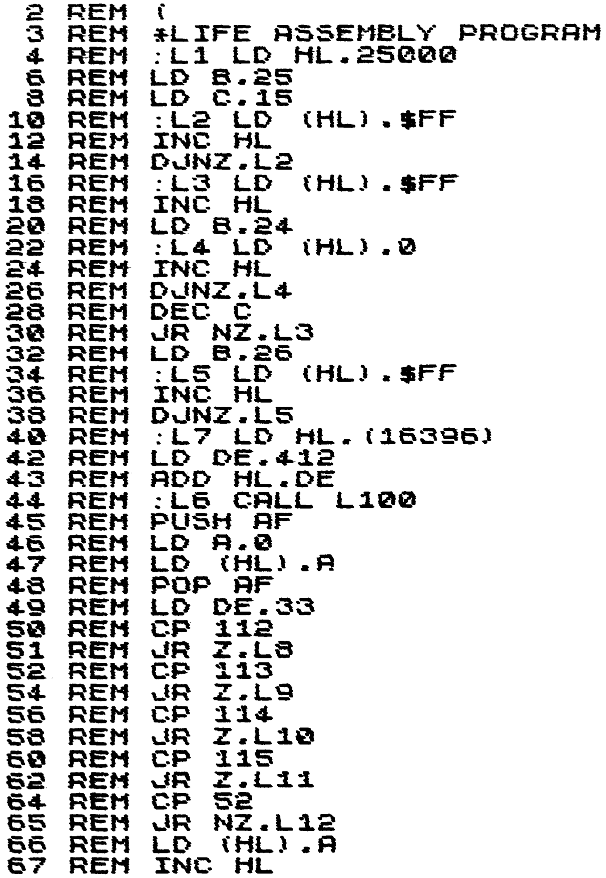
The rules for deciding whether a cell will be alive or dead in the next generation are:

(a) Any cell with exactly two live neighbours will remain the same in the next generation.

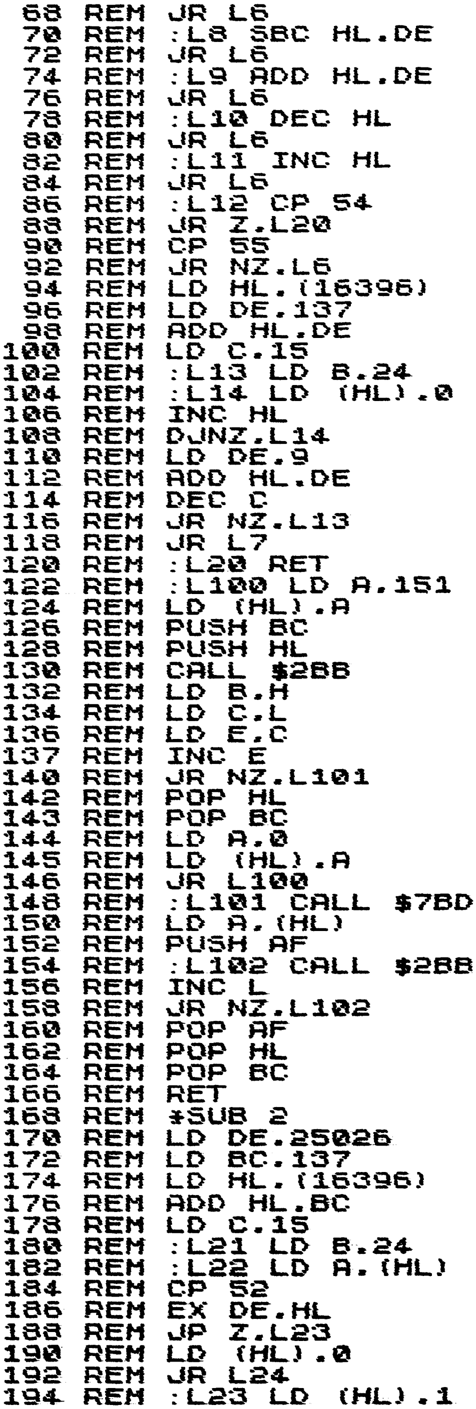
(b) Any cell, dead or alive, with exactly three live neighbours will be alive in the next generation.

(c) Any cell with more than three or less than two neighbours will be dead in the next generation.

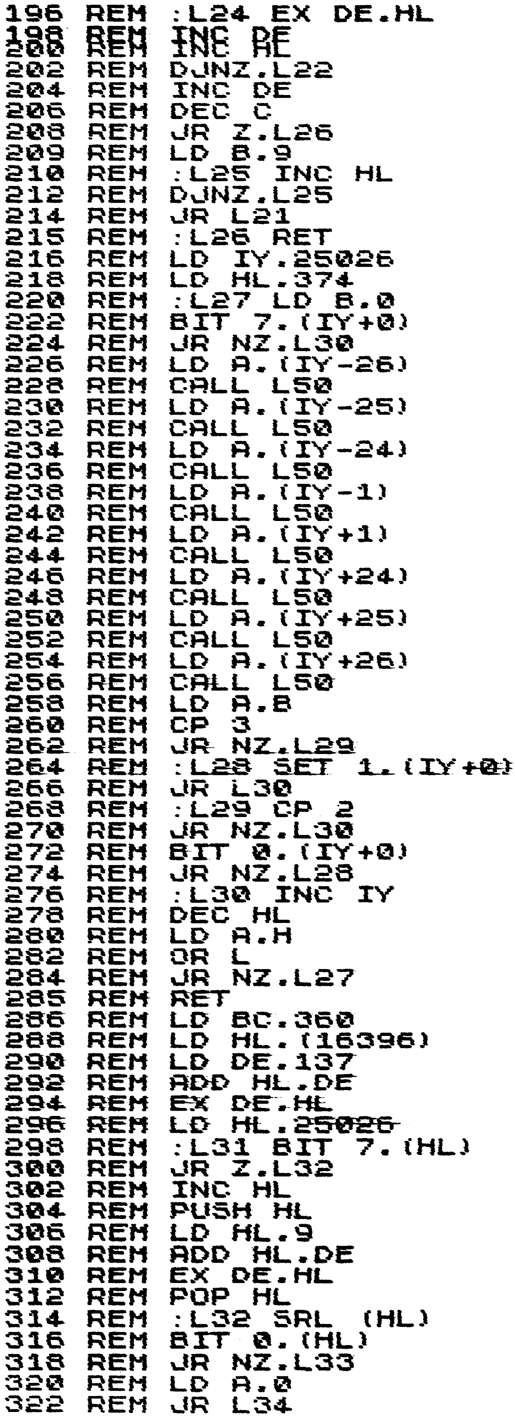
As you can see from the listing of the accompanying BASIC program, the program actually includes four separate assembly language subroutines. The first subroutine allows the player to set up the initial colony on the screen. The second subroutine copies the grid from the screen to memory for processing. The next subroutine carries out the processing and determines the pattern of the colony in the next generation. This routine is run in FAST mode so that one of the index registers may be used. The final subroutine displays the new generation on the screen. Figure F.1 is a listing of the assembly language subroutines and figure F.2 is the BASIC program.



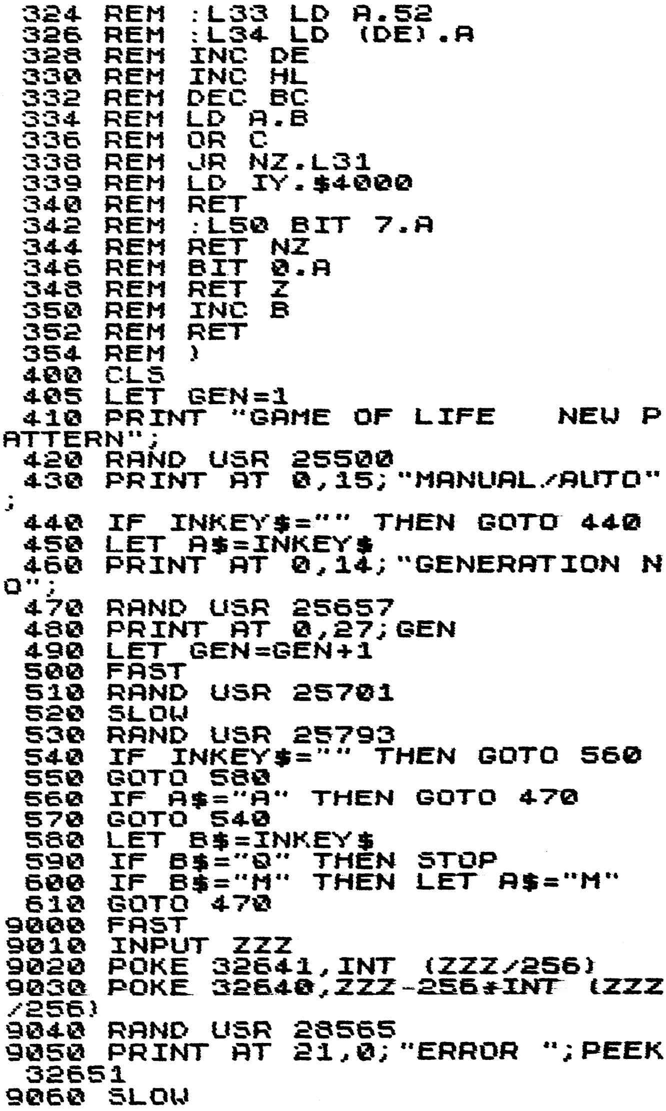
```
 2 REM (
 3 REM *LIFE ASSEMBLY PROGRAM
 4 REM :L1 LD HL.25000
 6 REM LD B.25
 8 REM LD C.15
10 REM :L2 LD (HL).$FF
12 REM INC HL
14 REM DJNZ.L2
16 REM :L3 LD (HL).$FF
18 REM INC HL
20 REM LD B.24
22 REM :L4 LD (HL).0
24 REM INC HL
26 REM DJNZ.L4
28 REM DEC C
30 REM JR NZ.L3
32 REM LD B.26
34 REM :L5 LD (HL).$FF
36 REM INC HL
38 REM DJNZ.L5
40 REM :L7 LD HL.(16396)
42 REM LD DE.412
43 REM ADD HL.DE
44 REM :L6 CALL L100
45 REM PUSH AF
46 REM LD A.0
47 REM LD (HL).A
48 REM POP AF
49 REM LD DE.33
50 REM CP 112
51 REM JR Z.L8
52 REM CP 113
54 REM JR Z.L9
56 REM CP 114
58 REM JR Z.L10
60 REM CP 115
62 REM JR Z.L11
64 REM CP 52
65 REM JR NZ.L12
66 REM LD (HL).A
67 REM INC HL
```



```
 68 REM JR L6
 70 REM :L8 SBC HL.DE
 72 REM JR L6
 74 REM :L9 ADD HL.DE
 76 REM JR L6
 78 REM :L10 DEC HL
 80 REM JR L6
 82 REM :L11 INC HL
 84 REM JR L6
 86 REM :L12 CP 54
 88 REM JR Z.L20
 90 REM CP 55
 92 REM JR NZ.L6
 94 REM LD HL.(16396)
 96 REM LD DE.137
 98 REM ADD HL.DE
100 REM LD C.15
102 REM :L13 LD B.24
104 REM :L14 LD (HL).0
106 REM INC HL
108 REM DJNZ.L14
110 REM LD DE.9
112 REM ADD HL.DE
114 REM DEC C
116 REM JR NZ.L13
118 REM JR L7
120 REM :L20 RET
122 REM :L100 LD A.151
124 REM LD (HL).A
126 REM PUSH BC
128 REM PUSH HL
130 REM CALL $2BB
132 REM LD B.H
134 REM LD C.L
136 REM LD E.C
137 REM INC E
140 REM JR NZ.L101
142 REM POP HL
143 REM POP BC
144 REM LD A,0
145 REM LD (HL).A
146 REM JR L100
148 REM :L101 CALL $7BD
150 REM LD A,(HL)
152 REM PUSH AF
154 REM :L102 CALL $2BB
156 REM INC L
158 REM JR NZ.L102
160 REM POP AF
162 REM POP HL
164 REM POP BC
166 REM RET
168 REM *SUB 2
170 REM LD DE.25026
172 REM LD BC.137
174 REM LD HL.(16396)
176 REM ADD HL.BC
178 REM LD C.15
180 REM :L21 LD B.24
182 REM :L22 LD A.(HL)
184 REM CP 52
186 REM EX DE.HL
188 REM JP Z.L23
190 REM LD (HL).0
192 REM JR L24
194 REM :L23 LD (HL).1
```

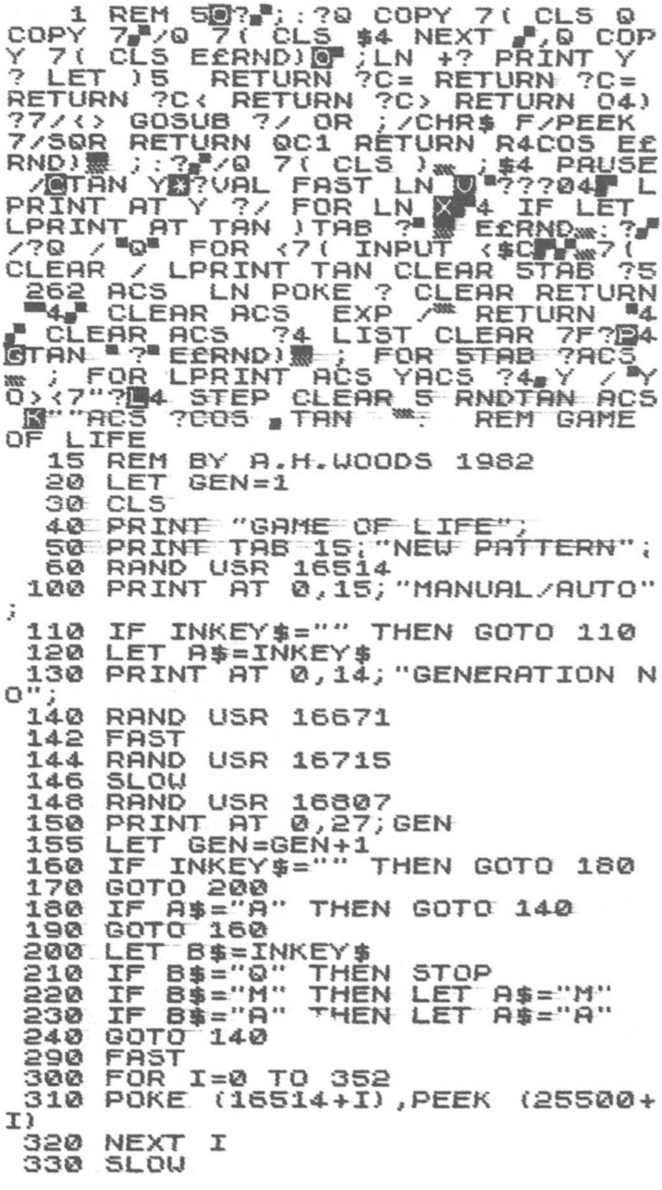


```
196 REM :L24 EX DE.HL
198 REM INC DE
200 REM INC HL
202 REM DJNZ.L22
204 REM INC DE
206 REM DEC C
208 REM JR Z.L26
209 REM LD B.9
210 REM :L25 INC HL
212 REM DJNZ.L25
214 REM JR L21
215 REM :L26 RET
216 REM LD IY.25026
218 REM LD HL.374
220 REM :L27 LD B.0
222 REM BIT 7.(IY+0)
224 REM JR NZ.L30
226 REM LD A.(IY-26)
228 REM CALL L50
230 REM LD A.(IY-25)
232 REM CALL L50
234 REM LD A.(IY-24)
236 REM CALL L50
238 REM LD A.(IY-1)
240 REM CALL L50
242 REM LD A.(IY+1)
244 REM CALL L50
246 REM LD A.(IY+24)
248 REM CALL L50
250 REM LD A.(IY+25)
252 REM CALL L50
254 REM LD A.(IY+26)
256 REM CALL L50
258 REM LD A.B
260 REM CP 3
262 REM JR NZ.L29
264 REM :L28 SET 1.(IY+0)
266 REM JR L30
268 REM :L29 CP 2
270 REM JR NZ.L30
272 REM BIT 0.(IY+0)
274 REM JR NZ.L28
276 REM :L30 INC IY
278 REM DEC HL
280 REM LD A.H
282 REM OR L
284 REM JR NZ.L27
285 REM RET
286 REM LD BC.360
288 REM LD HL.(16396)
290 REM LD DE.137
292 REM ADD HL.DE
294 REM EX DE.HL
296 REM LD HL.25026
298 REM :L31 BIT 7.(HL)
300 REM JR Z.L32
302 REM INC HL
304 REM PUSH HL
306 REM LD HL.9
308 REM ADD HL.DE
310 REM EX DE.HL
312 REM POP HL
314 REM :L32 SRL (HL)
316 REM BIT 0.(HL)
318 REM JR NZ.L33
320 REM LD A.0
322 REM JR L34
```



```
 324 REM :L33 LD A.52
 326 REM :L34 LD (DE).A
 328 REM INC DE
 330 REM INC HL
 332 REM DEC BC
 334 REM LD A.B
 336 REM OR C
 338 REM JR NZ.L31
 339 REM LD IY.$4000
 340 REM RET
 342 REM :L50 BIT 7.A
 344 REM RET NZ
 346 REM BIT 0.A
 348 REM RET Z
 350 REM INC B
 352 REM RET
 354 REM )
 400 CLS
 405 LET GEN=1
 410 PRINT "GAME OF LIFE   NEW P
ATTERN";
 420 RAND USR 25500
 430 PRINT AT 0,15;"MANUAL/AUTO"
;
 440 IF INKEY$="" THEN GOTO 440
 450 LET A$=INKEY$
 460 PRINT AT 0,14;"GENERATION N
O";
 470 RAND USR 25657
 480 PRINT AT 0,27;GEN
 490 LET GEN=GEN+1
 500 FAST
 510 RAND USR 25701
 520 SLOW
 530 RAND USR 25793
 540 IF INKEY$="" THEN GOTO 560
 550 GOTO 580
 560 IF A$="A" THEN GOTO 470
 570 GOTO 540
 580 LET B$=INKEY$
 590 IF B$="Q" THEN STOP
 600 IF B$="M" THEN LET A$="M"
 610 GOTO 470
9000 FAST
9010 INPUT ZZZ
9020 POKE 32641,INT (ZZZ/256)
9030 POKE 32640,ZZZ-256*INT (ZZZ
/256)
9040 RAND USR 28565
9050 PRINT AT 21,0;"ERROR ";PEEK
 32651
9060 SLOW
```

Figure F.1



```
   1 REM 5▣?▞; :?Q COPY 7( CLS Q
COPY 7▞/Q 7( CLS $4 NEXT ▞,Q COP
Y 7( CLS E£RND)▣▀ ;LN +? PRINT Y
? LET )5  RETURN ?C= RETURN ?C=
RETURN ?C< RETURN ?C> RETURN O4)
?7/<> GOSUB ?/ OR ;/CHR$ F/PEEK
7/SQR RETURN QC1 RETURN R4COS E£
RND)▒ ; :?▞/Q 7( CLS )▒ ;$4 PAUSE
 /CTAN Y*?VAL FAST LN U▀???04▛ L
PRINT AT Y ?/ FOR LN X▛4 IF LET
LPRINT AT TAN )TAB ?▀▒ E£RND▒:?▞
/?Q / ▀Q▀ FOR <7( INPUT <$C▛▞▒7(
CLEAR / LPRINT TAN CLEAR 5TAB ?5
 262 ACS  LN POKE ? CLEAR RETURN
▀4▞▀ CLEAR ACS  EXP /▒ RETURN ▀4
▞ CLEAR ACS  ?4 LIST CLEAR 7F?P4
GTAN ▀?▀E£RND)▒ ; FOR 5TAB ?ACS
▒ ; FOR LPRINT ACS YACS ?4▄Y / ▀Y
O><7"?L4 STEP CLEAR 5 RNDTAN ACS
 K""ACS ?COS ▄TAN ▒:  REM GAME
OF LIFE
  15 REM BY A.H.WOODS 1982
  20 LET GEN=1
  30 CLS
  40 PRINT "GAME OF LIFE";
  50 PRINT TAB 15;"NEW PATTERN";
  60 RAND USR 16514
 100 PRINT AT 0,15;"MANUAL/AUTO"
;
 110 IF INKEY$="" THEN GOTO 110
 120 LET A$=INKEY$
 130 PRINT AT 0,14;"GENERATION N
O";
 140 RAND USR 16671
 142 FAST
 144 RAND USR 16715
 146 SLOW
 148 RAND USR 16807
 150 PRINT AT 0,27;GEN
 155 LET GEN=GEN+1
 160 IF INKEY$="" THEN GOTO 180
 170 GOTO 200
 180 IF A$="A" THEN GOTO 140
 190 GOTO 160
 200 LET B$=INKEY$
 210 IF B$="Q" THEN STOP
 220 IF B$="M" THEN LET A$="M"
 230 IF B$="A" THEN LET A$="A"
 240 GOTO 140
 290 FAST
 300 FOR I=0 TO 352
 310 POKE (16514+I),PEEK (25500+
I)
 320 NEXT I
 330 SLOW
```

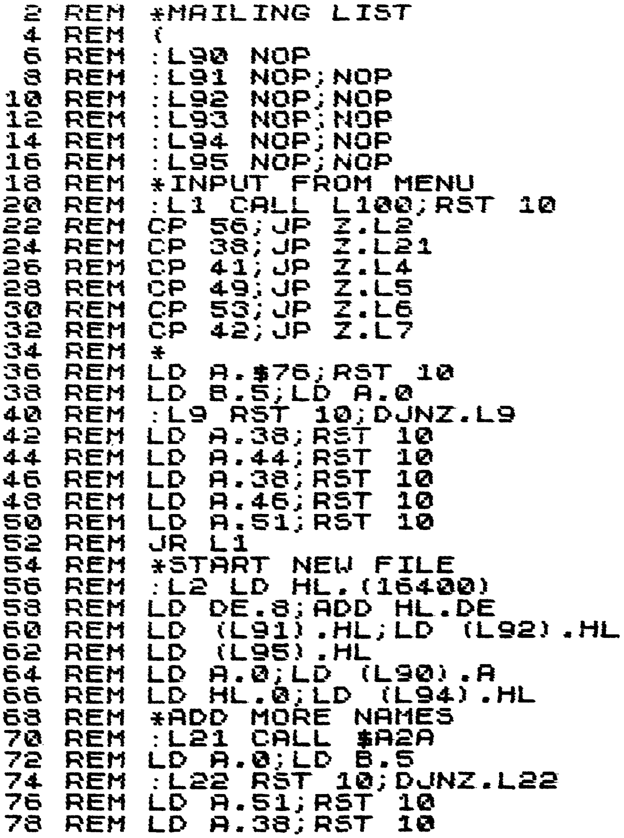
Figure F.2

The assembly language program was originally assembled into memory starting at location 25500 and it was then relocated into the memory starting at location 16514, which is the REM statement at the start of the program. The statements to carry out the relocation are included at the end of the BASIC program shown in figure F.2, starting at line 290.

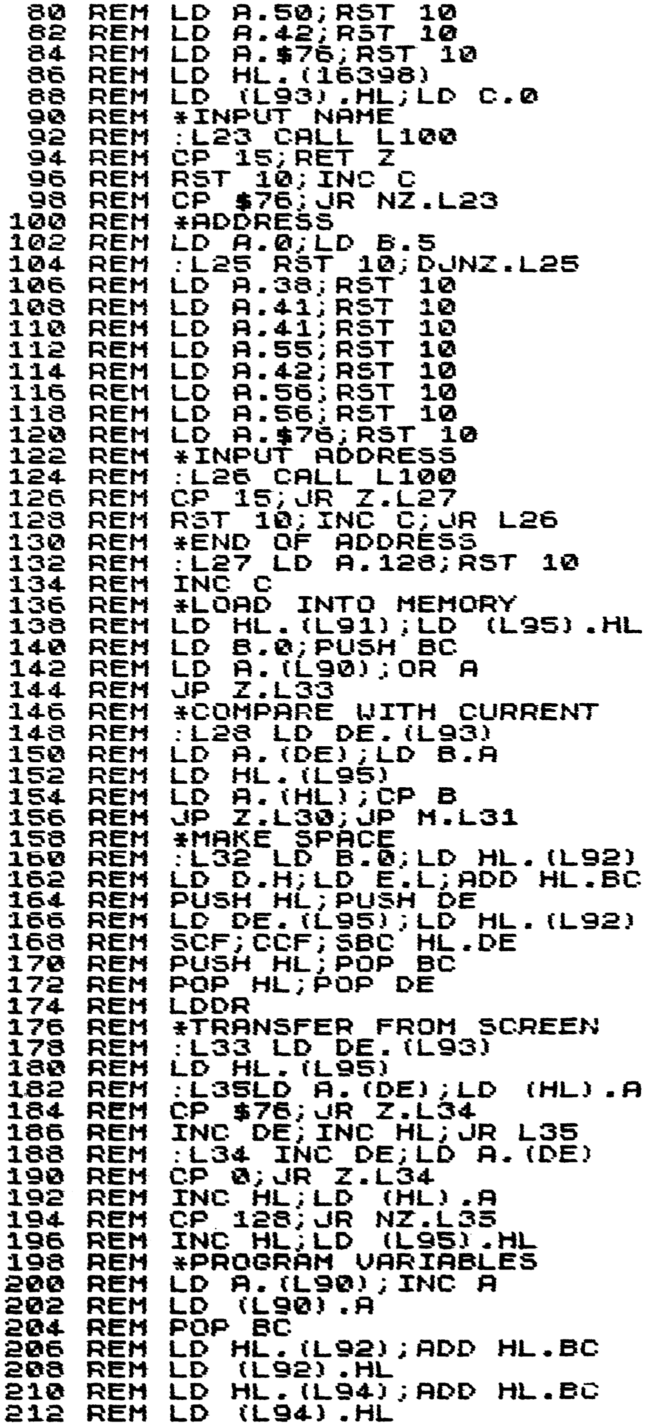
When the program is run the initial pattern is set up using the cursor keys to move a flashing cursor round the grid and a pattern of live cells is entered using the letter 'O' to indicate a live cell. When the pattern is complete, pressing the letter 'Q' will continue execution of the program. The program can be run in either automatic or manual mode, by entering either A or M. When the program is running the mode can be changed by entering A or M. To stop the program running press the letter 'Q'.

F.3 MAILING LIST

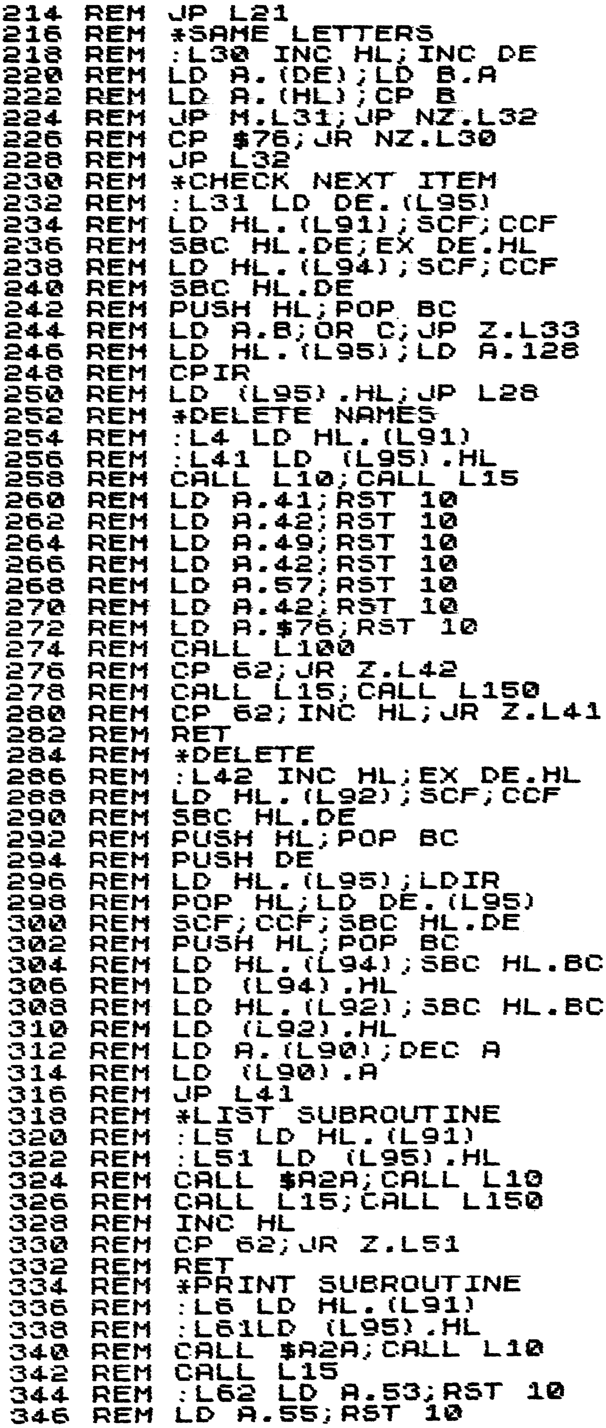
This is a simple program which shows how assembly language can save on memory used by the program and leave more space for information. In the form shown in figures F.3 and F.4, the program should be able to store approximately 80 names and addresses. The names and addresses are stored in alphabetic order in the VARS area of memory. Each name is inserted into its correct alphabetic order when it is entered.



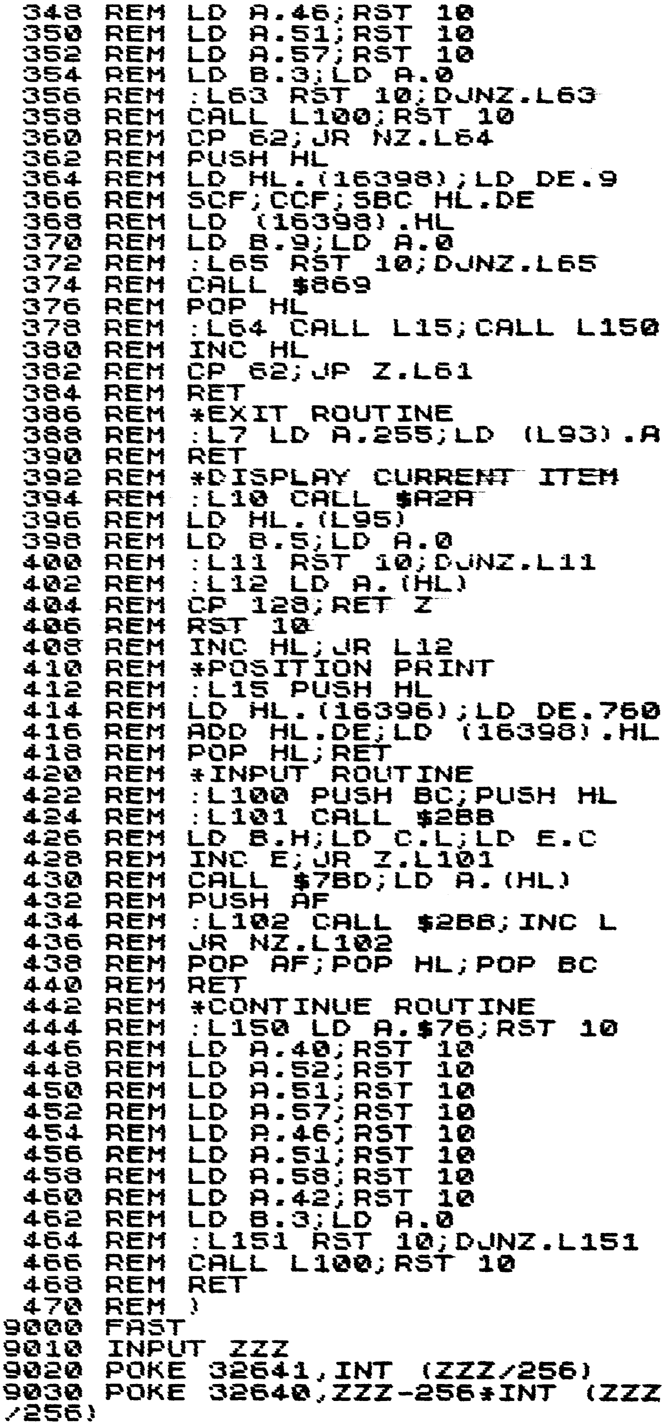
```
 2 REM *MAILING LIST
 4 REM (
 6 REM :L90 NOP
 8 REM :L91 NOP;NOP
10 REM :L92 NOP;NOP
12 REM :L93 NOP;NOP
14 REM :L94 NOP;NOP
16 REM :L95 NOP;NOP
18 REM *INPUT FROM MENU
20 REM :L1 CALL L100;RST 10
22 REM CP 56;JP Z.L2
24 REM CP 38;JP Z.L21
26 REM CP 41;JP Z.L4
28 REM CP 49;JP Z.L5
30 REM CP 53;JP Z.L6
32 REM CP 42;JP Z.L7
34 REM *
36 REM LD A.$76;RST 10
38 REM LD B.5;LD A.0
40 REM :L9 RST 10;DJNZ.L9
42 REM LD A.38;RST 10
44 REM LD A.44;RST 10
46 REM LD A.38;RST 10
48 REM LD A.46;RST 10
50 REM LD A.51;RST 10
52 REM JR L1
54 REM *START NEW FILE
56 REM :L2 LD HL.(16400)
58 REM LD DE.8;ADD HL.DE
60 REM LD (L91).HL;LD (L92).HL
62 REM LD (L95).HL
64 REM LD A.0;LD (L90).A
66 REM LD HL.0;LD (L94).HL
68 REM *ADD MORE NAMES
70 REM :L21 CALL $A2A
72 REM LD A.0;LD B.5
74 REM :L22 RST 10;DJNZ.L22
76 REM LD A.51;RST 10
78 REM LD A.38;RST 10
```



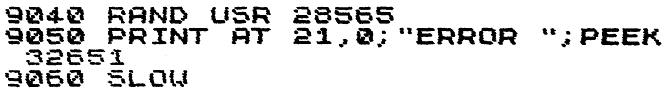
```
 80 REM LD A.50;RST 10
 82 REM LD A.42;RST 10
 84 REM LD A.$76;RST 10
 86 REM LD HL.(16398)
 88 REM LD (L93).HL;LD C.0
 90 REM *INPUT NAME
 92 REM :L23 CALL L100
 94 REM CP 15;RET Z
 96 REM RST 10;INC C
 98 REM CP $76;JR NZ.L23
100 REM *ADDRESS
102 REM LD A.0;LD B.5
104 REM :L25 RST 10;DJNZ.L25
106 REM LD A.38;RST 10
108 REM LD A.41;RST 10
110 REM LD A.41;RST 10
112 REM LD A.55;RST 10
114 REM LD A.42;RST 10
116 REM LD A.56;RST 10
118 REM LD A.56;RST 10
120 REM LD A.$76;RST 10
122 REM *INPUT ADDRESS
124 REM :L26 CALL L100
126 REM CP 15;JR Z.L27
128 REM RST 10;INC C;JR L26
130 REM *END OF ADDRESS
132 REM :L27 LD A.128;RST 10
134 REM INC C
136 REM *LOAD INTO MEMORY
138 REM LD HL.(L91);LD (L95).HL
140 REM LD B.0;PUSH BC
142 REM LD A.(L90);OR A
144 REM JP Z.L33
146 REM *COMPARE WITH CURRENT
148 REM :L28 LD DE.(L93)
150 REM LD A.(DE);LD B.A
152 REM LD HL.(L95)
154 REM LD A.(HL);CP B
156 REM JP Z.L30;JP M.L31
158 REM *MAKE SPACE
160 REM :L32 LD B.0;LD HL.(L92)
162 REM LD D.H;LD E.L;ADD HL.BC
164 REM PUSH HL;PUSH DE
166 REM LD DE.(L95);LD HL.(L92)
168 REM SCF;CCF;SBC HL.DE
170 REM PUSH HL;POP BC
172 REM POP HL;POP DE
174 REM LDDR
176 REM *TRANSFER FROM SCREEN
178 REM :L33 LD DE.(L93)
180 REM LD HL.(L95)
182 REM :L35LD A.(DE);LD (HL).A
184 REM CP $76;JR Z.L34
186 REM INC DE;INC HL;JR L35
188 REM :L34 INC DE;LD A.(DE)
190 REM CP 0;JR Z.L34
192 REM INC HL;LD (HL).A
194 REM CP 128;JR NZ.L35
196 REM INC HL;LD (L95).HL
198 REM *PROGRAM VARIABLES
200 REM LD A.(L90);INC A
202 REM LD (L90).A
204 REM POP BC
206 REM LD HL.(L92);ADD HL.BC
208 REM LD (L92).HL
210 REM LD HL.(L94);ADD HL.BC
212 REM LD (L94).HL
```



```
214 REM JP L21
216 REM *SAME LETTERS
218 REM :L30 INC HL;INC DE
220 REM LD A.(DE);LD B.A
222 REM LD A.(HL);CP B
224 REM JP M.L31;JP NZ.L32
226 REM CP $76;JR NZ.L30
228 REM JP L32
230 REM *CHECK NEXT ITEM
232 REM :L31 LD DE.(L95)
234 REM LD HL.(L91);SCF;CCF
236 REM SBC HL.DE;EX DE.HL
238 REM LD HL.(L94);SCF;CCF
240 REM SBC HL.DE
242 REM PUSH HL;POP BC
244 REM LD A.B;OR C;JP Z.L33
246 REM LD HL.(L95);LD A.128
248 REM CPIR
250 REM LD (L95).HL;JP L28
252 REM *DELETE NAMES
254 REM :L4 LD HL.(L91)
256 REM :L41 LD (L95).HL
258 REM CALL L10;CALL L15
260 REM LD A.41;RST 10
262 REM LD A.42;RST 10
264 REM LD A.49;RST 10
266 REM LD A.42;RST 10
268 REM LD A.57;RST 10
270 REM LD A.42;RST 10
272 REM LD A.$76;RST 10
274 REM CALL L100
276 REM CP 62;JR Z.L42
278 REM CALL L15;CALL L150
280 REM CP 62;INC HL;JR Z.L41
282 REM RET
284 REM *DELETE
286 REM :L42 INC HL;EX DE.HL
288 REM LD HL.(L92);SCF;CCF
290 REM SBC HL.DE
292 REM PUSH HL;POP BC
294 REM PUSH DE
296 REM LD HL.(L95);LDIR
298 REM POP HL;LD DE.(L95)
300 REM SCF;CCF;SBC HL.DE
302 REM PUSH HL;POP BC
304 REM LD HL.(L94);SBC HL.BC
306 REM LD (L94).HL
308 REM LD HL.(L92);SBC HL.BC
310 REM LD (L92).HL
312 REM LD A.(L90);DEC A
314 REM LD (L90).A
316 REM JP L41
318 REM *LIST SUBROUTINE
320 REM :L5 LD HL.(L91)
322 REM :L51 LD (L95).HL
324 REM CALL $A2A;CALL L10
326 REM CALL L15;CALL L150
328 REM INC HL
330 REM CP 62;JR Z.L51
332 REM RET
334 REM *PRINT SUBROUTINE
336 REM :L6 LD HL.(L91)
338 REM :L61LD (L95).HL
340 REM CALL $A2A;CALL L10
342 REM CALL L15
344 REM :L62 LD A.53;RST 10
346 REM LD A.55;RST 10
```

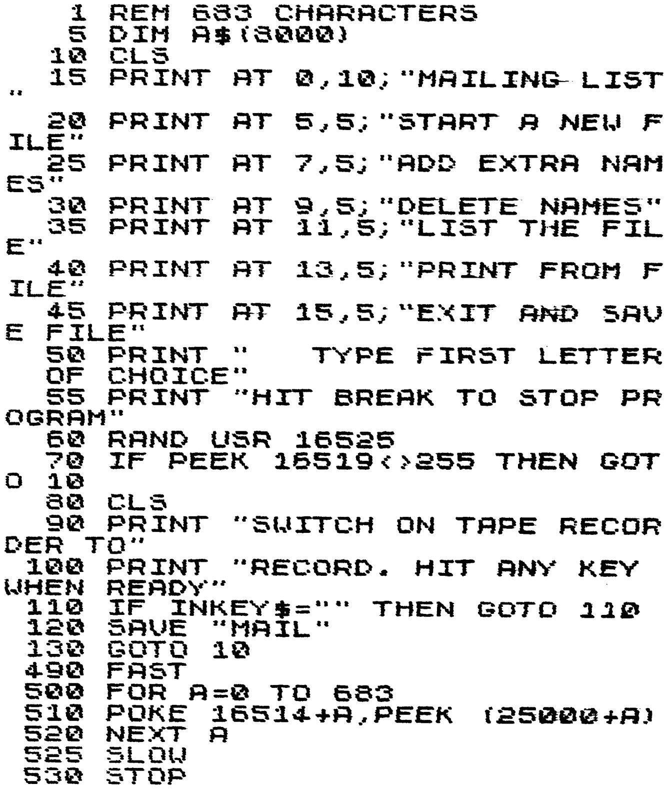


```
 348 REM LD A.46;RST 10
 350 REM LD A.51;RST 10
 352 REM LD A.57;RST 10
 354 REM LD B.3;LD A.0
 356 REM :L63 RST 10;DJNZ.L63
 358 REM CALL L100;RST 10
 360 REM CP 62;JR NZ.L64
 362 REM PUSH HL
 364 REM LD HL.(16398);LD DE.9
 366 REM SCF;CCF;SBC HL.DE
 368 REM LD (16398).HL
 370 REM LD B.9;LD A.0
 372 REM :L65 RST 10;DJNZ.L65
 374 REM CALL $869
 376 REM POP HL
 378 REM :L64 CALL L15;CALL L150
 380 REM INC HL
 382 REM CP 62;JP Z.L61
 384 REM RET
 386 REM *EXIT ROUTINE
 388 REM :L7 LD A.255;LD (L93).A
 390 REM RET
 392 REM *DISPLAY CURRENT ITEM
 394 REM :L10 CALL $A2A
 396 REM LD HL.(L95)
 398 REM LD B.5;LD A.0
 400 REM :L11 RST 10;DJNZ.L11
 402 REM :L12 LD A.(HL)
 404 REM CP 128;RET Z
 406 REM RST 10
 408 REM INC HL;JR L12
 410 REM *POSITION PRINT
 412 REM :L15 PUSH HL
 414 REM LD HL.(16396);LD DE.760
 416 REM ADD HL.DE;LD (16398).HL
 418 REM POP HL;RET
 420 REM *INPUT ROUTINE
 422 REM :L100 PUSH BC;PUSH HL
 424 REM :L101 CALL $2BB
 426 REM LD B.H;LD C.L;LD E.C
 428 REM INC E;JR Z.L101
 430 REM CALL $7BD;LD A.(HL)
 432 REM PUSH AF
 434 REM :L102 CALL $2BB;INC L
 436 REM JR NZ.L102
 438 REM POP AF;POP HL;POP BC
 440 REM RET
 442 REM *CONTINUE ROUTINE
 444 REM :L150 LD A.$76;RST 10
 446 REM LD A.40;RST 10
 448 REM LD A.52;RST 10
 450 REM LD A.51;RST 10
 452 REM LD A.57;RST 10
 454 REM LD A.46;RST 10
 456 REM LD A.51;RST 10
 458 REM LD A.58;RST 10
 460 REM LD A.42;RST 10
 462 REM LD B.3;LD A.0
 464 REM :L151 RST 10;DJNZ.L151
 466 REM CALL L100;RST 10
 468 REM RET
 470 REM )
9000 FAST
9010 INPUT ZZZ
9020 POKE 32641,INT (ZZZ/256)
9030 POKE 32640,ZZZ-256*INT (ZZZ
/256)
```



```
9040 RAND USR 28565
9050 PRINT AT 21,0;"ERROR ";PEEK
 32651
9060 SLOW
```

Figure F.3



```
   1 REM 683 CHARACTERS
   5 DIM A$(8000)
  10 CLS
  15 PRINT AT 0,10;"MAILING LIST
"
  20 PRINT AT 5,5;"START A NEW F
ILE"
  25 PRINT AT 7,5;"ADD EXTRA NAM
ES"
  30 PRINT AT 9,5;"DELETE NAMES"
  35 PRINT AT 11,5;"LIST THE FIL
E"
  40 PRINT AT 13,5;"PRINT FROM F
ILE"
  45 PRINT AT 15,5;"EXIT AND SAV
E FILE"
  50 PRINT "   TYPE FIRST LETTER
  OF CHOICE"
  55 PRINT "HIT BREAK TO STOP PR
OGRAM"
  60 RAND USR 16525
  70 IF PEEK 16519<>255 THEN GOT
O 10
  80 CLS
  90 PRINT "SWITCH ON TAPE RECOR
DER TO"
 100 PRINT "RECORD. HIT ANY KEY
WHEN READY"
 110 IF INKEY$="" THEN GOTO 110
 120 SAVE "MAIL"
 130 GOTO 10
 490 FAST
 500 FOR A=0 TO 683
 510 POKE 16514+A,PEEK (25000+A)
 520 NEXT A
 525 SLOW
 530 STOP
```

Figure F.4

When the program is run for the first time it can be started normally using the RUN command. Subsequently it must be started by GOTO 5. If RUN is used the variable area is cleared and all the saved names and addresses will be lost from memory. Of course, since the file is saved by using the SAVE command in the program, the program will start automatically when loaded from tape.

Apart from the main menu the majority of the program consists of a machine-code subroutine. Figure F.3 is a listing of the assembly language program for this subroutine and figure F.4 is the BASIC program.

The facilities offered by the program are:

START A NEW FILE

ADD EXTRA NAMES

DELETE NAMES

LIST THE FILE ON THE DISPLAY

PRINT SELECTED ITEMS ON A PRINTER

SAVE THE PROGRAM AND FILE ON TAPE

When the program is run a menu is displayed and a choice is made by entering the first letter of the choice. If either of the first two menu items is chosen, the computer prompts for the name first, entering NEWLINE indicates end of name and produces the address prompt. The address is entered in the normal way using as many lines as are required. When the address has been entered press the '?' key to enter the name and address to the file. After the name and address have been written to the file the program returns to the name prompt. To end the input press the '?' key after the name prompt. This will return you to the main menu. All other prompts in the program require a simple Y(es) or N(o) reply.

The program is very unsophisticated and it would be a good exercise for you to modify the program to suit your own requirements and to make the program easy to use.

Answers to exercises

The answers are in exercise number, then chapter number order. This allows you to read the answer to an exercise without seeing the answer to the next question.

ANSWERS TO EXERCISES NUMBERED 1

1.1 Accumulator used for subtraction.

Flag register indicates result.

3.1 Because the number is converted into an eight digit binary number.

4.1
```
:L1 LD A.0
ADD A.B
ADD A.C
ADD A.D
ADD A.E
```

5.1
```
LD A.$17
:L1 RST 10
CALL $F43
JP L1
```

6.1

Accumulator	S	Z
120 ($78)	?	?
-2 ($FE)	1	0
-2 ($FE)	1	0
0 ($00)	0	1
70 ($46)	0	0
-70 ($BA)	1	0

7.1 255

8.1

LD A.$17	4 times
:L2 RST10	24 times
DEC C	4 times

9.1 It produces a carry and the correct result. The carry can be ignored.

10.1 $53 in the accumulator and 1 in the carry flag.

11.1 Bit 1 of A is zero therefore the zero flag is set to 1.

12.1

	Accumulator	Carry
	10101011	0
RLA	01010110	1
RLCA	10101100	0
RRA	01010110	0
RRCA	00101011	0

13.1

Unsigned	0 to 65535
Signed	-32768 to +32767

14.1 Replace LD B.10 by LD BC. number of bytes and replace DJNZ.L1 by the following program segment:

```
DEC BC
LD A.B
CP 0
JP NZ.L1
LD A.C
CP 0
JP NZ.L1
```

15.1 01010111B

A.1 $974 is equivalent to 2420

101B is equivalent to 5

ANSWERS TO EXERCISES NUMBERED 2

1.2 Decimal 16384 Hexadecimal 4000

3.2

```
LD A.73
SUB 21
ADD A.55
```

4.2 $2C and +

5.2 A JP instruction uses 3 bytes and a JR instruction uses 2 bytes.

6.2

```
LD A.($620C)
SUB 10
JP Z.L1
LD A.0
ADD A.$1C
RST 10
RET
```

7.2

```
CALL L200 * INPUT SUBROUTINE
SUB $1C
LD B.A
:L1 LD A.$17
RST 10
DJNZ.L1
```

8.2 (i) Implied - the value in the accumulator

(ii) Register - the value in register D

(iii) Immediate - the value 50

(iv) Extended - the value in the address $6352

9.2 :L5 because a carry is produced.

10.2

	B	C	Carry
LD B.11	00001011 (11)	?	?
SRA B	00000101 (5)	?	1
LD C.F8	00000101 (5)	11111000 (-8)	1
SRA C	00000101 (5)	11111100 (-4)	0

```
11.2  BIT 0.A
      JR Z.L1
      :L2 SET 7.B
      JR L3
      :L1 RES 7.B
      :L3 RET
```

```
12.2  LD A.B
      RRCA
      OR C
      LD ($620C).A
```

13.2 $06, $68, $09, $00

```
14.2  LD HL.(16396)
      INC HL
      LD BC.10
      ADD HL.BC
      LD D.H
      LD E.L
      LD HL.$620C
      LDDR
      RET
```

15.2 A byte can contain BCD numbers in the range 0 to 99 and unsigned binary numbers in the range 0 to 255.

A.2 0 and 1

ANSWERS TO QUESTIONS NUMBERED 3

```
1.3    $BO

3.3    LD A.56
       SUB 22
       LD B.A
       ADD A.B
       ADD A.B

5.3    $24 (36) and $08 (8)

6.3    LD A.B
       ADD A.C
       JP M.L1
       JP Z.L2
       LD A.$35
       JP L3
       :L1 LD A.$33
       JP L3
       :L2 LD A.$3F
       :L3 RST 10
       RET

7.3    LD A.$25
       :L1 RST 10
       DEC A
       CP $1C
       JP P L1
       RET

8.3    LD HL.($620C)
       :L1 LD A.(HL)
       INC HL
       CP $50
       JR Z.L2
       RST 10
       JR L1
       :L2 RET
```

```
9.3   SCF
      CCF
```

```
10.3  SLA A
      LD B.A
      SLA A
      SLA A
      ADD A.B
```

11.3 The result of the XOR operator is one if the two binary values to be XORed are different, and zero if they are the same.

```
12.3                PARITY
      AND $FE         1
      SLA A           0
      RLA             1
```

```
13.3  SCF
      CCF
      SBC HL.BC
```

```
14.3  LD HL.(16396)
      INC HL
      LD D.H
      LD E.L
      LD HL.$620C
      LD BC.10
      :L1 LDI
      JP PE.L1
      RET
```

```
15.3    00010111     BCD 17
      + 01101001     BCD 69
        --------
        10000000     BCD 80 incorrect
      +     0110
        --------
        10000110     BCD 86
```

A.3 $9AB3 is equivalent to 1001101010110011B.

110011101111 is equivalent to $CEF.

This question shows the problems of working with binary numbers.

ANSWERS TO QUESTIONS NUMBERED 4

1.4 10000001B

3.4 27 or $1B

-27 or $E5

-26 or $E6

6.4

```
LD A.100
CP B
JP M.L1
JP Z.L2
JP L3
```

7.4

```
LD B.9
LD C.A
:L1 ADD A.C
DJNZ.L1
RET
```

11.4

	A	S	Z
LD A.$B5	10110101	?	?
LD C.$F0	10110101	?	?
AND $1F	00010101	0	0
OR C	11110101	1	0
XOR $CC	00111001	0	0
CPL	11000110	0	0
XOR A	00000000	0	1

12.4

```
:L10 BIT 0.B
JP Z.L1
AND $FF
JP PO.L2
JP L3
:L1 AND $FF
JP PE.L2
:L3 LD C.1
JP L4
:L2 LD C.0
:L4 RET
```

13.4 Replace the ADC instruction by a SBC instruction.

14.4

```
LD HL.$620C
LD DE.$640C
LD BC.400
:L1 LDI
LD A.(HL)
CP 0
JP NZ.L1
RET
```

15.4

```
  10000010     BCD 82
- 01010110     BCD 56
  --------
  00101100
      0110
  --------
  00100110     BCD 26
```

A.4

```
  $C7BA          01101101B
- $9FF8        + 01011110B
  -----          ---------
  $272C          11001011B
```

ANSWERS TO QUESTIONS NUMBERED 5

1.5 INC B

3.5 (i) $35 or 53

(ii) 79 or $4F

7.5

	A	B	C	D	E	SP
LD A.$OA	OA	?	?	?	?	?
LD B.$OB	OA	OB	?	?	?	?
LD C.$OC	OA	OB	OC	?	?	?
LD D.$OD	OA	OB	OC	OD	?	?
LD E.$OE	OA	OB	OC	OD	OE	?
LD SP.$7000	OA	OB	OC	OD	OE	$7000
PUSH AF	OA	OB	OC	OD	OE	$6FFE
PUSH BC	OA	OB	OC	OD	OE	$6FFC
PUSH DE	OA	OB	OC	OD	OE	$6FFA
POP BC	OA	OD	OE	OD	OE	$6FFC
POP DE	OA	OD	OE	OB	OC	$6FFE

14.5

```
LD HL.20499
LD DE.20599
LD BC.500
LDDR
```

15.5

	A
LD A.$43	$43
LD B.$28	$43
ADD A.B	$6B
DAA	$71

A.5 Unsigned numbers can be in the range 0 to 1111111111111111B, which is $FFFF or 65535.

ANSWERS TO QUESTIONS NUMBERED 6

14.6 Replace all the instructions after and including CPIR by the following:

```
:L1 CPI
JP PO.L2
JR NZ.L1
LD A.C
ADD A.$1C
RST 10
LD A.0
JR L1
:L2 RET
```

A.6

```
(a)               11000100        -60
                + 01000110      + +70
                  --------        ---
            (1)   00001010        +10

(b)               11101001        -23
                + 11010010      + -46
                  --------        ---
            (1)   10111011        -69

(c)               01010101        +85
                - 01100000      - +96
                  --------        ---
            (1)   11110101        -11
```

Assembly Language Assembled for the Sinclair ZX81

The Macmillan Press have made arrangements with BUG-BYTE Software to supply on cassette both the ZXAS assembler and the ZXDB disassembler programs which this book has been designed to accompany.

(Please note that the 16K RAM pack is required.)

Cassette ISBN 0 333 34589 4 £10.00 plus VAT
(for both programs)

Available through all major bookshops ...
but in case of difficulty order direct from

Globe Book Services
Canada Road
Byfleet
Surrey KT14 7JL