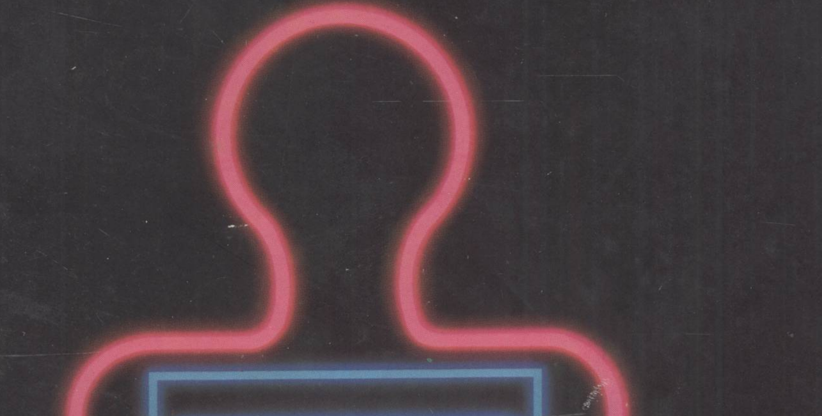
Z80 MACHINE CODE FOR HUMANS



LAN TOOTILL

ND DAVID BARROW

Z80 Machine Code for Humans

Alan Tootill and David Barrow

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd.,
8 Grafton Street, London W1X 3LA

First published in Great Britain by Granada Publishing 1983
Reprinted 1984

British Library Cataloguing in Publication Data
Tootill, Alan
Z80 machine code for humans.
1. Sinclair ZX80 (Computer) – Programming
I. Title II. Barrow, David
001.64'25 QA76.8.5625

ISBN 0-246-12031-2

Typeset by Columns of Reading
Printed and bound in Great Britain by Mackays of Chatham, Kent

Preface

We do not pretend that machine or assembler language coding is better than coding in some high-level language. It all depends on what you want to do and the resources you have at the time, particularly the time, for doing it.

If you want something up and running within a few days, and know and practise a high-level language suitable for solving your problem, and if you also have the facilities needed to make good use of that language, then use it. Machine code will only match it for speed of getting the job done if you have already prepared, tested and practised using machine code routines that can be put together as building blocks to form the bulk of your application. If, on the other hand, you need to fit your program into a small area of memory, or need extreme speed of execution, precise timing or control for such applications as animated graphics or monitoring critical processes, then you have little choice but to use machine code. Vague statements are often made about the relative speeds of machine code and BASIC. In Appendix C we give one concrete example.

But personal computing is about more than getting applications programmed quickly and easily. It is partly about the pleasure and satisfaction to be gained from exercising your mind. Hard mental effort, like physical or spiritual effort, has its rewards.

The satisfaction that comes from working with the instruction set that actually controls the computer, rather than through somebody else's translation or interpretation, is something that must be experienced to be understood. Working with the processor's own instructions is an obvious necessity to those who are not content with just knowing what the computer can do but also feel impelled to find out how it does it. It is an exercise that will dispel any misleading notions of the computer as some spooky 'black box' which is capable of understanding only highly complex languages and concepts. The machine will be revealed as it really is, capable of performing a few simple switching operations superbly well.

Programming, and machine code programming in particular, is a good recreation that takes you away from the mindless mouthings of the manipulators and image makers into a sphere where only what is true will work. That is probably why its practice will put you in company with some fine people.

Several people have told us that they cannot understand machine code even though we know they are quite capable of it. We have also seen how programmers like to check on how to do things by looking at code that is already working, rather than by digging information out of the suppliers' documentation. This book has been written primarily to help the many who have not got round to understanding machine code, by presenting enough code, doing interesting things, to show how it works. The book does not set out to explain, systematically, each instruction. Other books, not least the Zilog or Mostek programming manuals, do that. We hope that experienced coders will find the routines stimulating as well and that the detailed explanations, though not essential for them, will save them some time in understanding the routines.

Looking at other people's code can be mental torture for the novice if it is not fully explained. We have tried, therefore, to explain not only what the code is doing, line by line, but also to tell you in advance what it is going to do, so that you approach it knowing what to expect.

When following the code, you will probably think of different ways of doing things. There are so many possible ways of programming any function in machine code that most code can be improved if somebody spends enough time on it. So, if you find you can do any of these routines better than we have done, we will not mind at all. Write and tell us or, better still, write your own book and tell everybody!

We suggest that the novice first runs a routine to see what it does, then tries using it with other data and ideas of his or her own where suitable. He should then examine the code to see how the routine works and, finally, try to work out better code for doing the same thing. The secret of understanding machine code lies in not trying to take in too much at a time and in going back, after a break, as many times as necessary, to anything not fully understood.

It is well worth the effort to make any machine coding you do clear and visible to others, so that they, and you when you have moved on to other things, can see how the code works too. That is what we are trying to do in this book. Code that is visible can be improved, changed to serve other purposes and, best of all, shared with other users. This sharing of clearly documented effort should be applied to all aspects of computer work, otherwise the computer's potential to enhance man's effectiveness will not be fully realised. How this can be brought about, whilst it is so clearly in conflict with established commercial practice, is something that needs working on. Many personal computer owners are already showing the way!

We would like to thank Richard Miles of Granada Publishing for masterminding the production of this book, the authors' families who did not escape the traumas of its writing and the contributors to our machine code series *PCW SUB SET* in the British magazine, *Personal Computer World*, whose enthusiasm for machine code programming encouraged us to make this effort.

Most of the material in this book is new but two of the routines have appeared before in *Personal Computer World*. These are the delay routines, DL1S and URDZ in Chapter 4. They are included here with kind permission of *PCW*.

Alan Tootill
David Barrow

Contents

Chapter One
Getting Started

When a personal computer is switched on, it automatically carries out a cycle of fetching and executing instructions, from a set that its processor has been designed to handle. These instructions are in code and the Z80 codes are given in Appendix A. Any other form of instruction has to be converted into equivalent machine code instructions for execution, either in advance by a program called a *compiler*, or whilst running by a program called an *interpreter*. There is, in fact, another program, called an *assembler*, which helps to produce machine code by converting more easily remembered mnemonics of the machine instructions into the actual codes. In Appendix A, and throughout the book, we give these mnemonics alongside the machine code.

Computer codes, representing both instructions and data, are *binary* numbers, expressed for convenience in *hexadecimal* notation. If you are not familiar with these number systems, or the meaning of BCD (binary coded decimal) and the 2's complement of numbers, read Appendix B for a brief account. Other books on mathematics or computer technology will deal more fully with them but they are simple enough and you will soon become accustomed to using them.

It is possible, but not very interesting, to read machine code without trying it out. Since the machine code in this book is for humans (and we doubt whether reading machine code without seeing it work is!) it is assumed that you are working through the book using a microcomputer with a Z80 processor. There are certain facts you need to know or find out about your particular computer, before you can use machine code on it.

Your computer

You must know what RAM (memory that you can write into and read from) is available to you and the addresses at which it is located. In machine code you put everything where it should go and keep track of where it is. It follows from this that you will want to see what is contained in certain RAM locations and be able to change the contents at will.

You will have to know how to cause your computer to jump to execute code at the address of the memory location you have put it in. It is also useful to know the address to jump to after your code has finished, to produce an orderly return to your computer system and to allow it to await your next command. We have ended code in this book with a HALT (machine code 76H) because every Z80 processor has one. A return to await your next monitor, BASIC or other system command is preferable.

A *monitor* is a program that accepts commands to enable you to do all the things just mentioned and, probably, more. Typically, as well as allowing you to see and alter memory in hexadecimal, execute machine code programs and return for its next command, a monitor will stop a program at an address you specify and display all registers. It will also execute one instruction at a time displaying all registers in between, perform simple hexadecimal arithmetic, copy from one area of memory to another, copy memory to and from tape or disk and send data to and from an input/output port. A monitor will also have many other useful subroutines you could use in your own machine code programs.

Experienced BASIC programmers who do not have a custom-built monitor will probably find that they can do in BASIC all that is needed to enter and execute machine code programs. Another book in this series, *Introducing Spectrum Machine Code* by Ian Sinclair, approaches machine code through Spectrum BASIC and might well be useful to users of other BASICs who want to adopt the same approach.

For use with some of the code in this book, you will need a routine – as specified in Chapter 4 – to scan the keyboard for an input character. Since such a routine must depend on the hardware design of your computer, we cannot supply anything that would be of general use. Such a routine must be there somewhere, in monitor, BASIC interpreter or systems software, and you need to know the address at which it can be called.

You will need to know the codes for the characters that can be displayed in your system. Any character codes in the routines in this book are from the standard ASCII character set, used by most, but not all, personal computers. Most personal computers have a memory-mapped display, where a code in a location in RAM causes the character it represents to be displayed in a related screen character position. The display routines in this book are written for this system, more fully described in Chapter 5. To use them, you will have to know the start address of the display memory, the number of characters displayed on a line and the line difference (i.e. the difference in the memory addresses of the first character positions of two adjacent lines), since this is not the same in all systems as the number of characters in a line. If your computer uses some other display system, the display routines in this book will have to be adapted to suit and you will need to know how to do this.

Though not essential, an assembler program to enable you to enter and edit instructions in Zilog mnemonics and convert them into machine code is well worth having, if you are going to do much in machine code. Its main advantage is that it allows you to branch to, and call subroutines at, labelled instructions, calculating the addresses for you. If you then insert or delete instructions, the source code (mnemonics) can be re-assembled, taking care of all necessary re-addressing, without your having to decide which addresses are affected by the changes. An assembler also allows you to decide, each time it is used, where the assembled machine code is to go in memory.

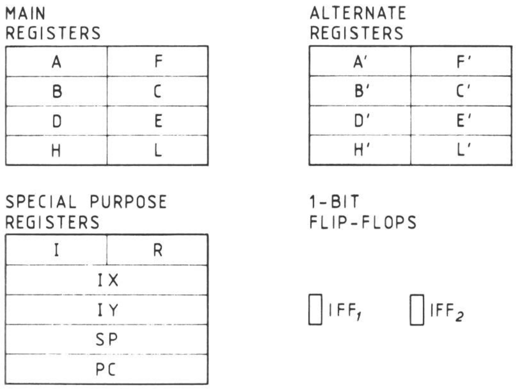


Fig. 1.1. Z80 register set.

The Z80 registers

The Z80 processor has a number of internal registers, as shown in Figure 1.1, into and out of which the machine code programmer can load data.

bit	7	6	5	4	3	2	1	0
	S	Z	-	H	-	P/V	N	C

Fig. 1.2. F register (flags).

The A register, or *accumulator*, holds the result of 8-bit arithmetic or logical operations whilst in the F register, or *flags*, six of the eight bits, as shown in Figure 1.2, are used to indicate conditions arising from 8- or 16-bit operations. The S, or *sign flag*, has the state of the most significant bit of the result. The Z, or *zero flag*, is set to 1 when the result is zero. The H, or *half carry flag*, is set to 1 when an add or subtract produces a carry into or a borrow from bit 4 of the accumulator. It is used by the DAA (decimal adjust) instruction in binary coded decimal arithmetic. Parity and overflow share the same P/V flag. In logical operations even parity is shown by 1 and odd parity by 0. In arithmetic operations the flag is set to 1 if the result produced overflow. The N, or add/subtract flag, is set to 1 if the previous operation was a subtract. It is used by the DAA (decimal adjust) instruction in binary coded decimal arithmetic. The C, or carry flag, is set when an add instruction generates a carry or a subtract instruction generates a borrow. The DAA instruction will also set the carry when an adjustment has been made. As you work through the book, you will come across instructions that act on the state of these flags.

B, C, D, E, H and L are 8-bit general purpose registers that can also be used in pairs – BC, DE and HL as 16-bit registers. These, with the A and F registers, are duplicated in a set of alternate registers. The main and alternate general purpose registers can be switched with a single instruction, as can the main and alternate A and F registers. Because they can be exchanged so quickly, alternate registers are sometimes reserved for use in interrupt service routines, where a swift response is crucial.

The I, 8-bit *interrupt vector*, is used in one of the interrupt methods, as discussed in Chapter 11. The R, *refresh register*, is a 7-bit counter that can be used in conjunction with the hardware to maintain automatically the contents of a commonly used type of memory, dynamic RAM. It is not normally used by the programmer, although it can be accessed for testing purposes.

IX and IY are 16-bit *index registers*, which are used to hold a 16-bit base address for accessing data stored in tables. You will see this use of the IY register in the next chapter.

The SP (*stack pointer*) register holds a 16-bit address of the current position in an area of RAM, used as a temporary store. It is discussed below. Another 16-bit register, the PC (*program* counter) holds the address of the current instruction being fetched from memory. It is automatically increased to point to the next instruction, as soon as its contents have been transferred to the address lines. Jump, call, return and restart instructions, requiring a jump out of the program sequence, override the address of the next instruction. The IFF flip flops enable the processor to keep track of interrupts and are discussed in Chapter 11.

Addressing

Most Z80 instructions operate on information stored in the registers, memory or ports and the Z80 instructions provide various methods of addressing this information.

Some instructions indicate in themselves where the information is and this is known as *implied addressing*. They alter the state of the accumulator, the carry flag, the IFF flip flops or perform block moves. In *register addressing*, the information is in an 8-bit or 16-bit register.

Sometimes the information – in either one or two bytes – is in the program, immediately following the operation code. This is known as *immediate addressing*. When the information is a 16-bit number or address, the low order byte, contrary to mathematical practice, is placed first in the machine code, before the high order byte. The machine code of instruction LD BC,0B0CH is 01 0C 0B. This is to suit the Zilog electronics. In our machine code, because we do not know the addresses in your system at which you will locate these routines, we show calls to them as CD lo hi. If you are storing 16-bit numbers or addresses to be picked up one byte at a time, there is no reason why they should not be stored in the normal order, with the most significant byte before the least significant one. We do this with addresses in key area RAM described in the next chapter.

The Z80 has eight locations in page zero memory, 00H, 08H, 10H, 18H, 20H, 28H, 30H and 38H, to which special call instructions, the single byte RST (restart), can direct the program to perform a subroutine. These locations are usually reserved for most frequently used, or interrupt service, subroutines. The use of these restart instructions is known as *modified zero page addressing*.

In *relative addressing* a jump operation code is followed by a signed, 2's complement, 8-bit displacement. This displacement is added to the program counter after it has been incremented to point to the next instruction, so that the jump is always relative to the address of the byte after the displacement. A close look at some displacements is taken in Chapter 2, following the routine HLARR. The displacement can only be in the range minus 128 to plus 127.

Extended addressing refers to accessing information at any 16-bit address, which, in the source code or mnemonics, is shown in brackets. LD A,(0F60H) will put into the accumulator the contents of the byte at location 0F60H.

Another way of doing this is by *register indirect addressing*, where a 16-bit register pair holds the address of the information to be accessed. LD A,(HL) will put into the accumulator the contents of the byte at the address in HL.

The two special purpose registers IX and IY are used in *indexed addressing*. This differs from register indirect addressing in that the two-byte operation code is followed by a displacement byte, which is added to the address in the register to form the address of the information to be accessed.

The Z80 has *bit addressing*; instructions that allow you to operate on a single bit, which can be set, reset or tested.

Finally, there is *stack pointer addressing*, where the address of the memory locations to be operated on is in the stack pointer. The two instructions which use this addressing method are PUSH and POP.

The stack

The *stack* is a stack of bytes, wherever the program places it in RAM, into which certain Z80 instructions place and remove data – in two-byte units – on a last in, first out basis. The machine code programmer has to arrange for there to be enough stack

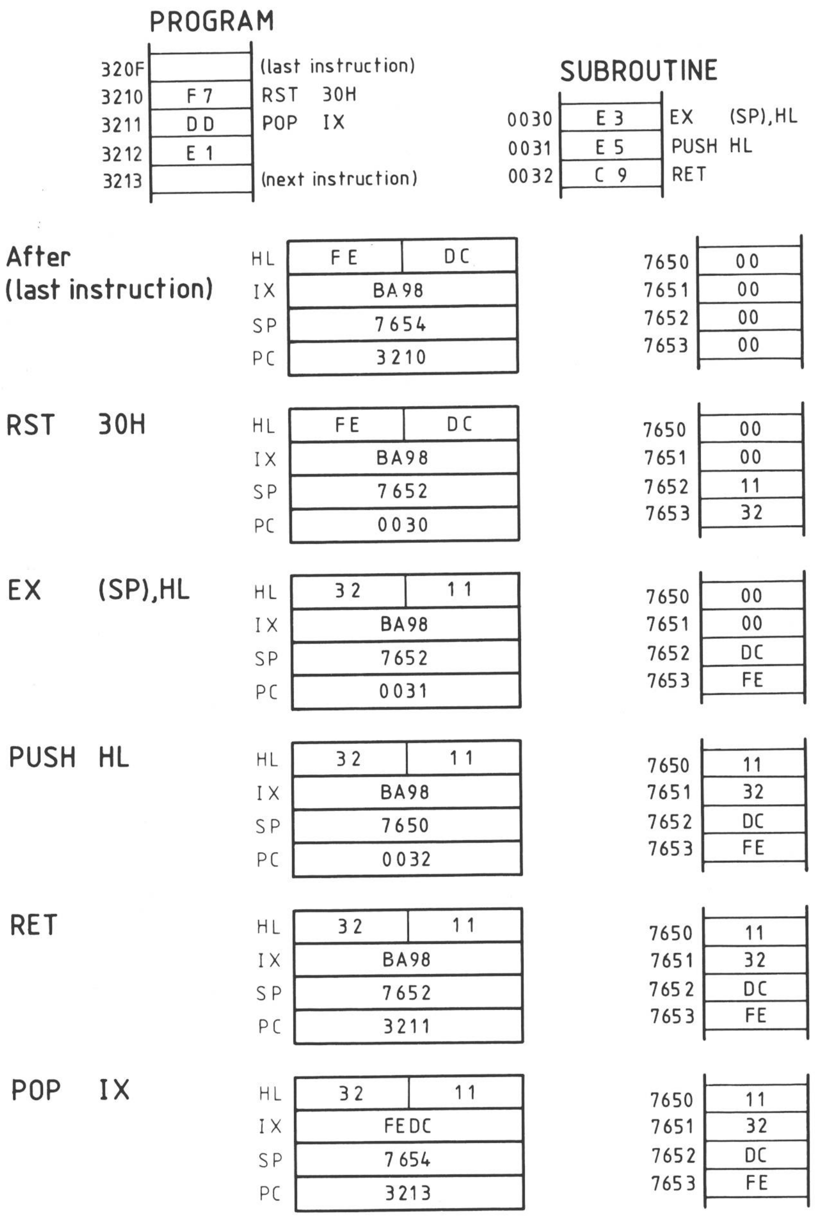


Fig. 1.3. Program counter (PC), stack pointer (SP) and stack changes.

memory to hold the amount of data placed onto and not removed from the stack at any one time.

When your Z80 microcomputer is switched on and the cycle of fetching and executing instructions begins, either a monitor or BASIC interpreter will set the stack pointer pointing to an area of RAM reserved for that program's stack. That area of RAM might, or might not, be enough for your machine code program's stack. If it is not, or you do not know how much RAM has been reserved, you can load the stack pointer, via either the HL, IX or IY register, with any address you choose.

The stack pointer, after being given its address by the program, is automatically adjusted to keep track of the currently available stack location, by the instructions that use the stack. When a unit of information is placed, or pushed, onto the stack, the stack pointer is decreased by 1, the high order byte of the data is stored at the address in the stack pointer, the stack pointer is decreased by 1 again and the low order byte is stored at the new address in the stack pointer. When a unit of information is popped from the stack, the low order byte at the address in the stack pointer is recovered, the stack pointer is increased by 1, the high order byte at the new address in the stack pointer is recovered and the stack pointer is increased by 1 again. PUSH and POP instructions move data between 16-bit registers and the stack, as in Figure 1.3.

Call and restart instructions stack the contents of the program counter, after it has been increased to point to the next instruction and before it has been overriden by the subroutine address to which the program next goes, as shown in Figure 1.3. Any of the return instructions put the address at the current stack location into the program counter, to cause program execution from the instruction following the call or restart. This allows for unlimited nesting of subroutines within subroutines, provided you reserve enough stack RAM for your program.

Documentation of routines in this book

Our code is preceded by standard information helpful to its use. The length is given in bytes and the stack use is the number of bytes used by the routine and any routine it calls but excluding the call to the routine itself. Each line of code consists of a label field (occupied occasionally where necessary), the *Zilog* instruction mnemonic, comment preceded by a semi-colon, and then the machine code instruction (from one to four bytes long) in hexadecimal.

Chapter Two

Organising for Character Display

Programming a major application in machine code is a very different matter from writing a single routine to perform some limited, easily defined task. In the latter case, all the information needed can often be passed to and from the routine in registers, of which the Z80 has a good supply. But when a lot of these routines are being used together in a larger work, saving on the stack and accessing these registers as needed can become a major brain bender. It is less efficient for the processor perhaps but better for your peace of mind, to reserve certain areas of memory (RAM) where the information most commonly needed will always be found by whatever routine needs it.

There is another advantage in having these reserved memory areas. They offer more scope for writing routines that can be made to do different things *simply* by altering the information in memory, rather than by altering any code calling the routine. This should all become clear as you read on.

We set about here organising some RAM to hold information we will need in a system to ask for, receive and check data supplied by the personal computer user through the keyboard. The IY register will be set to point to the key area of the RAM, which will look like this:

IY+			
00	hi	the home	**The current display area**
01	lo	address.	
02	col	N (the number of columns).	
03	row	N (the number of rows).	
04	col	B diff (byte difference between columns).	
05	row	B diff (byte difference between rows).	
06	col	PP (print position).	