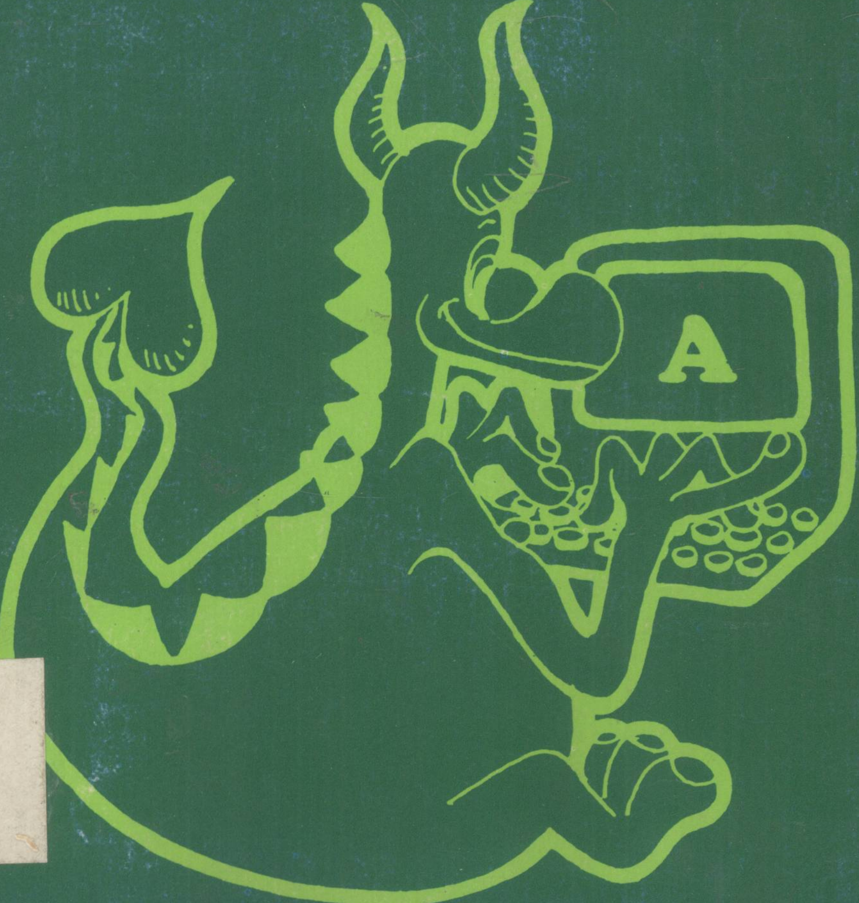
dP Series in Software
Number 3

8080 Machine Language Programming for Beginners

Ron Santore



8080 MACHINE LANGUAGE PROGRAMMING FOR BEGINNERS

Ron Santore

dilithium Press
Portland, Oregon

10 9 8 7 6 5 4 3 2 1

ISBN: 0-918398-14-2
Library of Congress number: 78-53003

Printed in the United States of America

dilithium Press
30 N.W. 23rd Place
Portland, Oregon 97210

PREFACE

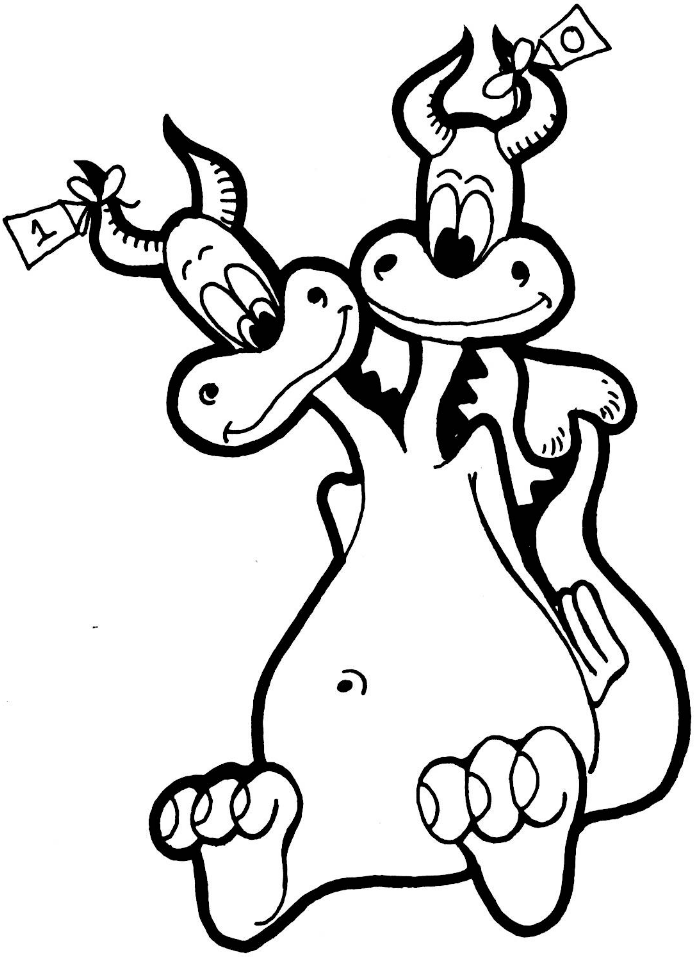
This book is not simply a description of 8080 op-codes and their definitions, but is rather a course which will lead you step by step into the basics of machine language programming. Although machine language may appear difficult at first glance, I believe you will find this book takes nothing for granted. In writing it, I have assumed you know nothing about programming. As we go along, everything will be defined for you, and in each chapter you will write a program or subroutine. In this format you will only be introduced to a few new programming instructions at a time. You will start by writing simple subroutines, then you will progress to longer programs, and, as the chapters proceed, you will become familiar with common 8080 machine language programming instructions.

A lot of care was taken to condense the subject matter covered, and I hope you won't find this a wordy text. Because each and every paragraph is important, you should not try to rush through the chapters or try to cover a lot of pages in a short period of time.

Since I am presenting this material from a beginner's standpoint, some of it may be old hat to you, but I wanted to give every benefit to those who are new to this field. Understand that this is a book on basics, not technique, so I assume you are a beginner with an 8080 microcomputer that you want to learn how to use. If you find yourself reading something that you already know, read it through anyway. In that way you may gain a better foundation for what is to follow.

INTRODUCTION

The first section of Chapter 1 of this book provides a foundation for your introduction into programming. In these pages I have provided brief definitions of the basic terms you will be using in the rest of the book and for as long as you remain associated with computers. These first pages contain vital information, so don't skim over them—take the time to absorb what they offer and you will be better able to appreciate the rest of the book.


1
0

CONTENTS

1 BACKGROUND AND THE OUTPUT SUBROUTINE

THE BINARY SYSTEM

Our decimal number system contains ten integers. They are: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The binary number system contains only two integers. They are 0 and 1. In the binary system of numbers, a "1" in the lowest (rightmost) column represents a decimal 1. A binary "1" in the next column represents a decimal 2. A binary "1" in the next column represents a decimal 4, and the next column represents a decimal 8. Study the following table:

decimal		binary
0	=	0
1	=	1
2	=	10
3	=	11
4	=	100
5	=	101
6	=	110
7	=	111
8	=	1000

Notice that we, as humans, could write any decimal number and then also write its binary equivalent, but the binary numbers are a little awkward for us because they take up so much space on paper (decimal 256 = 100000000 in binary). As it turns out, though, the binary system is much easier for a computer to handle than our ten-digit decimal system is.

Adding binary numbers is even easier than adding decimals:

$$\begin{array}{r}0\\+0\\\hline =0\end{array}\qquad\begin{array}{r}0\\+1\\\hline =1\end{array}\qquad\begin{array}{r}1\\+0\\\hline =1\end{array}\qquad\begin{array}{r}1\\+1\\\hline =10\end{array}$$

Notice that in the last example, a "carry" was performed—the lowest column filled up, so a "1" was carried to the second column. Can you see that in binary?

$$\begin{array}{r}011\\+001\\\hline =100\end{array}$$

See if you can answer these questions:

1. Give the equivalents for these numbers:

decimal		binary
1	=	
0	=	
2	=	
6	=	
5	=	
	=	11
	=	111
	=	100

2. How many integers are there in the binary system?
3. What do you think the decimal number 9 would look like in binary?
4. Add the binary numbers, then write the sums in binary and in decimal:

$$\begin{array}{r}000\\+001\\\hline =\end{array}\qquad\begin{array}{r}000\\+010\\\hline =\end{array}\qquad\begin{array}{r}100\\+001\\\hline =\end{array}\qquad\begin{array}{r}001\\+101\\\hline =\end{array}\qquad\begin{array}{r}101\\+010\\\hline =\end{array}\qquad\begin{array}{r}001\\+111\\\hline =\end{array}$$

Binary numbers can get very large; when they do, they become hard for us to remember. For now, learn the first eight binary numbers and be able to recognize them at a glance.

BIT

Your computer only understands binary numbers. A bit is a binary digit or integer and can only be 1 or 0. The binary number 01101011 contains eight bits.

This is a bit . . . 1
or this is a bit . . . 0

BYTE

A single bit by itself can only represent two states, a "1" or a "0," so in order to make the system more useful, bits are grouped together to form bytes or words. The 8080 computer always uses eight-bit bytes, so for your computer a byte is always eight bits of binary information.

This is a byte . . . 11100100
or this is a byte . . . 00111101

ADDRESS

An address is a place or location in memory. At each address in memory, there is one byte of data. To see a particular data byte in memory, just examine its address. For the 8080 system, an address is always sixteen bits.

This is an address . . . 00001011,01101111
or this is an address . . . 10110000,10110011

Remember, each address *contains* one byte of data, and each address is sixteen bits. As an example, if we look at address 00001011,01101111 we might find byte 11100100.

THE OCTAL CODE

Bytes and addresses are a little hard to remember because they are so long, so bits are usually grouped as follows:

a typical eight-bit byte
00101011 becomes 00 101 011

and a sixteen-bit address
00000101,01101111 becomes 00 000 101,01 101 111

Now if you remember your binary numbers, you can see how to code these numbers into octal:

the eight-bit byte
00 101 011 becomes 0 5 3

the sixteen-bit address
00 000 101,01 101 111 becomes 0 0 5,1 5 7

The octal code is very important. Study it closely and answer these questions before going on;

1. How many bits are in a byte?
2. How many bits are in an address?

3. Convert these:

binary		octal
00 000 100	=	0 0 4
00 000 011	=	
00 001 000	=	
01 000 101	=	
10 001 001	=	
01 111 000	=	
	=	0 0 1
	=	3 0 3
	=	3 7 2
	=	2 1 1
	=	0 6 5
	=	3 1 1

There are other ways to group bytes and addresses, but the octal code seems to be the easiest for the beginning programmer to understand. For this reason, the rest of this book is based on octal programming. The binary numbers used in octal programming are repeated as follows. You will need to know them by memory.

decimal		binary
0	=	0
1	=	1
2	=	10
3	=	11
4	=	100
5	=	101
6	=	110
7	=	111

AND/OR LOGIC

This part is easy! AND/OR logic is a kind of test we will be using to check our data bits. It is a little like adding two numbers, only with different rules.

AND: Let's assume we want to "AND" two bits:
If both bits are 0, the result is 0.
If one bit is 1 and one bit is 0, the result is 0.
If both bits are 1, the result is 1.

OR: Let's assume we want to "OR" two bits:
If both bits are 0, the result is 0.
If one bit is 1 and one bit is 0, the result is 1.
If both bits are 1, the result is 1.

Remember:

$$\begin{array}{r} 0 \\ \underline{\text{AND } 0} \\ \text{IS } 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{AND } 1} \\ \text{IS } 0 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{AND } 0} \\ \text{IS } 0 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{AND } 1} \\ \text{IS } 1 \end{array}$$

$$\begin{array}{r} 0 \\ \underline{\text{OR } 0} \\ \text{IS } 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{OR } 1} \\ \text{IS } 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{OR } 0} \\ \text{IS } 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{OR } 1} \\ \text{IS } 1 \end{array}$$

You can do it with bytes, too:

$$\begin{array}{r} 10001110 \\ \underline{\text{AND } 11000101} \\ \text{IS } 10000100 \end{array} \qquad \begin{array}{r} 11110001 \\ \underline{\text{AND } 10011000} \\ \text{IS } 10010000 \end{array} \qquad \begin{array}{r} 11111111 \\ \underline{\text{AND } 00010001} \\ \text{IS } 00010001 \end{array}$$

$$\begin{array}{r} 11111111 \\ \underline{\text{OR } 00011000} \\ \text{IS } 11111111 \end{array} \qquad \begin{array}{r} 01010101 \\ \underline{\text{OR } 01011100} \\ \text{IS } 01011101 \end{array} \qquad \begin{array}{r} 11000011 \\ \underline{\text{OR } 00000000} \\ \text{IS } 11000011 \end{array}$$

EXCLUSIVE OR

"Exclusive OR" is very much like "OR" logic. It is abbreviated "XOR," and the rules are:

If both bits are 0, the result is 0.
If one bit is 1 and one bit is 0, the result is 1.
If both bits are 1, the result is 0.

$$\begin{array}{r} 0 \\ \underline{\text{XOR } 0} \\ \text{IS } 0 \end{array} \qquad \begin{array}{r} 0 \\ \underline{\text{XOR } 1} \\ \text{IS } 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{XOR } 0} \\ \text{IS } 1 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\text{XOR } 1} \\ \text{IS } 0 \end{array}$$

Look back at the "OR" logic to see the difference between OR and XOR.

Some examples using bytes:

$$\begin{array}{r} 10001110 \\ \underline{\text{XOR } 11000101} \\ 01001011 \end{array} \qquad \begin{array}{r} 11110001 \\ \underline{\text{XOR } 10011000} \\ 01101001 \end{array} \qquad \begin{array}{r} 11111111 \\ \underline{\text{XOR } 00010001} \\ 11101110 \end{array}$$

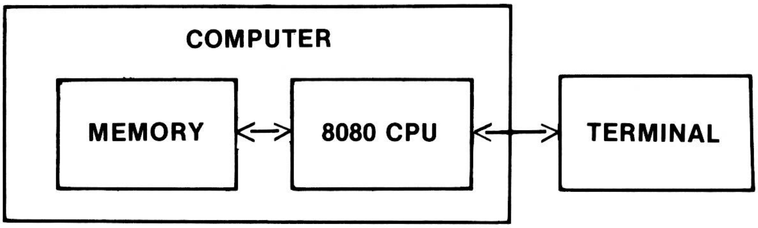
$$\begin{array}{r} 10001110 \\ \underline{\text{OR } 11000101} \\ 11001111 \end{array} \qquad \begin{array}{r} 11110001 \\ \underline{\text{OR } 10011000} \\ 11111001 \end{array} \qquad \begin{array}{r} 11111111 \\ \underline{\text{OR } 00010001} \\ 11111111 \end{array}$$

You will need to know the rules for AND, OR, and XOR by memory.

THE COMPUTER

A computer consists of three main elements:

1. The *central processor unit* (CPU or MPU) controls the computer. In small systems, it is usually a single integrated circuit which will "read" your program, decide what you want done, and do it. The central processor is the brains of your computer (besides you, of course).
2. The *memory* is simply a storage area for data. The computer's memory can't carry out your commands; it can only store them while they are waiting to be read by the CPU. Memory can store other data in addition to your programming commands.
3. The *terminal* usually consists of a keyboard and a printout device, both of which let you communicate with the computer. The terminal is usually in a cabinet separate from the main computer and is connected with wires.



In some of the newest home computers, all three elements are contained in the same cabinet.

The Central Processor

The central processor has eight registers in it. A register is a "container" in which data is temporarily stored, and each register will hold the same amount of data.

The registers are called: B
C
D
E
H
L
ACCUMULATOR
and the Condition word

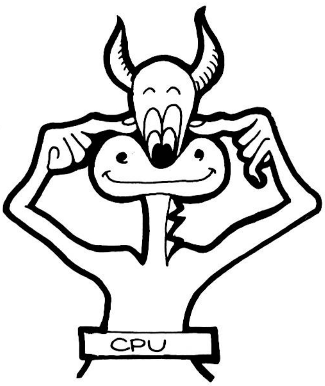
What good are the registers? You will find that registers are necessary in programming.

Machine language programming involves:
  Putting data into a register,
  or moving data from one register to another,
  or retrieving data from a register
Remember:
  The registers are all inside the CPU.
  The most useful register is the ACCUMULATOR.

In the pages that follow we are going to start looking at the actual operation of the terminal and computer. The terminal is not linked directly to the computer—there is a small circuit in between called an *interface*. In most 8080 computer systems, the circuit is a serial interface. With a serial interface, one bit of data at a time is exchanged from computer to terminal or vice versa. Since we know that one byte is eight bits, it takes time for a whole byte to be exchanged one bit at a time. For this reason, the usual method of data exchange is the following:

1. Ask for the terminal STATUS byte.
2. Test the STATUS byte to see if the terminal is ready to input or output data.
3. If the STATUS test says that the terminal is ready, then input or output the data; if the STATUS test says that the terminal is not ready, then go back to step 1 and recheck the STATUS.

The rest of this book will assume that your terminal is connected with a serial interface; the only portion of programming that concerns interfacing, however, is the input/output routine. So . . . if you have some other interface system, just disregard my Input/Output routines, which will be labeled as such, and substitute your own. MITS and IMSAI 8080 serial systems both use the same input/ouput STATUS routines that I have used in this text.



Your Terminal

If you remember from page 6, the terminal lets you communicate with the computer CPU. Each terminal has two numbers associated with it. I will be using the octal numbers 000 and 001 for the terminal. Here's how it works:

> If you input from the terminal using the number 000, you will get the terminal STATUS byte. A typical STATUS byte might look like

01100011

If the first bit is a 0, the terminal is ready to display output data. If the first bit is a 1, the terminal is not ready to display data.

If the last bit is a 0, the terminal is ready to input data to the CPU. If the last bit is a 1, the terminal is not ready to input data.

The middle six bits of the STATUS byte might be any combination of 1's and 0's, but they don't matter to us right now; we only care about the first or last bit when determining the terminal STATUS.

If you input from the terminal using the number 001, you will get the terminal DATA byte. The reason you have to get the STATUS byte first and then get the DATA is that the computer will operate much faster than the terminal. Your computer can take in DATA, or put it out, much faster than the terminal can.

To input DATA from the terminal,
 First get the STATUS word using octal number 000.
 Wait for that last bit to go from 1 to 0.
 Then get the DATA using octal number 001.
To output DATA to the terminal,
 First get the STATUS word using number 000.
 Wait for that first bit to go from 1 to 0.
 Then output the DATA using number 001.

You will understand better how this works in just a few pages when we get into the actual programming codes, but the main thing to remember is that the terminal can be addressed using two different octal numbers, 000 and 001; 000 is used to determine terminal STATUS, and 001 is used to determine terminal DATA.

The above information could be different for your system—for instance, you might test two middle bits from the STATUS word to determine terminal status, or your terminal might be addressed differently than 000 and 001.

THE ASCII CODE

The ASCII code is just a way to represent a letter of the alphabet or number using an eight-bit data byte. I have listed the most common codes here. You do not need to know these by memory, but take a minute to study the table.

character	binary code	octal code
A	01000001	1 0 1
B	01000010	1 0 2
C	01000011	1 0 3
D	01000100	1 0 4
E	01000101	1 0 5
F	01000110	1 0 6
G	01000111	1 0 7
H	01001000	1 1 0
I	01001001	1 1 1
J	01001010	1 1 2
K	01001011	1 1 3
L	01001100	1 1 4
M	01001101	1 1 5
N	01001110	1 1 6
O	01001111	1 1 7
P	01010000	1 2 0
Q	01010001	1 2 1
R	01010010	1 2 2
S	01010011	1 2 3
T	01010100	1 2 4
U	01010101	1 2 5
V	01010110	1 2 6
W	01010111	1 2 7
X	01011000	1 3 0
Y	01011001	1 3 1
Z	01011010	1 3 2
1	00110001	0 6 1
2	00110010	0 6 2
3	00110011	0 6 3
4	00110100	0 6 4
5	00110101	0 6 5
6	00110110	0 6 6
7	00110111	0 6 7
8	00111000	0 7 0
9	00111001	0 7 1
0	00110000	0 6 0