8080 MACHINE LANGUAGE PROGRAMMING FOR BEGINNERS

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.

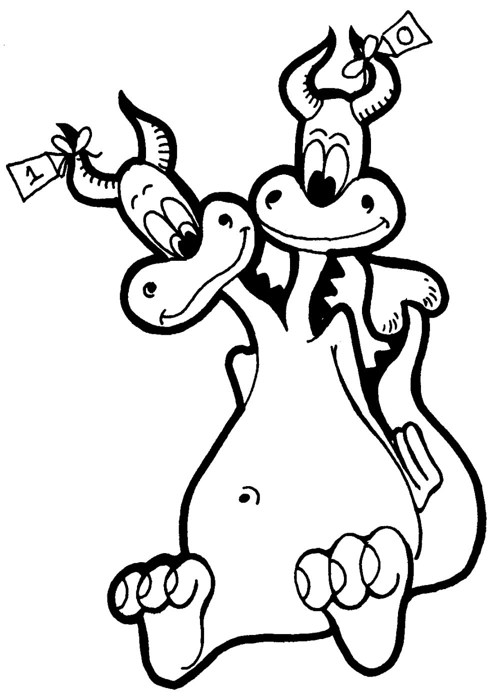
A lot of care was taken to condense the subject matter covered, and I hope you won't find this a wordy text. Because each and every paragraph is important, you should not try to rush through the chapters or try to cover a lot of pages in a short period of time.

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.

CONTENTS


1
0

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:

0	0	1	1
+ 0	+ 1	+ 0	+ 1
= 0	= 1	= 1	= 10

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?

011
+001
=100

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:

000	000	100	001	101	001
+001	+010	+001	+101	+010	+111
=	=	=	=	=	=

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:

```
     0         0         1         1
 AND 0     AND 1     AND 0     AND 1
  IS 0      IS 0      IS 0      IS 1

     0         0         1         1
  OR 0      OR 1      OR 0      OR 1
  IS 0      IS 1      IS 1      IS 1
```

You can do it with bytes, too:

```
     10001110        11110001        11111111
 AND 11000101    AND 10011000    AND 00010001
  IS 10000100     IS 10010000     IS 00010001

     11111111        01010101        11000011
  OR 00011000     OR 01011100     OR 00000000
  IS 11111111     IS 01011101     IS 11000011
```

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.

```
     0         0         1         1
 XOR 0     XOR 1     XOR 0     XOR 1
  IS 0      IS 1      IS 1      IS 0
```

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

Some examples using bytes:

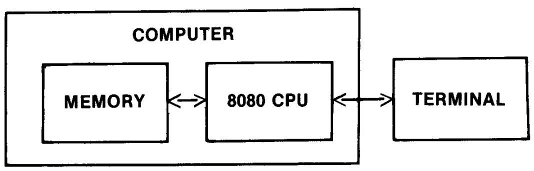
```
     10001110        11110001        11111111
 XOR 11000101    XOR 10011000    XOR 00010001
     01001011        01101001        11101110

     10001110        11110001        11111111
  OR 11000101     OR 10011000     OR 00010001
     11001111        11111001        11111111
```

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

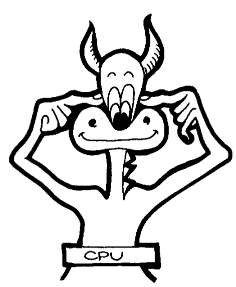
What good are the registers? You will find that registers are necessary in programming.

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

Your terminal understands only ASCII DATA bytes. The data byte 01,000,001 is what comes from the terminal DATA line when you type the letter "A" on the keyboard.

The DATA byte 00,110,010 is what comes from the DATA line when you type "2" on the keyboard. Notice the difference between a binary 2 (00,000,010) and an ASCII "2" (00,110,010).

If you want to display a "3" on the terminal, you would have the computer send out an ASCII "3," which is 00,110,011.

Remember the terminal transmits and receives only ASCII. If you send any other DATA bytes to the terminal besides ASCII DATA bytes, they probably won't be displayed.

SHORT QUIZ

```
      11101111         00010001         11110000
  AND 10010001     AND 00010000     AND 00111111
  IS               IS               IS

      00011111         11111111         00000000
  OR  11001010     OR  10000010     OR  00011000
  IS               IS               IS
```

1. How many bits are in a byte?
2. How many bits are in an address?
3. Change the binary number 01 000 011 to octal.
4. Change the octal number 303 to binary.
5. In an 8080 computer, where are the eight registers located?
6. What does the STATUS register tell us?
7. In the STATUS byte, which two bits are most important?
8. What do these bits tell us?
9. When inputting or outputting to the terminal,
   what octal number is used to ask for STATUS?
   what octal number is used to ask for DATA?
10. Which works faster, the computer or the terminal?
11. Why do we need the STATUS register?
12. Look back to page 9 and write down the ASCII code
    in octal for the letter "A."
    in binary for the letter "A."

In your later programming, you may have occasion to write a program which will put words in alphabetical order. Can you see anything about the ASCII code on page 9 which might make such a program fairly easy to write?

LET'S START PROGRAMMING!

If you know the material in the previous section, you are ready to start programming. To run a program, we will first store the programming instructions, in order, in memory. Then we will start the computer at the beginning of the program and run it.

Programming instructions are the machine language commands that tell the CPU what to do. They are often called *op-codes*. In this text the commands will always be in octal. Remember, however, that octal codes are really just a shorthand way of writing an eight-bit binary byte; therefore, machine language commands (op-codes) are eight bits long.

The procedure for writing your program will be to start at address 000,000 and store an op-code there; then move on to the next address 000,001 and store another op-code; and so on until every command in your program has been stored in memory.

Your first program is listed as follows. You are not expected to understand it yet, but look it over and notice that each octal address contains an octal op-code command. The following section will explain each program command individually and define its function.

The purpose of your first program is to display an ASCII character repeatedly on the terminal. Read the following pages carefully, so that you understand how the program is supposed to work, before running it on page 16. Remember that the STATUS check was written for a serial interface—if your system uses a parallel interface or some other method, you will have to substitute your STATUS routine for mine.

Here is the program:

OCTAL ADDRESS		OP-CODE		EXPLANATION
000,000	STATUS CK	333	IN	Input
000,001		000		the terminal STATUS byte.
000,002		346	ANI	AND the ACCUMULATOR
000,003		200		with 10 000 000.
000,004		302	JNZ	Jump if not zero
000,005		000		to address 000,000.
000,006		000		
000,007		076	MVIA	Move into the ACCUMULATOR
000,010		101		an ASCII "A."

000,011	323 OUT	Output
000,012	001	DATA to the terminal.
000,013	303 JMP	Jump
000,014	000	to address 000,000.
000,015	000	

The first part of the program checks terminal status. Look at the first six steps:

ADDRESS	OP-CODE	EXPLANATION
000,000	333	This is the INPUT instruction and it must always be followed by the terminal number. The INPUT op-code causes one byte of DATA (or the STATUS byte) to be moved from the terminal into the ACCUMULATOR.
000,001	000	This is the terminal number for STATUS. The STATUS byte is moved into the ACCUMULATOR (for now, assume the STATUS byte is 01,100,011).
000,002	346	This is the "AND" ACCUMULATOR op-code. It must always be followed by one DATA byte. The byte that follows will be "ANDed" with the ACCUMULATOR, and the result will be put into the ACCUMULATOR.
000,003	200	Since the "AND" instruction is followed by 200, the ACCUMULATOR will be ANDed with 10,000,000. This is a little tricky, but let's look at it:

The ACCUMULATOR contains STATUS	01 100 011
and we AND it with	<u>10 000 000</u>
so the ACCUMULATOR now contains	00 000 000
If the STATUS word had been	11 100 011
and we AND it with 200	<u>10 000 000</u>
then the ACCUMULATOR would be	10 000 000

		When the STATUS byte is ANDed with 10 000 000, we find out if the first bit of the STATUS byte is a "1" or if it is a "0"; this tells us if the terminal is ready to accept output DATA. This concept is important.
000,004	302	The JUMP IF NOT ZERO op-code must be followed by an address. Since an address is sixteen bits, it must be followed by two bytes.
000,005 000,006	000 000	The JNZ instruction uses the result of the AND test just given. If the result of the AND test was *not* zero, then the CPU will jump back to address 000,000 and continue from there. If the result of the AND test was zero, then the CPU would not jump back, but instead it would continue on to address 000,007.

When the first bit of the STATUS byte goes to zero, then the terminal is ready to display—and the program continues:

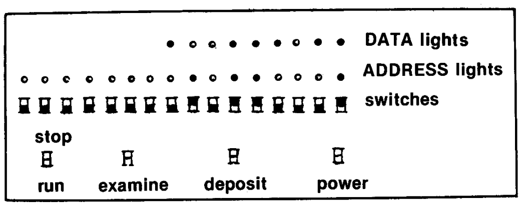
ADDRESS	OP-CODE	EXPLANATION
000,007	076	MVIA. This command moves one data byte into the ACCUMULATOR. The MVIA command must always be followed by one byte.
000,010	101	An ASCII "A" (01 000 001) is moved into the ACCUMULATOR.
000,011	323	OUT. This instruction takes whatever byte that is in the ACCUMULATOR and outputs it to the terminal.
000,012	001	The OUTPUT instruction must always be followed by the terminal number, which is 001 for DATA.
000,013	303	JUMP. The JUMP instruction must be followed by an address. Since an address is sixteen bits, it must be followed by two bytes.

000,014	000	The CPU will jump back to address 000,000
000,015	000	and continue from there.

You should be able to see that each time the program runs through, an "A" will be printed on the terminal. Also note that the program will never get past address 000,015 because each time it sees the JUMP at 000,013, it will return back to 000,000, which will start the program all over again. The terminal will be printing A's as fast as it can.

Look at this summary:

000,000	333	IN	This tells the CPU if the terminal is ready to display DATA.
000,001	000	STATUS	
000,002	346	ANI	
000,003	200	10 000 000	
000,004	302	JNZ	
000,005	000	address 000,000	
000,006	000		
000,007	076	MVIA	This puts an "A" into the ACCUMULATOR.
000,010	101	an ASCII "A"	
000,011	323	OUT	This outputs the ACCUMULATOR to the terminal.
000,012	001	terminal DATA	
000,013	303	JUMP	Return back to the beginning.
000,014	000	address 000,000	
000,015	000		



Locate these features on your computer and note:

If a light is on, it means it is a "1."
If a light is off, it means it is a "0."
If a switch is up, it means it is a "1."
If a switch is down, it means it is a "0."

PROGRAMMING AN OP-CODE

If you want to store the INPUT instruction 333 (11,011,011) at address 000,000(00,000,000 00,000,000), do the following.

Flip the RESET switch if you have one.
Put all sixteen switches down.
Flip the EXAMINE switch to EXAMINE.
All the address lights will go off, which means you are looking at address 00,000,000 00,000,000.
Flip the last eight switches to make 11 011 011.
Flip the DEPOSIT switch to DEPOSIT.
The eight DATA lights will show 11 011 011.

You have stored 11 011 011 at address 00,000,000 00,000,000. Note that the DATA lights and ADDRESS lights are separate but that the switches are used for both DATA and ADDRESSES; and since DATA is only eight bits, use only the eight switches on the right for storing DATA.

The preceding method is the way to enter programs into the computer after they have been written. You flip the RESET switch once before you start, then examine each address one at a time and deposit the op-code or DATA that belongs there.

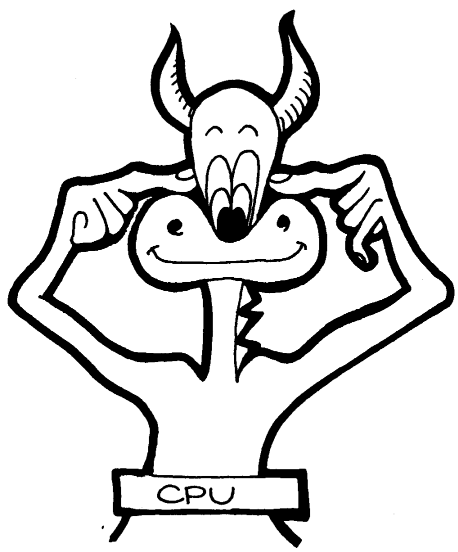
It's a good idea to get into the habit of checking each address for the correct op-code after the complete program has been stored in memory. Programming with switches is a tedious job, and no matter how careful you are, mistakes will creep in.

If you understand how the program should work, try it on your computer. Start at address 000,000 and put the 333 op-code there. Continue putting each op-code at its address until all sixteen are there. Then go back and examine address 000,000 and flip the RUN switch.

There will be a lot of A's being displayed on the terminal—your first program is running!

When you have had enough A's, flip the STOP switch.

How could you change one op-code to make the program print B's on the terminal instead of A's? Try it.



2
OUTPUT A MESSAGE

The previous chapter acquainted you with the terms and general operation of your computer. It contained a lot of information; hopefully, you understood its contents. In Chapter 2, we will write a very useful short program utilizing part of the program from Chapter 1. Its purpose is to display on the terminal a complete printed sentence. With this little program, you can display a message on the terminal rather than just a single letter.

Carriage Return and Line Feed

When DATA is sent to the terminal, it is normally in the form of ASCII DATA. There are many other ASCII DATA codes besides just the letters of the alphabet (see Appendix III).

Two of these are:

012, which is line feed and
015, which is carriage return

These two commands are needed when the terminal has printed an entire line of DATA. When a line has been filled on the screen (or paper) of the terminal, it must then be told to begin printing at the *beginning* of the *next* line. When using a normal typewriter, if you return the carriage it will automatically advance to the next line, but on a computer terminal, you must send two commands—one to return to the beginning of the line, and one to advance to the next line. In most cases it does not matter which command is sent first.

Here's the program;

ADDRESS	OP-CODE			EXPLANATION
000,000	041		LXI H/L	Load into the H and L registers
000,001	026			000,026.
000,002	000			
000,003	333	STATUS CK	IN	INPUT
000,004	000		STATUS	the STATUS byte.
000,005	346		ANI	AND the ACCUMULATOR
000,006	200			with 10 000 000.
000,007	302		JNZ	JUMP if NOT ZERO
000,010	003			to address 000,003.
000,011	000			
000,012	176		MOV A,M	MOVE the DATA at address H/L to A.
000,013	376		CPI	COMPARE the ACCUMULATOR
000,014	377			with 11 111 111.
000,015	312		JZ	JUMP if ZERO
000,016	015			to location 000,015.
000,017	000			
000,020	323		OUT	OUTPUT
000,021	001			DATA to the terminal.
000,022	043		INX H/L	INCREMENT the address H/L.
000,023	303		JMP	JUMP
000,024	003			to address 000,003.
000,025	000			
000,026	111			ASCII I
000,027	040			ASCII space
000,030	114			ASCII L
000,031	111			ASCII I
000,032	113			ASCII K
000,033	105			ASCII E
000,034	040			ASCII space
000,035	115			ASCII M
000,036	131			ASCII Y
000,037	040			ASCII space
000,040	103			ASCII C
000,041	117			ASCII O
000,042	115			ASCII M
000,043	120			ASCII P
000,044	125			ASCII U
000,045	124			ASCII T
000,046	105			ASCII E
000,047	122			ASCII R

000,050	012		line feed
000,051	015		carriage return
000,052	377		code for stop

Here's an explanation for each op-code:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	This instruction loads the register pair H and L with the two data bytes that follow.
000,001	026		026 is loaded into register L.
000,002	000		000 is loaded into register H.
000,003	333	IN	These seven instructions are the "output STATUS check" that you learned in Chapter 1. Its purpose is to find out if the terminal is ready to accept DATA.
000,004	000	STATUS	
000,005	346	ANI	
000,006	200		
000,007	302	JNZ	
000,010	003		
000,011	000		
000,012	176	MOV A,M	The MOVE instruction moves DATA from one place to another. "A" represents the ACCUMULATOR, and "M" represents the DATA at address H/L. H and L are registers and each register contains one byte, so if we consider H and L together as sixteen bits, they can describe an address: this address is labeled M. The 176 op-code MOVES the DATA at address M to the ACCUMULATOR.
000,013	376	CPI	The COMPARE IMMEDIATE instruction compares the following byte to the ACCUMULATOR.
000,014	377		11 111 111 is COMPARED to the ACCUMULATOR.
000,015	312	JZ	JUMP if ZERO. In the preceding paragraph, the ACCUMULATOR is compared with 377. If they are the same, the result will be zero
000,016	015		
000,017	000		

			and the CPU will jump to address 000,015. If they are different, then operation continues:
000,020	323	OUT	OUTPUT
000,021	001	DATA	DATA to the terminal.
000,022	043	INX H/L	INCREMENT H/L. The address M is increased by one.
000,023	303	JMP	JUMP
000,024	003		back to address 000,003.
000,025	000		

Here is a summary:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	Set up H and L registers with the starting address of the message.
000,001	026		
000,002	000		
000,003	333	IN	Check STATUS of the terminal. Keep looping here until the terminal is ready, then continue.
000,004	000	STATUS	
000,005	346	ANI	
000,006	200		
000,007	302	JNZ	
000,010	003		
000,011	000		
000,012	176	MOV A,M	Move the DATA stored at address H/L to the ACCUMULATOR.
000,013	376	CPI	Test to see if the last DATA byte (377) has been reached.
000,014	377		
000,015	312	JZ	If the result of this test is zero, the entire message has been printed and the program stays in this small loop.
000,016	015		
000,017	000		
000,020	323	OUT	If the result is not zero, the DATA byte is output to the terminal.
000,021	001		
000,022	043	INX H/L	The address H/L is increased by one.

000,023	303	JMP	Jump back to address 000,012.
000,024	012		
000,025	000		
000,026	111		ASCII I
000,027	040		ASCII space
000,030	114		ASCII L
000,031	111		ASCII I
000,032	113		ASCII K
000,033	105		ASCII E
000,034	040		ASCII space
000,035	115		ASCII M
000,036	131		ASCII Y
000,037	040		ASCII space
000,040	103		ASCII C
000,041	117		ASCII O
000,042	115		ASCII M
000,043	120		ASCII P
000,044	125		ASCII U
000,045	124		ASCII T
000,046	105		ASCII E
000,047	122		ASCII R
000,050	012		line feed
000,051	015		carriage return
000,052	377		stop code

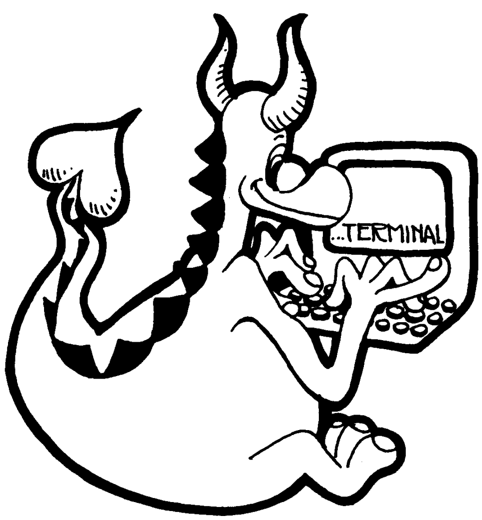
If you understand how the program should work, try running it on your computer. Store the op-codes at each address (000,000 through 000,052), then examine address 000,000 and run the program.

If all goes well, the program will print "I LIKE MY COMPUTER" on the terminal, and then appear to stop. Actually the computer is still running; it is continuously looping at addresses 000,015; 000,016; and 000,017. It will continue to do this until you flip the "stop" switch. This may seem wasteful, but for now it is the only way to stop the printout.

You must understand the idea that the computer will not do anything on its own; everything you want it to do must be specified in your program!

As an exercise, think of a different message and modify the program and make it print your message. It can be any length; just remember to store 377 at the end so the CPU will know when to stop printing it. Try it.

This program, as simple as it is, is probably the most useful one you will ever learn. You will use this as a part of almost every program you ever write. The message it prints does not need to be limited to only one sentence; you can print pages and pages of text with it and the terminal will happily display it all, stopping when it finds a 377.



3

THE INPUT SUBROUTINE

This chapter deals with the INPUT subroutine. Until now you have only sent data *to* the terminal, but in order for you to communicate freely with the computer, it must be able to get data from you. The INPUT routine is a short one:

ADDRESS		OP-CODE	
000,000		333	IN
000,001		000	STATUS
000,002		346	ANI
000,003	STATUS	001	00,000,001
000,004		302	JNZ
000,005		000	
000,006		000	
000,007		333	IN
000,010		001	DATA
000,011		323	OUT
000,012		001	DATA
000,013		303	JMP
000,014		000	
000,015		000	

Almost all major programs require some kind of response from the person at the terminal, so the INPUT subroutine is usually needed in any large program.

Look at the first seven steps. This is the STATUS test; it tells the CPU when the terminal is ready to input data. Now is a good time to glance back at pages 0 and 00.

Notice that the INPUT STATUS check is exactly like the OUTPUT STATUS check except that:

To INPUT, we check the RIGHT bit of the STATUS word.
TO OUTPUT, we check the LEFT bit of the STATUS word.

A one bit means that the terminal is not ready, a zero bit means it is ready*.

STATUS WORD

01100011	Terminal is ready to output but not input.
11100010	Terminal is ready to input but not output.
01100010	Terminal is ready to input or output.

There are no new op-codes in this program.

ADDRESS		OP-CODE		EXPLANATION
000,000		333	IN	Input
000,001		000	STATUS	the terminal STATUS byte.
000,002		346	ANI	AND the ACCUMULATOR
000,003	STATUS	001		with 00,000,001.
000,004		302	JNZ	Jump if not zero
000,005		000		to address 000,000.
000,006		000		
000,007		333	IN	Input
000,010		001	DATA	DATA from the terminal.
000,011		323	OUT	Output
000,012		001	DATA	DATA to the terminal.
000,013		303	JMP	Jump
000,014		000		to address 000,000.
000,015		000		

A summary: This program first checks the terminal input STATUS word until the right bit become zero. The rightmost bit of the STATUS word will remain a "one" until someone types a terminal key. The instant a key is typed, the right STATUS bit goes to a zero and the CPU moves on to address 000,007. The terminal DATA is input to the ACCUMULATOR. The ACCUMULATOR DATA is output to the terminal. The CPU then jumps back to address 000,000 to perform another STATUS check.

*This is one of those places where it might be different for your particular computer.

The program will display on the terminal anything that is typed on the terminal keyboard. It "echoes" what you type.

Have you noticed anything missing from the program? At address 000,011 DATA is output to the terminal—but we didn't check the terminal output STATUS. How do we know it is ready to accept DATA? The answer is simple enough. Your computer and terminal can process data surprisingly fast; usually each op-code you give it can be handled in less than 0.00001 seconds. If you typed as fast as you can on the keyboard, I'll bet you couldn't do better than one key every 0.1 seconds. When a key is typed, the computer completes the whole INPUT routine and outputs the data back to the terminal much faster than you can type the next key. So—since you are in effect setting the pace of data coming to the terminal, and since the terminal can take data much faster than you can type it, the terminal will *always* be ready to accept DATA no matter how fast you can type.

Load the program in memory starting at address 000,000 and run it. If it runs properly, whatever you type on the terminal will be displayed.

Take a look at this program:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	Load register pair H and L with address 000,100.
000,001	100		
000,002	000		
000,003	333	IN	Input STATUS check.
000,004	000	STATUS	
000,005	346	ANI	
000,006	001		
000,007	302	JNZ	
000,010	003		
000,011	000		
000,012	333	IN	Input to the ACCUMULATOR DATA from the terminal.
000,013	001	DATA	
000,014	167	MOV M,A	Move the DATA in the ACCUMULATOR to the memory address H/L. (M)
000,015	323	OUT	Output from the ACCUMULATOR to the terminal.
000,016	001	DATA	
000,017	043	INX H/L	Increment address H/L

000,020	303	JMP	Jump
000,021	003		back to address 000,003.
000,022	000		

This one echoes what you type on the terminal just as before; but it also stores what you type in memory address 000,100 and on up. If you type "HELLO" on the terminal, memory would contain:

000,100	110	ASCII H
000,101	105	ASCII E
000,102	114	ASCII L
000,103	114	ASCII L
000,104	117	ASCII O

4
THE RANDOM NUMBER GENERATOR

Computers are very precise machines. They do exactly what they are told to do—which means they never do anything on their own. So how do you get a computer to shuffle an imaginary deck of cards; or how would a computer pick a number?

The answer is a small program called a "random number generator (RND). No random number generator *really* picks random numbers, but it will put out a string of numbers which looks random at first glance. For instance:

21007564331072100756433107 . . . .

As you can see by close inspection, the number chain does repeat itself, but if you didn't have the numbers printed here before you, you would have a hard time trying to guess what comes after 6.

A random number generator takes one number at a time out of a string of numbers and puts it into the ACCUMULATOR. With this new tool, you can effectively ask the computer to "pick a number."

The random number generator chosen here is a short program, but it contains four new op-codes which might need explaining:

RRC Rotate the ACCUMULATOR right. This instruction will cause the ACCUMULATOR byte to be shifted one bit to the right.

If ACCUMULATOR byte is	00,101,001
then after RRC it will be	10,010,100

ADDM Add register M to the ACCUMULATOR. Remember how registers H and L can be paired to form an address? If H contains 00,000,000 and L contains 00,000,100 then the address H/L is 00,000, 000 00,000,100. The DATA byte at this address is called register M.

The ADD M instruction adds the ACCUMULATOR byte to register M and puts the sum back in the ACCUMULATOR:

If ACCUMULATOR byte is	10,111,010
and DATA byte at address H/L is	01,000,010
then after ADD M,	
ACCUMULATOR will be	11,111,100

XRAM Exclusive-OR register M with the ACCUMULATOR. The DATA byte at address H/L is Exclusive-ORed with the ACCUMULATOR, and the result is put in the ACCUMULATOR:

If ACCUMULATOR byte is	11,001,100
and DATA byte	
at address H/L is	00,101,101
then after XRAM,	
ACCUMULATOR will be	11,100,001

ORI OR the ACCUMULATOR with DATA byte 00,110,000. The ORI op-code is always followed by a DATA byte; the byte is ORed with the ACCUMULATOR and the result is put in the ACCUMULATOR:

If the ACCUMULATOR byte is	00,000,100
then after ORI 060	00,110,000
ACCUMULATOR will be	00,110,100

We turned a binary four (00,000,100) into an ASCII four (00,110,100).

This program randomly picks a number from 0 to 7 and outputs it to the terminal:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	Load registers H and L with 000,024.
000,001	024		
000,002	000		
000,003	176	MOV A,M	Move the DATA byte at address H/L to the ACCUMULATOR.
000,004	017	RRC	Rotate the ACCUMULATOR byte one bit to the right.
000,005	206	ADD M	Add ACCUMULATOR byte to register M (register M is the DATA at address H/L).
000,006	017	RRC	Rotate ACCUMULATOR right again.
000,007	167	MOV M,A	Move the ACCUMULATOR byte to the address H/L (register M).
000,010	043	INX H/L	Address H/L is incremented by one.
000,011	256	XRA M	Register M is Exclusive ORed with the ACCUMULATOR.
000,012	167	MOV M,A	Move the ACCUMULATOR byte to the address H/L.
000,013	346	ANI	AND the ACCUMULATOR with 00,000,111.
000,014	007		
000,015	366	ORI	OR the ACCUMULATOR with 00,110,000.
000,016	060		
000,017	323	OUT	Output the ACCUMULATOR to the terminal.
000,020	001		
000,021	303	JMP	Jump back to address 000,021 and loop here.
000,022	021		
000,023	000		

000,024	XXX	DATA	These two locations are used to
000,025	XXX	DATA	temporarily store DATA during the program.

In summary:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	The DATA byte at address
000,001	024		000,024 is moved into the ACCUMULATOR.
000,002	000		
000,003	176	MOV A,M	
000,004	017	RRC	
000,005	206	ADD M	The ACCUMULATOR DATA is changed,
000,006	017	RRC	then put back at address 000,024.
000,007	167	MOV M,A	
000,010	043	INX H/L	The DATA byte at 000,025 is
000,011	256	XRA M	Exclusive ORed with ACCUMULATOR,
000,012	167	MOV M,A	then it's put back at 000,025.
000,013	346	ANI	ACCUMULATOR is ANDed with 00,000,111
000,014	007		(so it can never be larger than seven).
000,015	366	ORI	The ACCUMULATOR, which
000,016	060		is now a number from 000 to 007, is made into an ASCII number from 060 to 067.
000,017	323	OUT	Output the ASCII number
000,020	001		to the terminal.
000,021	303	JMP	Loop here.
000,022	021		
000,023	000		

Assume address 000,024 contains DATA byte 022 (00,010,010), and address 000,025 contains DATA byte 001 (00,000,001). Now take a piece of scratch paper and pencil, work the program on paper, and find out what ASCII number will be output to the terminal.

The answer is ASCII 064, which is the number "4."

Load the program into the computer, store 022 at address 000,024 and store 001 at 000,025. Then start at address 000,000 and run the program—you should get a "4" on the terminal. Now stop the computer, start at 000,000 and run again. This time you will get a different number between 0 and 7.

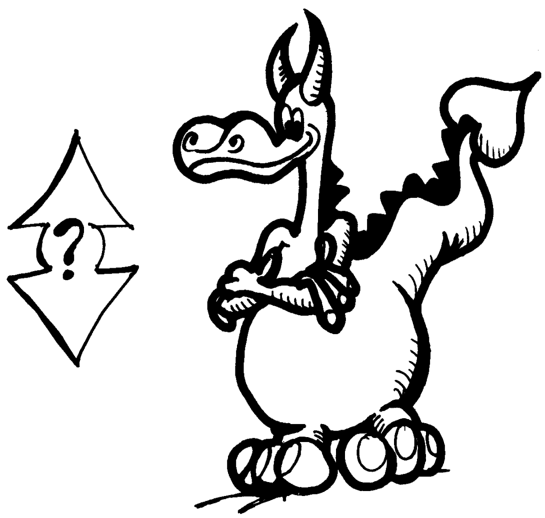
Each time the program runs, it changes the data at address 000,024 and 000,025. Since the ASCII number that shows up on the terminal depends on the data in these two addresses, a different number is output each time.

The random number generator has many uses. You can use it to generate a binary number by omitting the "ORI 060" instruction.

Modify the program so that it will generate an ASCII number from 060 to 063.

Now make it generate a binary number from 000 to 003.

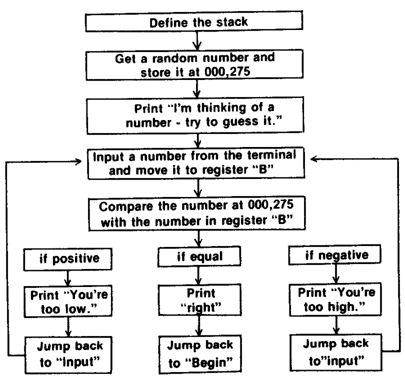
Instead of manually stopping and starting the computer each time, how could you change the program to make it print out a whole string of random numbers?



5

HI-LO

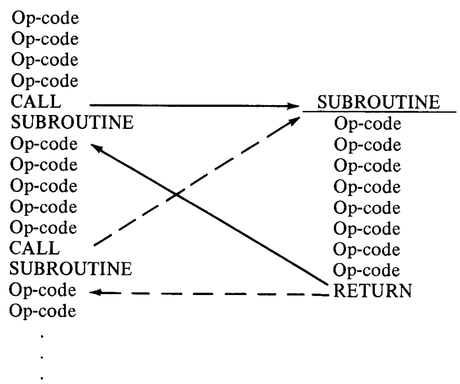
This section explains the program to play the game HI-LOW. It's a number guessing game in which the computer picks a number from 0 to 7 and you try to guess it. The computer tells you if your guess is too high or too low. The game continues until you guess the number. Here is a diagram showing the modular parts of the program:



Each block in the diagram represents a subroutine; you should recognize most of them. The only new op-codes needed to combine them into a game program are the CALL and RETURN instructions, and you will need to know about the STACK.

The new idea here is that we're going to write a main program which uses a smaller subroutine.

MAIN PROGRAM



The MAIN PROGRAM calls the SUBROUTINE. After the SUBROUTINE has been completed, the MAIN PROGRAM continues at the next op-code. The SUBROUTINE can be called as many times as needed, and each time it will return to the MAIN PROGRAM where it left off.

THE STACK

The STACK is a section of memory where DATA is temporarily stored. This section of memory is *defined by you*. To locate the STACK in memory, you use the instruction LXI STACK POINTER or LXI SP. The op-code is 061. The LXI SP instruction will define the highest (or first) STACK address and from then on, as more DATA is put into the STACK, the DATA will be stored in consecutive lower addresses.

For example: 061 LXI SP
100
000

ADDRESS		
.		
.		
.		
000,074		
000,075		
000,076		For this op-code,
000,077		the top of the stack would be
000,100		here

ADDRESS	DATA	
.		
.		
.		
000,074		
000,075		If an address 000,033 is pushed
000,076	033	onto the STACK, it will be
000,077	000	stored like this
000,100		

As DATA is pushed onto the STACK the STACK pointer shifts down in memory. When address 000,033 is moved onto the STACK first, DATA 033 is pushed into address 000,076; then DATA 000 is pushed into address 000,077, and the STACK pointer is reduced by two—the STACK pointer is now 000,076.

ADDRESS	DATA	
.		
.		Now if address 000,034 is pushed
.		onto the STACK, it will be stored
000,074	034	like this
000,075	000	
000,076	033	
000,077	000	
000,100		and the STACK pointer will then be 000,074

Notice that each time DATA is pushed onto the STACK, the top of the STACK shifts down. Therefore, the STACK has no set size—the more DATA put into it, the larger it gets.

When DATA is taken off the STACK ("popped" off), it comes off in reverse order. In the example just given, address 000,034 would pop off first, then 000,033.

Getting back to the CALL and RETURN op-codes—when the CPU sees the CALL instruction, it stores the next op-code *address* in the STACK before going to the subroutine that was called. After completing the subroutine, the CPU sees the RETURN op-code, so it looks into the STACK for the return address. The program then continues from that address.

When a program will need the STACK, the programmer must define where the STACK will be located in memory. Usually, the *first* op-code in any main program of any length is LXI SP. Always be sure that the memory area you set aside for the STACK is not being used for anything else—make sure it's empty memory. For most small programs, ten memory addresses are enough for the STACK.

The STACK is a handy place in which the CPU can store ADDRESSES temporarily. There are two op-codes which will allow *you* to put DATA onto the STACK and pull the DATA back off: PUSH and POP. I won't be using the PUSH or POP instructions in this book, but keep in mind (for future programming) that you can store DATA temporarily in the STACK.

Here's the game program. There is a subroutine at address 000,017, so a STACK is needed. Carefully study the program (it's a long one), and compare it to the block diagram on page 33.

ADDRESS	OP-CODE		EXPLANATION
000,000	061	LXI SP	Define the top of the STACK as address 000,300.
000,001	300		
000,002	000		
000,003	041	LXI H/L	Get a random number between 0 and 7.
000,004	273		
000,005	000		
000,006	176	MOV A,M	
000,007	017	RRC	
000,010	206	ADDR M	
000,011	017	RRC	
000,012	167	MOV M,A	
000,013	043	INX H/L	
000,014	256	XRA M	
000,015	167	MOV M,A	
000,016	346	ANI	
000,017	007		
000,020	366	ORI	
000,021	060		

000,022	062	STA	Store the random number at address 000,275.
000,023	275		
000,024	000		
000,025	041	LXI H/L	Print "HI-LOW, I'm thinking of a number from 0 to 7 . . ."
000,026	140		
000,027	000		
000,030	315	CALL	
000,031	117		
000,032	000		
000,033	333	IN	This INPUTS a number from the keyboard into the ACCUMULATOR and echoes the number on the terminal. It's exactly the same routine you learned in Chapter 3.
000,034	000	STATUS	
000,035	346	ANI	
000,036	001		
000,037	302	JNZ	
000,040	033		
000,041	000		
000,042	333	IN	
000,043	001	DATA	
000,044	323	OUT	
000,045	001	DATA	
000,046	107	MOV B,A	Move ACCUMULATOR to register B.
000,047	072	LDA	Load the ACCUMULATOR with the number at address 000,275 and compare it to the number in register B.
000,050	275		
000,051	000		
000,052	270	CMP B	
000,053	312	JZ	Jump if the result is zero to address 000,064.
000,054	064		
000,055	000		
000,056	362	JP	Jump if the result is positive to address 000,075.
000,057	075		
000,060	000		
000,061	372	JM	Jump if the result is minus to address 000,106.
000,062	106		
000,063	000		

000,064	041	LXI H/L	Print "You got it!!"
000,065	253		
000,066	000		
000,067	315	CALL	
000,070	117		
000,071	000		
000,072	303	JMP	Jump back to the beginning.
000,073	000		
000,074	000		
000,075	041	LXI H/L	Print "Too low."
000,076	240		
000,077	000		
000,100	315	CALL	
000,101	117		
000,102	000		
000,103	303	JMP	Jump back to INPUT.
000,104	033		
000,105	000		
000,106	041	LXI H/L	Print "Too high."
000,107	225		
000,110	000		
000,111	315	CALL	
000,112	117		
000,113	000		
000,114	303	JMP	Jump back to INPUT.
000,115	033		
000,116	000		
000,117	333	IN	This SUBROUTINE prints a message out on the terminal.
000,120	000	STATUS	
000,121	346	ANI	It starts with the ASCII DATA at address H/L and prints each DATA byte one letter at a time until it gets to 377 (stop code). Then it returns to the main program.
000,122	200		
000,123	302	JNZ	
000,124	117		
000,125	000		
000,126	176	MOV A,M	
000,127	376	CPI	

000,130	377		
000,131	310	RZ	This "PRINT" routine is exactly like the one you learned in Chapter 2.
000,132	323	OUT	
000,133	001	DATA	
000,134	043	INX H/L	
000,135	303		
000,136	117		
000,137	000		

ADDRESS	OP-CODE	EXPLANATION
000,140	110	ASCII H
000,141	111	ASCII I
000,142	040	ASCII space
000,143	114	ASCII L
000,144	117	ASCII O
000,145	127	ASCII W
000,146	012	ASCII line feed
000,147	015	ASCII carriage return
000,150	111	ASCII I
000,151	047	ASCII '
000,152	115	ASCII M
000,153	040	ASCII space
000,154	124	ASCII T
000,155	110	ASCII H
000,156	111	ASCII I
000,157	116	ASCII N
000,160	113	ASCII K
000,161	111	ASCII I
000,162	116	ASCII N
000,163	107	ASCII G
000,164	040	ASCII space
000,165	117	ASCII O
000,166	106	ASCII F
000,167	040	ASCII space
000,170	101	ASCII A
000,171	040	ASCII space
000,172	116	ASCII N
000,173	125	ASCII U
000,174	115	ASCII M
000,175	102	ASCII B
000,176	105	ASCII E
000,177	122	ASCII R
000,200	012	ASCII line feed

000,201	015	ASCII carriage return
000,202	124	ASCII T
000,203	122	ASCII R
000,204	131	ASCII Y
000,205	040	ASCII space
000,206	124	ASCII T
000,207	117	ASCII O
000,210	040	ASCII space
000,211	107	ASCII G
000,212	125	ASCII U
000,213	105	ASCII E
000,214	123	ASCII S
000,215	123	ASCII S
000,216	040	ASCII space
000,217	111	ASCII I
000,220	124	ASCII T
000,221	072	ASCII :
000,222	012	ASCII line feed
000,223	015	ASCII carriage return
000,224	377	stop code
000,225	072	ASCII :
000,226	124	ASCII T
000,227	117	ASCII O
000,230	117	ASCII O
000,231	040	ASCII space
000,232	110	ASCII H
000,233	111	ASCII I
000,234	056	ASCII .
000,235	012	ASCII line feed
000,236	015	ASCII carriage return
000,237	377	stop code
000,240	072	ASCII :
000,241	124	ASCII T
000,242	117	ASCII O
000,243	117	ASCII O
000,244	040	ASCII space
000,245	114	ASCII L
000,246	117	ASCII O
000,247	056	ASCII .
000,250	012	ASCII line feed
000,251	015	ASCII carriage return
000,252	377	stop code

000,253	072		ASCII :
000,254	131		ASCII Y
000,255	117		ASCII O
000,256	125		ASCII U
000,257	040		ASCII space
000,260	107		ASCII G
000,261	117		ASCII O
000,262	124		ASCII T
000,263	040		ASCII space
000,264	111		ASCII I
000,265	124		ASCII T
000,266	041		ASCII !
000,267	012		ASCII line feed
000,270	015		ASCII carriage return
000,271	012		ASCII line feed
000,272	377		stop code
000,273	213	RND 1	Used for random number generator
000,274	115	RND 2	Used for random number generator
000,275	000	STORE	Used for temporary storage
000,276	000	STACK	
000,277	000	STACK	
000,300	000	STACK	

When an ASCII carriage return and line feed (015 and 012) are output to the terminal, it causes the print head to start printing at the *beginning* of the *next* line.

This program is a little longer than the routines we have given up to now, but if you study it one part at a time, I think you can understand how it works. Note that over half the program is ASCII data to be printed out. Many times, especially in game programs, the ASCII data takes up more memory space than the program itself.

Run the program.

This is a simple game, but you have learned how to program the computer to "talk" with you—take your data from the keyboard and respond accordingly. This simple game contains the basic building blocks of INPUT, INTERACTION, and OUTPUT that are used in even the most complex programs.

6
NIM

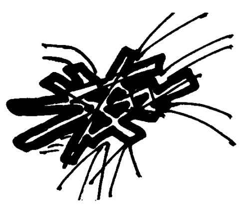
The game of NIM is one of the oldest games known to man. NIM has many variations, but we are going to write a program to play the following version:

There are fifteen sticks in a pile.
Each player, on his turn, may remove one, two or three sticks from the pile.
The player who takes the last stick loses.

The computer will go first, and because of this the computer will always win—but don't get discouraged, it's fun to play the game and it's a good lesson in programming.

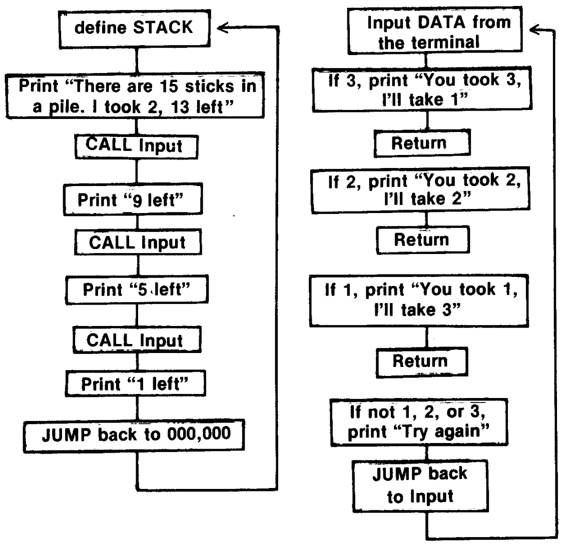
The logic to make the computer win the game has already been figured out for you: Basically, the computer looks at how many sticks you took from the pile, subtracts that number from 4, and takes the difference. If you take one, it will take three; if you take two, it will take two; if you take three, it will take one.

Take a look at the block diagram of the game.



NIM

MAIN PROGRAM **INPUT SUBROUTINE**



This program contains no new op-codes.

ADDRESS	OP-CODE		EXPLANATION
000,000	061	LXI SP	Define top of STACK.
000,001	200		
000,002	001		
000,003	041	LXI H/L	
000,004	160		
000,005	000		Print "There are 15 sticks in
000,006	315	CALL	a pile; I'll take 2, 13 left."
000,007	137		
000,010	000		

000,011	315	CALL	Call INPUT.
000,012	047		
000,013	000		
000,014	041	LXI H/L	Print "9 left."
000,015	100		
000,016	001		
000,017	315	CALL	
000,020	137		
000,021	000		
000,022	315	CALL	Call INPUT.
000,023	047		
000,024	000		
000,025	041	LXI H/L	Print "5 left."
000,026	112		
000,027	001		
000,030	315	CALL	
000,031	137		
000,032	000		
000,033	315	CALL	Call INPUT.
000,034	047		
000,035	000		
000,036	041	LXI H/L	Print "You lose . . . 1 left."
000,037	124		
000,040	001		
000,041	315	CALL	
000,042	137		
000,043	000		
000,044	303	JMP	Jump back to the beginning.
000,045	000		
000,046	000		
000,047	333	IN	Input DATA from the terminal and echo it back to the terminal.
000,050	000	STATUS	
000,051	346	ANI	
000,052	001		
000,053	302	JNZ	
000,054	047		
000,055	000		
000,056	333	IN	

000,057	001	DATA	
000,060	323	OUT	
000,061	001	DATA	
000,062	376	CPI	If 3, jump to address 000,112.
000,063	063	ASCII 3	
000,064	312	JZ	
000,065	112		
000,066	000		
000,067	376	CPI	If 2, jump to address 000,121.
000,070	062	ASCII 2	
000,071	312	JZ	
000,072	121		
000,073	000		
000,074	376	CPI	If 1, jump to address 000,130.
000,075	061	ASCII 1	
000,076	312	JZ	
000,077	130		
000,100	000		
000,101	041	LXI H/L	If not 1, 2, or 3, print "Try again."
000,102	042		
000,103	001		
000,104	315	CALL	
000,105	137		
000,106	000		
000,107	303	JMP	Jump back to input.
000,110	047		
000,111	000		
000,112	041	LXI H/L	Print "You took 3, I'll take 1."
000,113	272		
000,114	000		
000,115	315	CALL	
000,116	137		
000,117	000		
000,120	311	RET	
000,121	041	LXI H/L	
000,122	334		
000,123	000		

000,124	315	CALL	Print "You took 2, I'll take 2."
000,125	137		
000,126	000		
000,127	311	RET	
000,130	041	LXI H/L	
000,131	377		
000,132	000		
000,133	315	CALL	Print "You took 1, I'll take 3."
000,134	137		
000,135	000		
000,136	311	RET	
000,137	333	IN	
000,140	000	STATUS	
000,141	346	ANI	
000,142	200		
000,143	302	JNZ	
000,144	137		
000,145	000		This subroutine prints DATA out to the terminal starting at H/L and ending at the STOP CODE 377.
000,146	176	MOV A,M	
000,147	376	CPI	
000,150	377	STP CODE	
000,151	310	RZ	
000,152	323	OUT	
000,153	001	DATA	
000,154	043	INX H/L	
000,155	303	JMP	
000,156	137		
000,157	000		
000,160	012		line feed
000,161	015		carriage return
000,162	124		ASCII T
000,163	110		ASCII H
000,164	105		ASCII E
000,165	122		ASCII R
000,166	105		ASCII E
000,167	040		space
000,170	101		ASCII A
000,171	122		ASCII R
000,172	105		ASCII E
000,173	040		space
000,174	061		ASCII 1

000,175	065	ASCII 5
000,176	040	space
000,177	123	ASCII S
000,200	124	ASCII T
000,201	111	ASCII I
000,202	103	ASCII C
000,203	113	ASCII K
000,204	123	ASCII S
000,205	040	space
000,206	111	ASCII I
000,207	116	ASCII N
000,210	040	space
000,211	101	ASCII A
000,212	040	space
000,213	120	ASCII P
000,214	111	ASCII I
000,215	114	ASCII L
000,216	105	ASCII E
000,217	054	ASCII ,
000,220	012	line feed
000,221	015	carriage return
000,222	111	ASCII I
000,223	047	ASCII '
000,224	114	ASCII L
000,225	114	ASCII L
000,226	040	space
000,227	124	ASCII T
000,230	101	ASCII A
000,231	113	ASCII K
000,232	105	ASCII E
000,233	040	space
000,234	062	ASCII 2
000,235	073	ASCII ;
000,236	040	space
000,237	061	ASCII 1
000,240	063	ASCII 3
000,241	040	space
000,242	114	ASCII L
000,243	105	ASCII E
000,244	106	ASCII F
000,245	124	ASCII T
000,246	056	ASCII .
000,247	012	line feed
000,250	015	carriage return

000,251	131	ASCII Y
000,252	117	ASCII O
000,253	125	ASCII U
000,254	122	ASCII R
000,255	040	space
000,256	124	ASCII T
000,257	125	ASCII U
000,260	122	ASCII R
000,261	116	ASCII N
000,262	056	ASCII .
000,263	056	ASCII .
000,264	056	ASCII .
000,265	056	ASCII .
000,266	056	ASCII .
000,267	012	line feed
000,270	015	carriage return
000,271	377	stop code
000,272	072	ASCII :
000,273	040	space
000,274	040	space
000,275	131	ASCII Y
000,276	117	ASCII O
000,277	125	ASCII U
000,300	040	space
000,301	124	ASCII T
000,302	117	ASCII O
000,303	117	ASCII O
000,304	113	ASCII K
000,305	040	space
000,306	063	ASCII 3
000,307	073	ASCII ;
000,310	012	line feed
000,311	015	carriage return
000,312	111	ASCII I
000,313	047	ASCII '
000,314	114	ASCII L
000,315	114	ASCII L
000,316	040	space
000,317	124	ASCII T
000,320	101	ASCII A
000,321	113	ASCII K
000,322	105	ASCII E
000,323	040	space

000,324	061	ASCII 1
000,325	056	ASCII .
000,326	056	ASCII .
000,327	056	ASCII .
000,330	056	ASCII .
000,331	012	line feed
000,332	015	carriage return
000,333	377	stop code
000,334	072	ASCII :
000,335	040	space
000,336	040	space
000,337	131	ASCII Y
000,340	117	ASCII O
000,341	125	ASCII U
000,342	040	space
000,343	124	ASCII T
000,344	117	ASCII O
000,345	117	ASCII O
000,346	113	ASCII K
000,347	040	space
000,350	062	ASCII 2
000,351	073	ASCII ;
000,352	012	line feed
000,353	015	carriage return
000,354	111	ASCII I
000,355	047	ASCII '
000,356	114	ASCII L
000,357	114	ASCII L
000,360	040	space
000,361	124	ASCII T
000,362	101	ASCII A
000,363	113	ASCII K
000,364	105	ASCII E
000,365	040	space
000,366	062	ASCII 2
000,367	056	ASCII .
000,370	056	ASCII .
000,371	056	ASCII .
000,372	056	ASCII .
000,373	056	ASCII .
000,374	012	line feed
000,375	015	carriage return
000,376	377	stop code
000,377	072	ASCII :

001,000	040	space
001,001	040	space
001,002	131	ASCII Y
001,003	117	ASCII O
001,004	125	ASCII U
001,005	040	space
001,006	124	ASCII T
001,007	117	ASCII O
001,010	117	ASCII O
001,011	113	ASCII K
001,012	040	space
001,013	061	ASCII 1
001,014	073	ASCII ;
001,015	012	line feed
001,016	015	carriage return
001,017	111	ASCII I
001,020	047	ASCII '
001,021	114	ASCII L
001,022	114	ASCII L
001,023	040	space
001,024	124	ASCII T
001,025	101	ASCII A
001,026	113	ASCII K
001,027	105	ASCII E
001,030	040	space
001,031	063	ASCII 3
001,032	056	ASCII .
001,033	056	ASCII .
001,034	056	ASCII .
001,035	056	ASCII .
001,036	056	ASCII .
001,037	012	line feed
001,040	015	carriage return
001,041	377	stop code
001,042	012	line feed
001,043	015	carriage return
001,044	131	ASCII Y
001,045	117	ASCII O
001,046	125	ASCII U
001,047	040	space
001,050	115	ASCII M
001,051	125	ASCII U
001,052	123	ASCII S
001,053	124	ASCII T
001,054	040	space

001,055	124	ASCII T
001,056	101	ASCII A
001,057	113	ASCII K
001,060	105	ASCII E
001,061	040	space
001,062	061	ASCII 1
001,063	054	ASCII ,
001,064	040	space
001,065	062	ASCII 2
001,066	054	ASCII ,
001,067	040	space
001,070	117	ASCII O
001,071	122	ASCII R
001,072	040	space
001,073	063	ASCII 3
001,074	012	line feed
001,075	012	line feed
001,076	015	carriage return
001,077	377	stop code
001,100	071	ASCII 9
001,101	040	space
001,102	114	ASCII L
001,103	105	ASCII E
001,104	106	ASCII F
001,105	124	ASCII T
001,106	012	line feed
001,107	012	line feed
001,110	015	carriage return
001,111	377	stop code
001,112	065	ASCII 5
001,113	040	space
001,114	114	ASCII L
001,115	105	ASCII E
001,116	106	ASCII F
001,117	124	ASCII T
001,120	012	line feed
001,121	012	line feed
001,122	015	carriage return
001,123	377	stop code
001,124	061	ASCII 1
001,125	040	space

001,126	114	ASCII L
001,127	105	ASCII E
001,130	106	ASCII F
001,131	124	ASCII T
001,132	055	ASCII –
001,133	055	ASCII –
001,134	055	ASCII –
001,135	131	ASCII Y
001,136	117	ASCII O
001,137	125	ASCII U
001,140	040	space
001,141	114	ASCII L
001,142	117	ASCII O
001,143	123	ASCII S
001,144	105	ASCII E
001,145	012	line feed
001,146	015	carriage return
001,147	124	ASCII T
001,150	122	ASCII R
001,151	131	ASCII Y
001,152	040	space
001,153	101	ASCII A
001,154	107	ASCII G
001,155	101	ASCII A
001,156	111	ASCII I
001,157	116	ASCII N
001,160	072	ASCII :
001,161	012	line feed
001,162	012	line feed
001,163	015	carriage return
001,164	377	stop code

Notice that in this game, just as in HI-LOW, the ASCII text is the major part of the program.

Run the program.

A good exercise, if you have time, would be to rewrite the program so that there are eighteen sticks in the pile to start with—the computer takes one stick and the play continues from there.


©1977

7
BUTTON-BUTTON

This final game program is called BUTTON-BUTTON. The game can actually be played without a computer, but as an interactive game with the computer it's a good one.

Eight people sit in a circle with you in the center:

```
          0
      7       1

    6    you    2

      5       3
          4
```

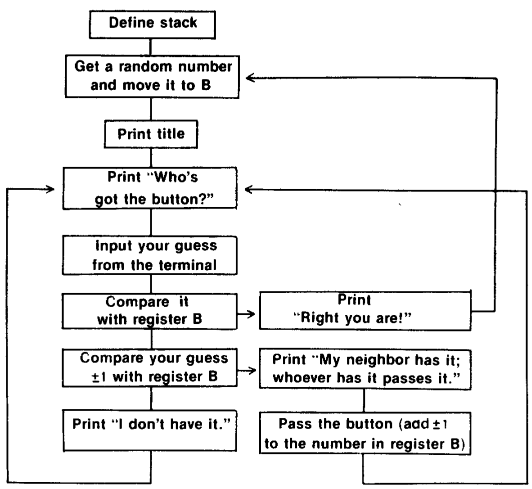
One of them has a button hidden in his hand and your job is to guess who has it. It's harder than you might think because the person with the button will sometimes pass it.

THE PROGRAM

The computer picks a random number *from 0 to 7* and stores it in register B. If random number 3 is chosen, then person 3 "has the button." If the button gets passed to another person, then the computer adds or subtracts one from register B—so either person 2 or person 4 has the button now. When the button gets passed, half the time the computer will add one to register B and half the time it will subtract one from register B. This is accomplished by getting another random number—if the number is 3, 2, 1, or 0, the computer adds one to register B; if the number if 4, 5, 6, or 7, the computer subtracts one from register B.

You will notice the op-code ANI 007 several times in the program. Here's the reason: If person 7 has the button, then register B contains 00,000,111. Now if the button gets passed, the computer adds or subtracts one from register B. If it subtracts one, then register B contains 00,000,110, which is person 6. But if the computer adds one to register B, then B contains 00,001,000, which is 8! We want it to contain 00,000,000 because next to person 7 sits person 0. ANDing register B with 00,000,111 will ensure that it always contains a number from 0 to 7.

BUTTON-BUTTON BLOCK DIAGRAM



THE NOP

The only new op-code in this program is NOP, [000]. The instruction means "no operation"; when used in a program, it does absolutely nothing. When the CPU finds a NOP in the program, it skips over it and continues with the next instruction. The value of the NOP instruction is in *writing* programs—when you start writing programs for yourself, you should include NOP op-codes spaced throughout the program so that if changes are needed after the program is finished, there will be room.

I've included two NOP's in BUTTON-BUTTON to show that they don't affect the operation of the program; when you begin writing programs, be sure to throw in a generous number of NOP's (usually in groups of two or three).

The value of the NOP will become very clear if you ever write a long program and then discover you need to add an op-code somewhere in the middle.

ADDRESS	OP-CODE		EXPLANATION
000,000	061	LXI SP	Define the STACK.
000,001	200		
000,002	001		
000,003	315	CALL	Get a random number and move it to register B.
000,004	202		
000,005	000		
000,006	107	MOV B,A	
000,007	041	LXI H/L	Print "BUTTON-BUTTON, who's got the button?"
000,010	220		
000,011	000		
000,012	315	CALL	
000,013	161		
000,014	000		
000,015	315	CALL	Input a number from the terminal.
000,016	125		
000,017	000		
000,020	270	CMP B	Compare it with register B.
000,021	312	JZ	If they're the same, jump to 000,114.
000,022	114		
000,023	000		

000,024	074	INR A	This checks to see if the neighbor has the button. We add 1 to the player's guess and compare it with register B—if they compare, then Jump to address 000,054
000,025	346	ANI	
000,026	007		
000,027	270	CMP B	
000,030	312	JZ	
000,031	054		
000,032	000		
000,033	075	DCR A	We added 1 to the player's guess above, so now we subtract 2 from it and compare that to register B. Again, if they compare, Jump to 000,054.
000,034	075	DCR A	
000,035	346	ANI	
000,036	007		
000,037	270	CMP B	
000,040	312	JZ	
000,041	054		
000,042	000		(These two routines take the player's guess, the ACCUMULATOR, and compare his two neighbors with register B.)
000,043	041	LXI H/L	Print "I don't have it; whoever has it, keeps it."
000,044	274		
000,045	000		
000,046	315	CALL	
000,047	161		
000,050	000		
000,051	303	JMP	Jump back to 000,007.
000,052	007		
000,053	000		
000,054	041	LXI H/L	Print "My neighbor has it; whoever has it, passes it."
000,055	363		
000,056	000		
000,057	315	CALL	
000,060	161		
000,061	000		
000,062	315	CALL	This routine passes the button to one of the neighbors—in other words, it adds or subtracts 1 from register B. Half the time it adds 1, and half the time is subtracts 1. It does this by CALLing the random
000,063	202		
000,064	000		
000,065	376	CPI	
000,066	003		
000,067	362	JP	
000,070	102		

000,071	000		number generator which puts a number from 0 to 7 in the ACCUMULATOR. It then compares the number with 3. Half the time the result will be positive, so it Jumps to 000,102, which is the routine that adds 1 to register B; otherwise it subtracts 1.
000,072	170	MOV A,B	
000,073	075	DCR A	
000,074	346	ANI	
000,075	007		
000,076	107	MOV B,A	
000,077	303	JMP	Jump back to address 000,007
000,100	007		
000,101	000		
000,102	170	MOV A,B	This routine adds 1 to register B.
000,103	074	INR A	
000,104	346	ANI	
000,105	007		
000,106	107	MOV B,A	
000,107	303	JMP	Jump back to address 000,007.
000,110	007		
000,111	000		
000,112	000	NOP	No operation.
000,113	000	NOP	No operation.
000,114	041	LXI H/L	Print "Right you are!"
000,115	070		
000,116	001		
000,117	315	CALL	
000,120	161		
000,121	000		
000,122	303	JMP	Jump back to the beginning.
000,123	000		
000,124	000		
000,125	333	IN	Input a number into the ACCUMULATOR.
000,126	000	STATUS	
000,127	346	ANI	
000,130	001		
000,131	302	JNZ	

000,132	125		
000,133	000		This is the INPUT subroutine which takes a number from the keyboard and echoes it back out on the terminal.
000,134	333	IN	
000,135	001	DATA	
000,136	323	OUT	
000,137	001	DATA	
000,140	376	CPI	The number input from the terminal can't be larger than 7, so the ACCUMULATOR is compared with 8; if the num- is 7 or smaller, then the CPU jumps to 000,156.
000,141	070	ASCII 8	
000,142	372	JM	
000,143	156		
000,144	000		
000,145	041	LXI H/L	
000,146	122		
000,147	001		
000,150	315	CALL	If the number is 8 or larger, Print "There's no one here with that number."
000,151	161		
000,152	000		
000,153	303	JMP	Then Jump back to Input (000,125).
000,154	125		
000,155	000		
000,156	346	ANI	If the number is 7 or smaller, AND it with 00,000,111. then Return.
000,157	007		
000,160	311	RET	
000,161	333	IN	
000,162	000	STATUS	
000,163	346	ANI	
000,164	200		
000,165	302	JNZ	
000,166	161		
000,167	000		
000,170	176	MOV A,M	
000,171	376	CPI	Print subroutine.
000,172	377	STOP CODE	
000,173	310	RZ	
000,174	323	OUT	
000,175	001	DATA	
000,176	043	INX H/L	
000,177	303		
000,200	161		
000,201	000		

000,202	041	LXI H/L	Random number generator subroutine.
000,203	170		
000,204	001		
000,205	176	MOV A,M	
000,206	017	RRC	
000,207	206	ADDR M	
000,210	017	RRC	
000,211	167	MOV M,A	
000,212	043	INX H/L	
000,213	256	XRA M	
000,214	167	MOV M,A	
000,215	346	ANI	
000,216	007		
000,217	311	RET	
000,220	012		line feed
000,221	012		line feed
000,222	015		carriage return
000,223	052		ASCII *
000,224	102		ASCII B
000,225	125		ASCII U
000,226	124		ASCII T
000,227	124		ASCII T
000,230	117		ASCII O
000,231	116		ASCII N
000,232	055		ASCII –
000,233	102		ASCII B
000,234	125		ASCII U
000,235	124		ASCII T
000,236	124		ASCII T
000,237	117		ASCII O
000,240	116		ASCII N
000,241	052		ASCII *
000,242	012		line feed
000,243	015		carriage return
000,244	127		ASCII W
000,245	110		ASCII H
000,246	117		ASCII O
000,247	047		ASCII '
000,250	123		ASCII S
000,251	040		space
000,252	107		ASCII G
000,253	117		ASCII O
000,254	124		ASCII T

000,255	040	space
000,256	124	ASCII T
000,257	110	ASCII H
000,260	105	ASCII E
000,261	040	space
000,262	102	ASCII B
000,263	125	ASCII U
000,264	124	ASCII T
000,265	124	ASCII T
000,266	117	ASCII O
000,267	116	ASCII N
000,270	077	ASCII ?
000,271	012	line feed
000,272	015	carriage return
000,273	377	stop code
000,274	072	ASCII :
000,275	127	ASCII W
000,276	110	ASCII H
000,277	117	ASCII O
000,300	040	space
000,301	115	ASCII M
000,302	105	ASCII E
000,303	077	ASCII ?
000,304	012	line feed
000,305	015	carriage return
000,306	111	ASCII I
000,307	040	space
000,310	104	ASCII D
000,311	117	ASCII O
000,312	116	ASCII N
000,313	047	ASCII '
000,314	124	ASCII T
000,315	040	space
000,316	110	ASCII H
000,317	101	ASCII A
000,320	126	ASCII V
000,321	105	ASCII E
000,322	040	space
000,323	111	ASCII I
000,324	124	ASCII T
000,325	041	ASCII !
000,326	012	line feed
000,327	015	carriage return

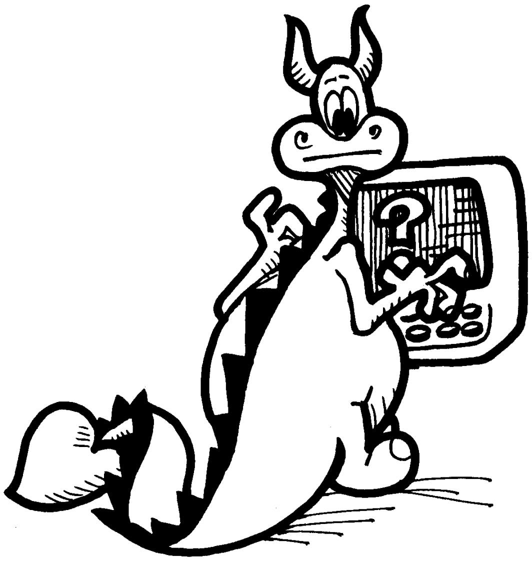
000,330	127	ASCII W
000,331	110	ASCII H
000,332	117	ASCII O
000,333	105	ASCII E
000,334	126	ASCII V
000,335	105	ASCII E
000,336	122	ASCII R
000,337	040	space
000,340	110	ASCII H
000,341	101	ASCII A
000,342	123	ASCII S
000,343	040	space
000,344	111	ASCII I
000,345	124	ASCII T
000,346	040	space
000,347	113	ASCII K
000,350	105	ASCII E
000,351	105	ASCII E
000,352	120	ASCII P
000,353	123	ASCII S
000,354	040	space
000,355	111	ASCII I
000,356	124	ASCII T
000,357	056	ASCII .
000,360	012	line feed
000,361	015	carriage return
000,362	377	stop code
000,363	072	ASCII :
000,364	111	ASCII I
000,365	040	space
000,366	104	ASCII D
000,367	117	ASCII O
000,370	116	ASCII N
000,371	047	ASCII '
000,372	124	ASCII T
000,373	040	space
000,374	110	ASCII H
000,375	101	ASCII A
000,376	126	ASCII V
000,377	105	ASCII E
001,000	040	space
001,001	111	ASCII I
001,002	124	ASCII T
001,003	012	line feed

001,004	015	carriage return
001,005	115	ASCII M
001,006	131	ASCII Y
001,007	040	space
001,010	116	ASCII N
001,011	105	ASCII E
001,012	111	ASCII I
001,013	107	ASCII G
001,014	110	ASCII H
001,015	102	ASCII B
001,016	117	ASCII O
001,017	122	ASCII R
001,020	040	space
001,021	104	ASCII D
001,022	117	ASCII O
001,023	105	ASCII E
001,024	123	ASCII S
001,025	056	ASCII .
001,026	012	line feed
001,027	015	carriage return
001,030	102	ASCII B
001,031	125	ASCII U
001,032	124	ASCII T
001,033	040	space
001,034	127	ASCII W
001,035	110	ASCII H
001,036	117	ASCII O
001,037	105	ASCII E
001,040	126	ASCII V
001,041	105	ASCII E
001,042	122	ASCII R
001,043	040	space
001,044	110	ASCII H
001,045	101	ASCII A
001,046	123	ASCII S
001,047	040	space
001,050	111	ASCII I
001,051	124	ASCII T
001,052	040	space
001,053	120	ASCII P
001,054	101	ASCII A
001,055	123	ASCII S
001,056	123	ASCII S
001,057	105	ASCII E
001,060	123	ASCII S

001,061	040	space
001,062	111	ASCII I
001,063	124	ASCII T
001,064	041	ASCII !
001,065	012	line feed
001,066	015	carriage return
001,067	377	stop code
001,070	072	ASCII :
001,071	122	ASCII R
001,072	111	ASCII I
001,073	107	ASCII G
001,074	110	ASCII H
001,075	124	ASCII T
001,076	040	space
001,077	131	ASCII Y
001,100	117	ASCII O
001,101	125	ASCII U
001,102	040	space
001,103	101	ASCII A
001,104	122	ASCII R
001,105	105	ASCII E
001,106	073	ASCII ;
001,107	040	space
001,110	114	ASCII L
001,111	125	ASCII U
001,112	103	ASCII C
001,113	113	ASCII K
001,114	131	ASCII Y
001,115	041	ASCII !
001,116	012	line feed
001,117	012	line feed
001,120	015	carriage return
001,121	377	stop code
001,122	072	ASCII :
001,123	116	ASCII N
001,124	117	ASCII O
001,125	040	space
001,126	117	ASCII O
001,127	116	ASCII N
001,130	105	ASCII E
001,131	040	space

001,132	110	ASCII H
001,133	105	ASCII E
001,134	122	ASCII R
001,135	105	ASCII E
001,136	040	space
001,137	127	ASCII W
001,140	111	ASCII I
001,141	124	ASCII T
001,142	110	ASCII H
001,143	040	space
001,144	124	ASCII T
001,145	110	ASCII H
001,146	101	ASCII A
001,147	124	ASCII T
001,150	040	space
001,151	043	ASCII #
001,152	056	ASCII .
001,153	040	space
001,154	124	ASCII T
001,155	122	ASCII R
001,156	131	ASCII Y
001,157	040	space
001,160	101	ASCII A
001,161	107	ASCII G
001,162	101	ASCII A
001,163	111	ASCII I
001,164	116	ASCII N
001,165	012	line feed
001,166	015	carriage return
001,167	377	stop code
001,170	077	random storage
001,171	116	random storage
001,172	000	stack
001,173	000	stack
001,174	000	stack
001,175	000	stack
001,176	000	stack
001,177	000	stack
001,200	000	stack

Load in all those op-codes and run the program. As these programs get longer, they become tedious and tiring for you to load into the computer, but it's all part of programming. Have patience and maybe rest ever so often. The most important op-codes are those from addresses 000,000 to 000,217; those must all be loaded perfectly. If a mistake is made in that area, when you go to run the program it'll more than likely disappear from memory—that's *real* frustration! If a mistake is made in the ASCII text portion of memory, it's not as important to the operation of the program; just make sure you get the "stop codes" in where they belong.



8
YOU'RE ON YOUR OWN

At this point you should try writing your own program. Decide what you want the computer to do, make a block diagram, then write the program, and run it. Very often when you run the program for the first time it won't work–don't get discouraged, it's a fact of life for even the most experienced programmers. Look at the program one op-code at a time, go slowly and keep track of what happens to each register on a piece of paper. If you can't find your mistake and a friend is available who's studying with you, have him follow it through. Something that helps me is just to put the program away and come back to it later. After working on the same program for many hours, a person's thinking gets sluggish; a fresh approach at a later time is sometimes the best solution.

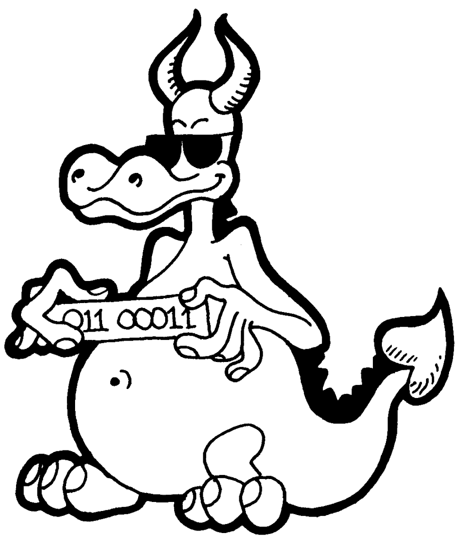
The process of fixing a program that doesn't work is called "debugging"; many times, debugging can be more work than writing the program.

Don't shoot for the moon on your first program. Keep the objective simple. If you can't think of anything on your own, here are two ideas.

Write a program to:

1. Add the numbers from 0 to 10 and store the result at address 000,100.
2. Roll a pair of imaginary dice and display the result of the roll on the terminal.

I have included versions of these two programs in Appendix I, but before you see the way I wrote them, write one yourself and make it work. Keep in mind that your objective here is to write a program that works–it doesn't necessarily have to be exactly like mine. If it works, it's right (at this point anyway).


011 00011

9
CONDITION BITS

The condition bits are contained (as a register) in the CPU. Three of the condition bits are the CARRY bit, the ZERO bit, and the SIGN bit. In the following discussion, "setting" a bit causes its value to be a 1, while "resetting" a bit causes its value to be a 0.

The condition bits are sometimes referred to as the "status bits," but I don't like that label because of possible confusion between the condition bits and the terminal STATUS word, which tells terminal input/output readiness.

It's not necessary to know this, but the Condition word has the following format:

```
SIGN BIT        AUXILIARY CARRY BIT        CARRY BIT
   |                     |                     |
   |      ZERO BIT       |      PARITY BIT     |
   |         |           |          |          |
   X         X     0     X     0    X     1    X
```

In this book I will not cover auxiliary, carry or parity.

<u>The ZERO bit</u>: If the result of any operation such as addition, subtraction, ANDing, ORing, or XORing is 0, then the ZERO bit is set to 1.

Example: If register C contains 00,000,001 and the DCR C (decrement register C) op-code is used, then register C will contain 00,000,000 and the ZERO bit will be set to 1.

The CARRY bit: The computer uses eight-bit words. If any operation such as addition, subtraction, rotating right, or rotating left causes a bit to move off the end of a word, then we say a bit was "carried" and the CARRY bit is set to 1.

Example: If the ACCUMULATOR contains 10,000,101 and the RRC (Rotate ACCUMULATOR right) instruction is used, then it will contain 11,000,010 and a CARRY occurred; the CARRY bit is set to 1 because the rightmost bit of the ACCUMULATOR was carried over to the leftmost bit.

A good way to use the CARRY bit might be to rewrite the INPUT STATUS subroutine we have been using. The purpose of the subroutine is to see if this bit

01,100,011

is a "1" or a "0." This was done by ANDing the STATUS word with 00,000,001. Can you see that another way of accomplishing it would be to "rotate" the STATUS word to the right one bit, and see if a CARRY occurs? These two subroutines produce the same result:

ADDRESS	OP-CODE		ADDRESS	OP-CODE	
000,000	333	IN	000,000	333	IN
000,001	000	STATUS	000,001	000	STATUS
000,002	346	ANI	000,002	017	RRC
000,003	001		000,003	332	JC
000,004	302	JNZ	000,004	000	
000,005	000		000,005	000	
000,006	000				

As long as the rightmost bit of the STATUS word is a "1," the computer will be stuck in this loop; when the right bit goes to "0," then operation will continue on to the rest of the program. Can you see one obvious advantage to the new subroutine? Can you apply this same principle of "rotation" of the STATUS word to create a new OUTPUT STATUS subroutine?

SIGNED NUMBERS

So far in this text we have been dealing with eight-bit binary numbers. The range is 00000000 (decimal 0), to 11111111 (decimal 255), and for the rest of the book we will continue in this manner. However, there is another way to assign a decimal number to our eight-bit binary number. The lower seven bits of the binary number can be used to represent the magnitude of the number, and the highest bit can represent whether the number is positive or negative.

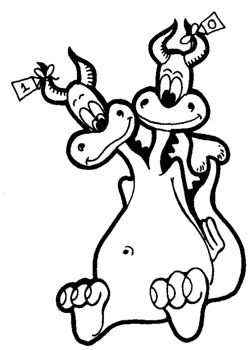
Therefore:

+127 decimal	=	01111111
0 decimal	=	00000000
−128 decimal	=	10000000

The "1" in the leftmost bit means the number is negative.

The SIGN bit: If an operation such as addition, subtraction, ANDing, and so on results in a negative number, the SIGN bit is set to 1. If an operation results in a number that is positive, the SIGN bit is reset to 0.

Example: If the ACCUMULATOR contains 00,000,011 (which is a binary 3), and the SBI 004 (subtract 00,000,100) instruction is used, the result is 11,111,111 (which is a binary -1), so the SIGN bit is set to 0.




MEMORY

10
THE OP-CODES: DEFINED

You don't know all the 8080 op-codes, but you know the most useful ones and, more importantly, you know the basic structure of the 8080 system of machine language.

Most of the op-codes are listed in this chapter, along with a short definition of each.

I feel that if you know, and can use, these common instructions for the 8080 microprocessor, you will have no trouble mastering the few that are left.

The key to learning machine language programming as a hobbyist is to experiment.

8080 OP-CODES

IN (Input)

333 terminal	An eight-bit byte is moved from the terminal into the ACCUMULATOR. Most terminals have two numbers associated with them, one for STATUS and one for DATA.

OUT (Output)

323 terminal	An eight-bit byte is moved from the ACCUMULATOR out to the terminal.

NOP (No operation)

000	Nothing happens; the machine proceeds to the next instruction.

JMP (Jump)

303 nnn mmm	The machine jumps to address mmm,nnn. (From this point on I will refer to a general address as mmm,nnn. The general address mmm,nnn can represent any octal address such as 000,100 and so on.)

JC (Jump if CARRY)

332 nnn mmm	This is a conditional instruction. If the CARRY bit is 1, then a carry has occurred and the machine will jump to address mmm,nnn. If the CARRY bit is 0, then no carry has occurred and operation continues with the next op-code.

JNC (Jump if no CARRY)

322 nnn mmm	This is also conditional. If the CARRY bit is 0, then no carry has occurred and the machine will jump to address mmm,nnn. If the CARRY bit is 1, then operation continues with the next op-code.

JZ (Jump if ZERO)

312 nnn mmm	Conditional. If the ZERO bit is 1, then the result of a previous test was 0 and the machine will jump to address mmm,nnn. If the ZERO bit is 0, then operation continues with the next op-code.

JNZ (Jump if not ZERO)

302 nnn mmm	If the ZERO bit is 0 (the result of a previous test was *not* 0), then the machine will jump to address mmm,nnn. Otherwise, operation continues with the next op-code.

JM (Jump if MINUS)

372 nnn mmm	Conditional instruction. If the SIGN bit is 1 (which means a result was negative), then the machine jumps to address mmm,nnn. If the SIGN bit is 0 (positive result), then we go to the next sequential op-code.

JP (Jump if POSITIVE)

362 nnn mmm	Conditional. If the SIGN bit is 0 (positive result), then the machine jumps to address mmm,nnn. If the SIGN bit is 1 (negative result), then operation continues with the next instruction.

CALL INSTRUCTIONS

A CALL instruction is like a JUMP except that when a CALL op-code is used, you usually intend to return back to the main program by use of the RETURN op-code. Before a CALL instruction is executed, the next sequential op-code address is saved on the STACK so that later, when the RETURN instruction is met, the CPU will know what address to return to.

CALL

315 nnn mmm	The machine unconditionally moves to address mmm,nnn and executes the subroutine at that address.

CZ (Call if ZERO)

314 nnn mmm	This is a conditional op-code. If the ZERO bit is 1, a previous test resulted in 0 and the machine will move to address mmm,nnn. If the ZERO bit is 0 (a non-zero result), then no subroutine is CALLed and the next op-code in line is executed.

CNZ (Call if not ZERO)

304	Conditional. If the ZERO bit is 0 (a previous test

nnn mmm	resulted in a non-zero number), then address mmm,nnn is CALLed. Otherwise, the ZERO bit is 1 and operation continues with the next instruction.

CC (Call if CARRY)

334 nnn mmm	This is conditional. If the CARRY bit is 1, then a CARRY has occurred and address mmm,nnn is CALLed. Otherwise, the CARRY bit is 0, which means no CARRY and operation continues with the next sequential op-code.

CNC (Call if no CARRY)

324 nnn mmm	Conditional. If the CARRY bit is 0, then a previous test resulted in no CARRY, so address mmm,nnn is CALLed. If the CARRY bit is 1, the machine continues on to the next op-code.

CP (Call if POSITIVE)

364 nnn mmm	Conditional. If a previous test resulted in a positive number, the SIGN bit will be 0 and the CALL to address mmm,nnn is made. If the SIGN bit is 1, then operation continues with the next op-code.

CM (Call if MINUS)

374 nnn mmm	Conditional. If the status of the SIGN bit is 1 (a negative result), the machine CALLs address mmm,nnn. If the SIGN bit is 0, the result was positive and the program continues sequential operation.

RETURN INSTRUCTIONS

When a CALL instruction is executed, the address of the next sequential op-code is automatically pushed onto the STACK. The subroutine CALLed will usually have a RETURN instruction in it. This instruction pops the address saved off the STACK and operation resumes at that address. RETURNs may or may not be conditional.

RET (Return)

311 The subroutine has been completed and the machine unconditionally returns back, to the next address following the initial CALL op-code.

RC (Return if CARRY)

330 Conditional. If the CARRY bit is 1, a CARRY has occurred and the machine will automatically return to the next sequential address following the original CALL instruction.

RNC (Return if no CARRY)

320 Also conditional. If the CARRY bit is 0, then a CARRY has not occurred and the CPU will return back to the main program at the next address after the initial CALL op-code.

RZ (Return if ZERO)

310 Conditional. If the ZERO bit is 1, a previous test resulted in zero, and the machine returns back to the main program at the next address after the original CALL instruction. If the ZERO bit is 0, then a result was non-zero and the subroutine continues.

RNZ (Return if not ZERO)

300 Conditional. If the ZERO bit is 0 (a non-zero answer to a previous test), then we return back to the main program at the next address after the initial CALL op-code. Otherwise, the subroutine continues.

RP (Return if POSITIVE)

360 Conditional. If the status of the SIGN bit is 0 (a positive result), the machine automatically returns to the next sequential address following the initial CALL instruction. If the SIGN bit is 1 (a negative result), then the subroutine continues with the next op-code.

RM (Return if MINUS)

370 Conditional. If the SIGN bit is 1 (negative result), we return back to the main program. If the SIGN bit is 0 (positive result), then the subroutine continues.

More than one conditional RETURN instruction can be put in a subroutine.

THE ACCUMULATOR INSTRUCTIONS

RLC (Rotate ACCUMULATOR left)

007 The ACCUMULATOR byte is moved one bit to the left. The end bit wraps around. For example, if the ACCUMULATOR is 00,101,111, then after RLC it will be 01,011,110.

RRC (Rotate ACCUMULATOR right)

017 The ACCUMULATOR byte is moved one bit to the right. Again, the end bit wraps around. If the ACCUMULATOR is 00,101,111, then after RRC it will be 10,010,111.

CMA (Complement the ACCUMULATOR)

057 Each bit in the ACCUMULATOR is complemented, which means 1's become 0's and 0's become 1's. If it was 00,101,111, then it will be 11,010,000.

ADI (Add immediate to the ACCUMULATOR)

306
ddd The DATA byte ddd is added to the ACCUMULATOR. The sum is then put back into the ACCUMULATOR. Since we are adding two numbers, the SIGN, ZERO and CARRY bits could be affected.

SUI (Subtract immediate from the ACCUMULATOR)

326
ddd The DATA byte ddd is subtracted from the ACCUMULATOR. The sum is put back into the ACCUMULATOR. Since we are subtracting two numbers, the SIGN, ZERO, and CARRY bits could be affected.

ANI (AND immediate with the ACCUMULATOR)

346
ddd

The DATA byte ddd is ANDed with the ACCUMULATOR, and the result is put into the ACCUMULATOR. The CARRY bit is reset to 0, and the SIGN and ZERO bits could be affected, depending on the result.
Example: If the ACCUMULATOR is 00,101,111 and ddd is 11,010,000, then the result after ANI will be 00,000,000, which will be put into the ACCUMULATOR. Since the answer is zero, the ZERO bit will be set to 1.

ORI (OR immediate with the ACCUMULATOR)

366
ddd

The DATA byte ddd is ORed with the ACCUMULATOR, and the result is put into the ACCUMULATOR. The CARRY bit is reset to 0, and the SIGN and ZERO bits might be affected.

XRI (Exclusive-OR immediate with the ACCUMULATOR)

356
ddd

The DATA byte ddd is Exclusive-ORed with the ACCUMULATOR and the result is put into the ACCUMULATOR. The CARRY bit is reset to 0. The SIGN and ZERO bits might be affected.
Example: If the ACCUMULATOR is 00,101,111 and ddd is 00,000,101, then after XRI the ACCUMULATOR will contain 00,101,010. The ZERO bit will be reset to 0 because the result was not zero, and the SIGN bit will be reset to 0 because the result was positive.

CPI (Compare immediate with the ACCUMULATOR)

376
ddd

The DATA byte ddd is compared with the ACCUMULATOR, by subtracting the DATA byte from the ACCUMULATOR. The result is *not* put into the ACCUMULATOR—it remains unchanged. The SIGN, ZERO and CARRY bits could be affected.

STA (Store the ACCUMULATOR)

062 nnn mmm	The contents of the ACCUMULATOR are stored at address mmm,nnn.

LDA (Load the ACCUMULATOR)

072 nnn mmm	The DATA byte at address mmm,nnn is loaded into the ACCUMULATOR.

STC (Set CARRY)

067	The CARRY bit is set to 1. No other condition bits are affected.

CMC (Complement CARRY)

077	The CARRY bit is complemented, which means if it's initially 1 it is reset to 0, and if it's initially 0 it is set to 1. ZERO and SIGN bits are not affected.

In the following section I have given the op-codes for the MOVE, INCREMENT, DECREMENT, and REGISTER instructions. Each of these instructions has several variations, depending on which register(s) is(are) used. For example: The INCREMENT instruction can add 1 to register B, C, D, E, H, L, or the ACCUMULATOR, depending on how it is written. The INCREMENT op-code is 0_4, where the blank is filled with the number of the register to be incremented. The code is:

Register B is 0
Register C is 1
Register D is 2
Register E is 3
Register H is 4
Register L is 5
Register M is 6 (where M is the DATA at H/L)
ACCUMULATOR is 7

So, if we wanted to INCREMENT register E, we would use the op-code 034. If we wanted to INCREMENT the DATA at address H/L, we would use the op-code 064. If we wanted to INCREMENT the ACCUMULATOR, we would use 074.

This method of "fill in the register" will be used on all the codes for INCREMENT, DECREMENT, MOVE, and REGISTER instructions that follow.

INR (Increment register or memory)

0_4	The specified register (or memory address H/L) is incremented by 1. The ZERO or SIGN bits can be affected.

DCR (Decrement register or memory)

0_5	The specified register or memory address H/L is decremented by 1. Again the ZERO or SIGN bits can be affected.

MVI (Move immediate DATA into register)

0_6 ddd	The DATA byte ddd is moved into the specified register (or memory address H/L). No condition bits are affected.

MOV (Move DATA from one register to another)

0__	There are two blanks in this op-code. DATA is moved from the register in the right blank to the register in the center blank. For example: 071 means to move the contents of register C into the ACCUMULATOR. 067 means move the DATA in the ACCUMULATOR to the address specified by H/L.

<u>REMEMBER</u>

Register B is 0
Register C is 1
Register D is 2
Register E is 3
Register H is 4
Register L is 5
Register M is 6 (address H/L)
ACCUMULATOR is 7

REGISTER INSTRUCTIONS

ADDR (Add register to the ACCUMULATOR)

20_ The specified register is added to the ACCUMULATOR. CARRY, SIGN, and ZERO bits can be affected. As an example: 204 means to add the contents of register H to the ACCUMULATOR and put the sum back in the ACCUMULATOR.

SUB (Subtract register from the ACCUMULATOR)

22_ The specified register is subtracted from the ACCUMULATOR and the result is put back in the ACCUMULATOR. CARRY, SIGN and ZERO bits can be affected. Example: 226 means to subtract the DATA byte at address H/L from the ACCUMULATOR.

ANA (AND register with the ACCUMULATOR)

24_ The specified register is ANDed with the ACCUMULATOR and the result is put back in the ACCUMULATOR. The CARRY bit is reset to 0. The ZERO and SIGN bits can be affected.

ORA (OR register with the ACCUMULATOR)

26_ The specified register is ORed with the ACCUMULATOR and the result is put back in the ACCUMULATOR. The CARRY bit is reset to 0. The ZERO and SIGN bits can be affected.

XRA (Exclusive-OR register with the ACCUMULATOR)

25_ The specified register is Exclusive-ORed with the ACCUMULATOR and the result put in the ACCUMULATOR. CARRY bit is reset to 0. The ZERO and SIGN bits can be affected.

CMP (Compare register to the ACCUMULATOR)

27_	The specified register is compared with the ACCUMULATOR, by subtracting the register from the ACCUMULATOR. The contents of the ACCUMULATOR and the contents of the register are *not* changed. The CARRY, ZERO, and SIGN bits can be affected.

The CMP instruction is useful in determining if a particular register is the same as the ACCUMULATOR. If the two bytes are equal, the ZERO bit will be set to 1. If the specified register is larger than the ACCUMULATOR, the CARRY bit will be set to 1. If the specified register is smaller than the ACCUMULATOR, the CARRY bit will be reset to 0. Do you see why?

PUSH, POP, and LOAD

These three op-codes have several variations, depending on which register *pairs* are used. The codes for the register pairs are:

Register pair B,C	is	0
Register pair D,E	is	2
Register pair H/L	is	4
STACK pointer	is	6 (top of STACK)
ACCUMULATOR and condition bits are also		6

LXI (Load register pair immediate)

0_1 nnn mmm	Two bytes of immediate DATA are loaded into the specified register pair. The DATA nnn is loaded into the second register of the pair, and mmm is loaded into the first register of the pair. DATA mmm and nnn does not need to represent an address—although many times it does.

For example: If the op-code 041 is used, then DATA nnn will be put in register L and DATA mmm will be put into register H. The condition bits (CARRY, and so on) are not affected.

Example: If the op-code 061 is used, then the top of the STACK (the STACK pointer) will be address mmm,nnn.

The LXI instruction is very useful for loading DATA or addresses into register pairs.

Remember that the STACK is a portion of memory where DATA or addresses are temporarily stored. The PUSH instruction moves the DATA contained in a register pair onto the STACK. Two bytes of DATA are moved onto the STACK at a time. The POP instruction moves the top two DATA bytes off the STACK and into a specified register pair. Two bytes of DATA are moved off the STACK at a time.

PUSH(Push DATA onto the STACK)

3_5 The contents of the specified register pair are stored in two bytes of memory, at an address specified by the STACK pointer. The contents of the first register are PUSHed into an address *one less than the STACK pointer*, the contents of the second register are PUSHed into an address *two less than the STACK pointer*.

Example: If the op-code 305 is used, register B will be PUSHed into an address one less than the STACK pointer, and register C will be PUSHed into an address two less than the STACK pointer. Condition bits (CARRY, and so on) are not affected.

POP (Pop DATA off the STACK)

3_1 The top two bytes of DATA on the STACK (defined by STACK pointer plus one, and STACK pointer plus two) are moved into the two registers specified.

Example: If the op-code 301 is used, the DATA at the STACK pointer address plus one is moved into register B, and the DATA at the STACK pointer address plus two is moved into register C. The condition bits are not affected unless op-code 361 is used.

INCREMENT/DECREMENT REGISTER PAIRS

As you've no doubt noticed, registers are often paired in 8080 programming. Thus a register pair can be used to represent a single sixteen-bit number. The INX (Increment) and DCX (Decrement) instructions consider the register pairs B,C; D,E; and H/L as single sixteen-bit binary numbers. Therefore, if register B contains 00,000,000 and C contains 11,111,111, then after the INX B,C op-code, register B will contain 00,000,001 and register C will contain 00,000,000. In other words, 00,000,000 11,111,111 was incremented to 00,000,001 00,000,000. Condition bits are *not* affected. The op-codes are:

003	Increment register pair B,C.
023	Increment register pair D,E.
043	Increment register pair H/L.
063	Increment the STACK pointer (which is a sixteen-bit number)
013	Decrement register pair B,C.
033	Decrement register pair D,E.
053	Decrement register pair H/L.
073	Decrement the STACK pointer.

THE H/L OP-CODES

XCHG (Exchange registers)

353 The sixteen-bit number formed by the H and L registers is exchanged with the sixteen-bit number formed by the D and E registers. H and D are exchanged, and L and E are exchanged. Condition bits are not affected.

XTHL (Exchange STACK)

343 The DATA byte in register L is exchanged with the DATA byte at the STACK pointer address, and the

byte in register H is exchanged with the byte at the STACK pointer address plus one. Condition bits are not affected.

SPHL (Load STACK pointer from H/L)

371 The sixteen-bit contents of the registers H/L replace the contents of the STACK pointer—this moves the STACK. The H and L registers are not changed. Condition bits are not affected.

PCHL (Load program counter from H/L)

351 This is in effect a jump instruction. The machine will jump to the address specified by the H and L registers and continue executing the program from there. H and L do not change. The condition bits are unaffected.

SHLD (Store H and L direct)

042
nnn
mmm The DATA byte in register L is stored at address mmm,nnn; and the DATA byte in register H is stored at address mmm,nnn plus one. The condition bits are not affected.

LHLD (Load H and L direct)

052
nnn
mmm Register L is loaded with the DATA byte at address mmm,nnn; and register H is loaded with the data byte at address mmm,nnn plus one. Condition bits are not affected.

APPENDIX I

THE SUM OF NUMBERS 0 TO 10

This is a program to add the numbers from 0 to 10 and store at 000,100.

ADDRESS	OP-CODE		EXPLANATION
000,000	006	MVI B	Move into register B
000,001	012		the binary number 10.
000,002	076	MVI A	Move into the ACCUMULATOR
000,003	000		the binary number 0.
000,004	200	ADDR B	Add register B to the ACCUMULATOR (the result goes into the ACCUMULATOR).
000,005	005	DCR B	Decrement register B.
000,006	302	JNZ	Jump if not zero
000,007	004		back to address 000,004.
000,010	000		
000,011	062	STA	Store ACCUMULATOR (sum
000,012	100		of 0 through 10) at address
000,013	000		000,100.
000,014	303	JMP	Loop here when done.
000,015	014		
000,016	000		

This program puts a binary 10 in register B and a 0 in the ACCUMULATOR, then adds them and the result goes back in the ACCUMULATOR. The ACCUMULATOR contains 0 + 10 = 10. Then register B is decremented to 9 and added to ACCUMULATOR. The ACCUMULATOR contains 10 + 9 = 19. Then register B is decremented to 8 and added to the ACCUMULATOR, and so on. After register B becomes 0, the ACCUMULATOR is stored at 000,100. It will be the binary number 00,110,111 (55).

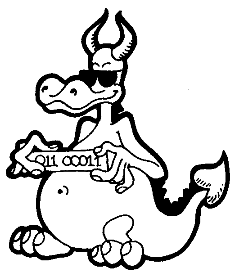
THE ROLL OF TWO DICE

In this program two dice are rolled and the results are displayed on a terminal when a carriage return is typed.

ADDRESS		OP-CODE	EXPLANATION
000,000	061	LXI SP	Initialize the STACK.
000,001	130		
000,002	000		
000,003	333	IN	Input a DATA byte from terminal. If it's a carriage return, go on to address 000,021; if not, go back and input again.
000,004	000	STATUS	
000,005	346	ANI	
000,006	001		
000,007	302	JNZ	
000,010	003		
000,011	000		
000,012	333	IN	
000,013	001	DATA	
000,014	376	CPI	
000,015	015	carriage return	
000,016	302	JNZ	
000,017	003		
000,020	000		
000,021	076	MVI A	Print a line feed.
000,022	012	line feed	
000,023	315	CALL	
000,024	104		
000,025	000		
000,026	076	MVI A	Print a carriage return.
000,027	015	carriage return	
000,030	315	CALL	
000,031	104		
000,032	000		
000,033	315	CALL	Get a random number from 0 to 6 and print it on the terminal.
000,034	057		
000,035	000		
000,036	315	CALL	
000,037	104		
000,040	000		

000,041	076	MVI A	Print a comma.
000,042	054	comma	
000,043	315	CALL	
000,044	104		
000,045	000		
000,046	315	CALL	Get a random number from 0 to 6 and print it on the terminal.
000,047	057		
000,050	000		
000,051	315	CALL	
000,052	104		
000,053	000		
000,054	303	JMP	Jump back to the beginning.
000,055	000		
000,056	000		
000,057	041	LXI H/L	Get a random number.
000,060	120		
000,061	000		
000,062	176	MOV A,M	
000,063	017	RRC	
000,064	206	ADDR M	
000,065	017	RRC	
000,066	167	MOV M,A	
000,067	043	INX H/L	
000,070	256	XRA M	
000,071	167	MOV M,A	
000,072	346	ANI	Make sure it's between 0 and 6.
000,073	007		
000,074	376	CPI	
000,075	007		
000,076	362	JP	
000,077	057		
000,100	000		
000,101	366	ORI	Convert the binary number to ASCII. Return.
000,102	060		
000,103	311	RET	
000,104	365	PUSH A	
000,105	333	IN	
000,106	000		
000,107	346	ANI	

000,110	200		Output the ACCUMULATOR to the terminal.
000,111	302	JNZ	
000,112	105		
000,113	000		
000,114	361	POP A	
000,115	323	OUT	
000,116	001	DATA	
000,117	311	RET	
000,120	253		These two addresses are storage for the random number generator.
000,121	144		



APPENDIX II

A BETTER RANDOM NUMBER GENERATOR

The random number generator we've been using is okay for simple programs, but here's a much better generator:

ADDRESS	OP-CODE		EXPLANATION
000,000	041	LXI H/L	This generator uses four addresses for temporary storage; they are 000,045; 000,046; 000,047; and 000,050.
000,001	050		
000,002	000		
000,003	006	MVI B	
000,004	010	binary 8	
000,005	176	MOV A,M	
000,006	007	RLC	
000,007	007	RLC	
000,010	007	RLC	
000,011	256	XRA M	
000,012	027	RAL	
000,013	027	RAL	
000,014	055	DCR L	
000,015	055	DCR L	
000,016	055	DCR L	
000,017	176	MOV A,M	
000,020	027	RAL	
000,021	167	MOV M,A	
000,022	054	INR L	
000,023	176	MOV A,M	
000,024	027	RAL	
000,025	167	MOV M,A	
000,026	054	INR L	
000,027	176	MOV A,M	
000,030	027	RAL	
000,031	167	MOV M,A	
000,032	054	INR L	
000,033	176	MOV A,M	
000,034	027	RAL	
000,035	167	MOV M,A	
000,036	005	DCR B	
000,037	302	JNZ	
000,040	006		
000,041	000		
000,042	XXX		Jump back to program or return.
000,043	XXX		
000,044	XXX		

APPENDIX III

8080 REFERENCE TABLE

returns	
RET	311
RNZ	300
RZ	310
RNC	320
RC	330
RP	360
RM	370

jumps	
JMP	303
nnn	
mmm	
JNZ	302
nnn	
mmm	
JZ	312
nnn	
mmm	
JNC	322
nnn	
mmm	
JC	332
nnn	
mmm	
JP	362
nnn	
mmm	
JM	372
nnn	
mmm	

codes		
B	=	0
C	=	1
D	=	2
E	=	3
H	=	4
L	=	5
M	=	6
A	=	7

in	
IN	333
ddd	

out	
OUT	323
ddd	

calls	
CALL	315
nnn	
mmm	
CNZ	304
nnn	
mmm	
CZ	314
nnn	
mmm	
CNC	324
nnn	
mmm	
CC	334
nnn	
mmm	
CP	364
nnn	
mmm	
CM	374
nnn	
mmm	

stack	
PUSH BC	305
PUSH DE	325
PUSH HL	345
PUSH A	365
POP BC	301
POP DE	321
POP HL	341
POP A	361

registers	
ADDRr	20_
ADC r	21_
SUB r	22_
ANA r	24_
XRA r	25_
ORA r	26_
CMP r	27_

increment	
INR r	0_4

decrement	
DCR r	0_5

moves	
MOV r r	1__
MVI r	0_6
ddd	

no-op	
NOP	000

H/L	
XCHG	353
XTHL	343
SPHL	371
PCHL	351

shifts	
RLC	007
RRC	017

accumulator	
ADI	306
ddd	
SUI	326
ddd	
ANI	346
ddd	
XRI	356
ddd	
ORI	366
ddd	
CPI	376
ddd	

direct	
SHLD	042
nnn	
mmm	
LHLD	052
nnn	
mmm	
STA	062
nnn	
mmm	
LDA	072
nnn	
mmm	

pairs	
STAX BC	002
STAX DE	022
LDAX BC	012
LDAX DE	032
LXI BC	001
nnn	
mmm	
LXI DE	021
nnn	
mmm	
LXI HL	041
nnn	
mmm	
LXI SP	061
nnn	
mmm	

incr pair	
INX BC	003
INX DE	023
INX HL	043
INX SP	063

decr pair	
DCX BC	013
DCX DE	033
DCX HL	053
DCX SP	073

ASCII CODES

BINARY	CHARACTER
00000000	NUL
00000111	BEL
00001010	line feed
00001101	carriage return
00100000	space
00100001	!
00100010	"
00100011	#
00100100	$
00100101	%
00100110	&
00100111	'
00101000	(
00101001	)
00101010	*
00101011	+
00101100	,
00101101	–
00101110	.
00101111	/
00110000	0
00110001	1
00110010	2
00110011	3
00110100	4
00110101	5
00110110	6
00110111	7
00111000	8
00111001	9
00111010	:
00111011	;
00111100	<
00111101	=
00111110	>
00111111	?
01000000	@

01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01000110	F
01000111	G
01001000	H
01001001	I
01001010	J
01001011	K
01001100	L
01001101	M
01001110	N
01001111	O
01010000	P
01010001	Q
01010010	R
01010011	S
01010100	T
01010101	U
01010110	V
01010111	W
01011000	X
01011001	Y
01011010	Z
01011011	[
01011100	\
01011101	]
01011110	∧
01011111	_

There are more ASCII codes, but these are the most common.

ANSWERS TO QUESTIONS

In this section I have tried to answer all questions asked in the book, in case you could not find the answer by reading. They are listed by page number.

page 2

1.

decimal	binary
1	001
0	000
2	010
6	110
5	101
3	011
7	111
4	100

2. The binary system contains only two integers: 0 and 1. With those two symbols any real number can be made.

3.

$$\begin{array}{rcl} \text{decimal } 8 & = & \text{binary } 1000 \\ \underline{\text{decimal } 1} & = & \underline{\text{binary } 0001} \\ \text{decimal } 9 & = & \text{binary } 1001 \end{array}$$

4.

$$\begin{array}{r} 000 \\ +001 \\ \hline 001 \end{array} \quad \begin{array}{r} 000 \\ +010 \\ \hline 010 \end{array} \quad \begin{array}{r} 100 \\ +001 \\ \hline 101 \end{array} \quad \begin{array}{r} 001 \\ +101 \\ \hline 110 \end{array} \quad \begin{array}{r} 101 \\ +010 \\ \hline 111 \end{array} \quad \begin{array}{r} 001 \\ +111 \\ \hline 1000 \end{array}$$

binary 001 = decimal 1
binary 010 = decimal 2
binary 101 = decimal 5
binary 110 = decimal 6
binary 111 = decimal 7
binary 1000 = decimal 8

page 3

for the 8080 microprocessor:

1. eight bits = one byte

2. sixteen bits = an address

3.

binary		octal
00 000 100	=	004
00 000 011	=	003
00 001 000	=	010
01 000 101	=	105
10 001 001	=	211
01 111 000	=	170
00 000 001	=	001
11 000 011	=	303
11 111 010	=	372
10 001 001	=	211
00 110 101	=	065
11 001 001	=	311

page 10

```
     11101111          00010001          11110000
AND  10010001     AND  00010000     AND  00111111
     10000001          00010000          00110000

     00011111          11111111          00000000
 OR  11001010      OR  10000010      OR  00011000
     11011111          11111111          00011000
```

1. A byte contains eight bits.

2. An address contains sixteen bits.

3. Binary 01 000 011 = octal 103.

4. Octal 303 = binary 11 000 011.

5. All eight working registers are contained in the 8080 central processor.

6. The STATUS word tells whether or not the terminal is ready to send or receive DATA.

7. The *first* and *last* bits of the STATUS word are important for the computer system used in this book. Which two bits are used for testing STATUS in your system?

8. The rightmost bit of the STATUS word tells if the terminal is ready to *send* DATA to the computer. The leftmost bit of the STATUS word tells if the terminal is ready to *receive* DATA from the computer. Are these the same two bits used for your system?

9. For my system, the octal numbers 000 and 001 are used to get the terminal STATUS and for terminal DATA.

10. Computers, in general, perform their work extremely fast, and they are constantly waiting on the terminal. This is the reason for checking terminal STATUS before inputting DATA from it or outputting DATA to it.

11. The ASCII code for the letter "A" is 01 000 001 in binary, or 101 in octal.

12. As you progress through the alphabet from A to Z, the ASCII equivalents get larger:

    A = 101
    M = 115
    Z = 132

    So if you wrote a program to put the ASCII codes in numerical order, then you would be putting the words in alphabetical order, too.

    Notice also that the ASCII codes of our decimal digits 0 through 9 are in numeric order:

    0 = 060
    5 = 065
    9 = 071

page 16

Change the op-code at 000,010 to 102 (an ASCII "B").

page 31

This program will generate an ASCII number from 060 to 063:

ADDRESS	OP-CODE	
000,000	041	LXI H/L
000,001	024	
000,002	000	
000,003	176	MOV A,M

000,004	017	RRC
000,005	206	ADD M
000,006	017	RRC
000,007	167	MOV M,A
000,010	043	INX H/L
000,011	256	XRA M
000,012	167	MOV M,A
000,013	346	ANI
000,014	003	
000,015	366	ORI
000,016	060	
000,017	323	OUT
000,020	001	
000,021	303	JMP
000,022	021	
000,023	000	

Note that the only real difference in this program is at address 000,014, where the ACCUMULATOR is ANDed with 00 000 011. This will always result in a number from 00 000 000 to 00 000 011. At address 000,015 the ACCUMULATOR is ORed with 00 110 000, which changes it to an ASCII code.

This program will generate a *binary* number from 00 000 000 to 00 000 011.

ADDRESS	OP-CODE	
000,000	041	LXI H/L
000,001	020	
000,002	000	
000,003	176	MOV A,M
000,004	017	RRC
000,005	206	ADD M
000,006	017	RRC
000,007	167	MOV M,A
000,010	043	INX H/L
000,011	256	XRA M
000,012	167	MOV M,A

000,013	346	ANI
000,014	003	
000,015	303	JMP
000,016	015	
000,017	000	

Of course, this program will not print the binary numbers on the terminal because only ASCII DATA can be sent out—binary DATA will not print on the terminal. When address 000,015 is reached, the ACCUMULATOR will contain a binary number from 00 000 000 to 00 000 011.

This program will continue to print out random numbers until you stop it:

ADDRESS	OP-CODE	
000,000	333	IN
000,001	000	STATUS
000,002	346	ANI
000,003	200	
000,004	302	JNZ
000,005	000	
000,006	000	
000,007	041	LXI H/L
000,010	033	
000,011	000	
000,012	176	MOV A,M
000,013	017	RRC
000,014	206	ADD M
000,015	017	RRC
000,016	167	MOV M,A
000,017	043	INX H/L
000,020	256	XRA M
000,021	167	MOV M,A
000,022	346	ANI
000,023	007	
000,024	366	ORI
000,025	060	

000,026	323	OUT
000,027	001	
000,030	303	JMP
000,031	000	
000,032	000	

Notice an output STATUS check was added.

page 72

An advantage to testing for CARRY is that one less memory location is used. The CARRY method uses only six address locations—the ANDing method uses seven. Memory space should always be conserved if possible.

Here is a STATUS subroutine using the "rotate and check CARRY" method to determine terminal output STATUS:

ADDRESS	OP-CODE	
000,000	333	IN
000,001	000	STATUS
000,002	007	RLC
000,003	332	JC
000,004	000	
000,005	000	

Notice that on this one, the ACCUMULATOR gets rotated left. On the output STATUS subroutine, we want to check the *leftmost* bit of the STATUS word.

INDEX