Laboratory Exercises

KENBAK-1 Computer

KENBAK CORPORATION

P.O. BOX 49324

LOS ANGELES, CA. 90049

Preface

These Laboratory Exercises, in a workbook format, are intended for independent work. Ideas and concepts are introduced in small increments and are interspersed with actual experience on the computer. Emphasis is placed on the principles of self-discovery, immediate feedback, and rewards.

Other books or material might be used as companion volumes for a more formal and theoretical presentation. However, the Exercises are independent and may be used as the only reference material.

Questions and comments are welcome.

Kenbak Corporation
P. O. Box 49324
Los Angeles, California 90049
U. S. A.

Exercise 1

Every computer has a memory in which numbers are stored. Some of the numbers are instructions which tell the computer what to do. Other numbers are data which the computer operates on to obtain the answers to problems. Before long we will expect you to be telling the computer what to do.

The memory in our computer is divided into 256 memory locations. At this time we can understand these best as 256 boxes or drawers, see Figure 1.1. Each box is one memory location and contains one number. A number may be changed in any location by using the pushbuttons on the front panel of the computer. When a number is stored or placed in a memory location, the number which was there before is destroyed or lost. Or a number in any location may be examined and displayed in the lights on the front panel of the computer. Examining or reading a number in a memory location does not change it or remove it.

Every memory location has an address which is the name of the location. This address is a number itself. If we wish to read or store a number in a memory location, we must first say what the address is.

Because the word "number" can mean two different things to us, we are likely to get confused. You may not know whether we mean an address of a memory location or whether we mean the contents of the memory location. Let's agree that when we say:

Number: We mean a number in general. It could be an address, or the contents of a memory location, or something else.

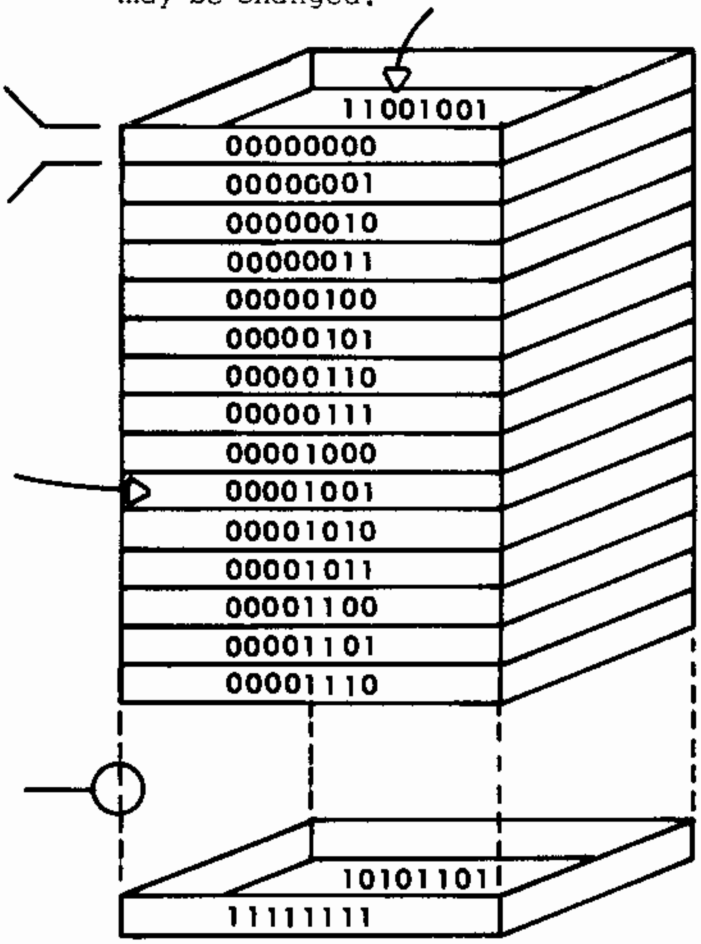
Address: We mean a number which is used as the name of a memory location.

Data: We mean a number which is stored or is going to be stored in a memory location.

All addresses are numbers. All data are numbers. Not all numbers are addresses or data.

Each box or memory location contains one number. We can see the number in this box because it happens to be on top. The other memory locations contain numbers too. The numbers may be changed.

This box represents one memory location. There are 256 memory locations in the computer.



The number on the outside of the box is the address of the memory location. It never changes.

The dashed lines indicate boxes which we did not draw.

Figure 1.1 The Memory In The Computer May Be Thought Of As A Group Of Boxes

Our addresses and data are made of eight digits where each digit is a 0 or a 1. This kind of number is called "binary" because there are only two choices in each position, 0 or 1. In our more familiar decimal numbers there are ten possibilities in each position. For the computer, binary numbers are best. For us, binary numbers seem strange. We will see that they are very easy to learn though.

We will let a 0 in a binary number mean that no button is to be pushed. We will let a 1 mean that a button is to be pushed. We will read a light that is off as a 0. A light that is on will be a 1.

0 = Button Not Pushed = □ or Light Off = o
1 = Button Pushed = ■ or Light On = ●

That's easy enough, isn't it?

Let's use what we have learned with the computer. Figure 1.2 shows the location of the lights and switches on the front panel. We won't tell you about all of the switches and lights now.

If power is off when you begin, turn the Power switch to the On position. Push the Start button and then push the Stop button. If power is already on, push Stop. Then you are ready.

The small toggle switch between the Address and the Memory lights should be placed in the UNL position. When this switch is in the LOCK position, data cannot be entered from the front panel. In the UNL (for Unlock) position, entry from the control panel is possible.

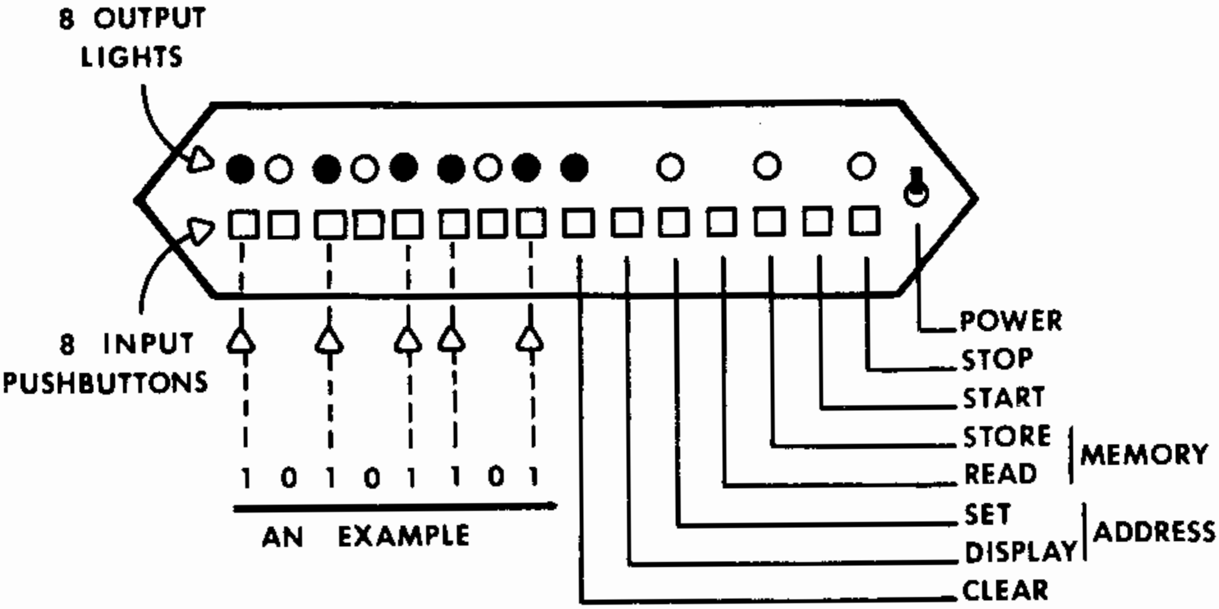


Figure 1.2 The Front Panel of the Computer

Let's see how we can enter data into a memory location. We must do two things. First, we must tell the computer the address of the memory location to be used. Second, we must tell the computer what our data to be stored is. Let's suppose we want to store the data 00111000 into the location whose address is 10101101. First push Clear, then use the eight input pushbuttons to enter the address. The picture shows this number being entered. As each 1 is entered, the output light above it will turn on. If you make a mistake, use Clear to erase all of the 1's and try again. When the output lights show that you have entered the number correctly, you can go on. To tell the computer that this is an address, push the Set Address pushbutton.

Now we can enter the data to be stored. To do this, push Clear. This erases the number which was our address. (When we pushed the Set Address button, the computer made a copy of this number in another place.) Now enter the data, pushing the input buttons which correspond to 1's. In our example, (data = 00111000), these are the 3rd, 4th, and 5th positions from the left. When these are correctly entered, we can store the data. To do that, push Store Memory.

How do we know it was done right? Did the computer really do what we had told it to do? Let's read what is in location 10101101.

To read the data in a location, the address of the location is entered first. Push Clear and then the input buttons which correspond to the 1's in the address. When the number is correctly entered, use the Set Address pushbutton to tell the computer that this is an address. After doing this, push Read Memory. The data in the memory location whose address you just entered will be shown in the output lights. In our example this should be 00111000.

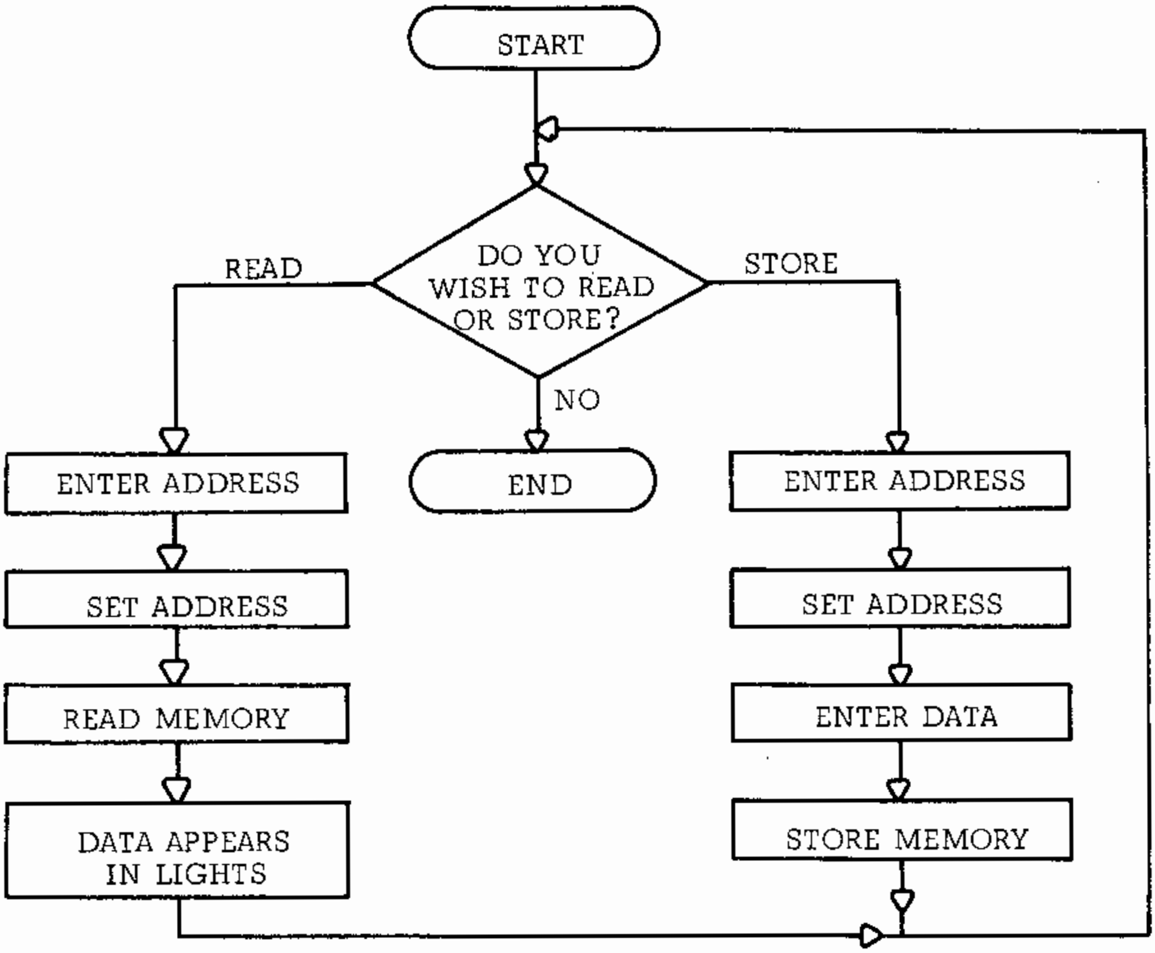


Figure 1.3 Can You Find Your Way Around The Boxes? What Does It Tell You?

Try entering the following numbers:

In the location whose address is	Store the data
10000001	11001100
01010011	11010001
00000000	00001111
00001111	10101010
11111110	00000000
00101100	00000001
00001010	00000011
00001011	00000100

After you have entered the information, read these same locations and see if the numbers are there. They should be. Read them a second time and see if they are still there. Reading or copying is non-destructive. The number remains, it is not destroyed.

Try this experiment. Store the data below. Read the data to check your entry. Turn Power off. Turn Power on. Push Start. Push Stop. Now read the contents of these locations and fill in the blanks.

Location	Data	After Power is Off and On
00000000	11001100	____________
11110000	00110011	____________
00001111	11001100	____________

Has the data changed? ________________________________

Do you know why? ________________________________

Try this also. Put the data 00000000 into location 11111111. Now read the contents of location 11111111.

What did you read? ________________________________

This is a special location and we will learn more about it later.

Put the small toggle switch in the LOCK position and try to enter data.

Can you enter data? _________ Can you read data? _________

Exercise 2

In Exercise 1, we learned how to read and to store one number in one memory location. We used different addresses, but the memory locations were not adjacent. They did not follow one after the other. Usually, we will read or store numbers in locations that follow one after the other. Their addresses are in a continuous sequence like 2, 3, 4, 5, etc. The computer contains a feature which makes it easier to read or store in locations that follow after each other.

Whenever the Read or Store pushbutton is depressed, the address counts by one inside the computer.

The address is a binary number. We haven't discussed how binary numbers count though we will shortly. For the time being we don't need to know what a consecutive sequence of binary numbers looks like.

To read the contents of a number of locations whose address are consecutive (like 17, 18, 19, 20, 21), the address of the smallest one is entered (including Set Address). Depressing the Read pushbutton once causes the data in that location to be displayed. Pushing the Read pushbutton again causes the data in the next location to be displayed. We can repeat this for as many times as there are locations to be read, Figure 2.1

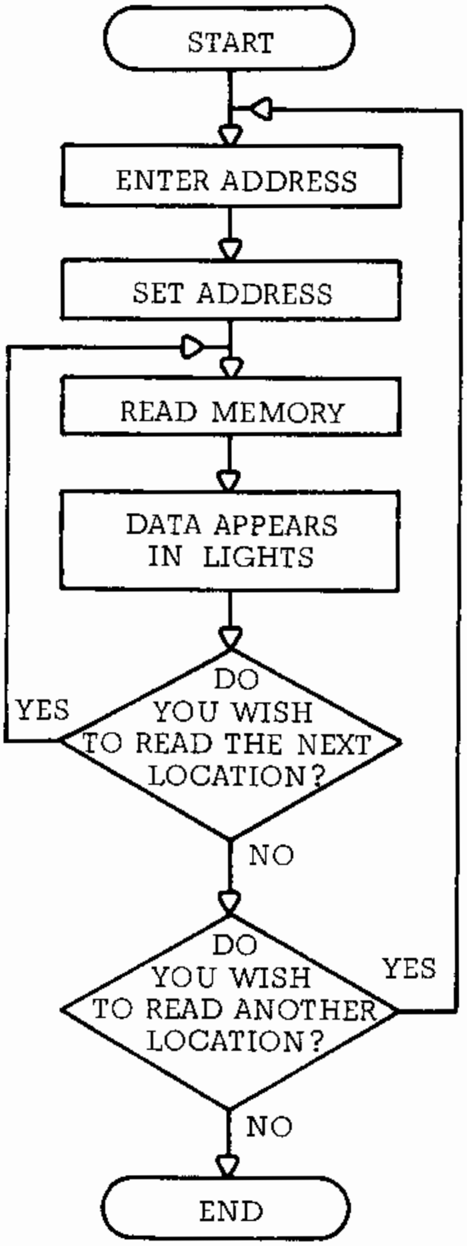

START
ENTER ADDRESS
SET ADDRESS
READ MEMORY
DATA APPEARS IN LIGHTS
DO YOU WISH TO READ THE NEXT LOCATION?
YES
NO
DO YOU WISH TO READ ANOTHER LOCATION?
YES
NO
END

Figure 2.1 Reading

To store numbers in consecutive locations, again the address of the smallest one is entered. Then the data to be stored in the first location is entered with the input buttons and the Store pushbutton is used once. The address is automatically advanced to the next location by the computer. The data to be stored in this second location is entered in the input buttons (after clearing) and Store is depressed again. The process is repeated until all of the data has been stored, Figure 2.2.

Let's try it on the computer, but first let's review whether you remembered how to store data in one location and to read it back. Into location 01000111 put the data 10111000. Read the contents of location 01000111. Does it contain 10111000, the data which you had put there? If you were unsuccessful or didn't remember how, you should go back and review Exercise 1 before continuing.

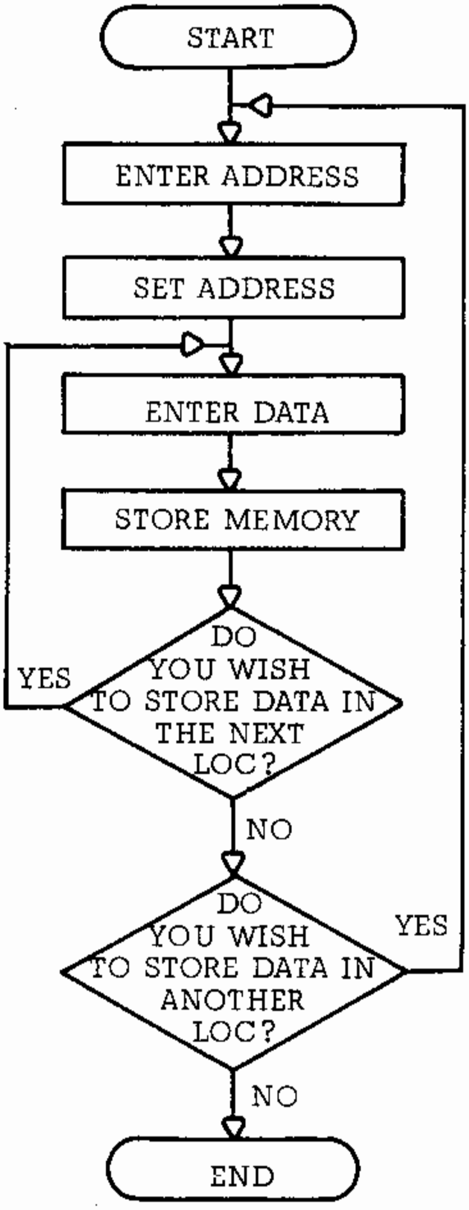

START
ENTER ADDRESS
SET ADDRESS
ENTER DATA
STORE MEMORY
DO YOU WISH TO STORE DATA IN THE NEXT LOC?
YES
NO
DO YOU WISH TO STORE DATA IN ANOTHER LOC?
YES
NO
END

Figure 2.2 Storing

Have you noticed that it is hard to keep track of your place when you are reading our binary numbers? It is easier to read the numbers if the eight 0's and 1's are placed in groups. Instead of

11100100

the grouping 11 100 100

is easier to read. You'll make fewer mistakes if you do it this way.

Starting with location 00 000 011, store the following data:

In Location	Store
00 000 011	00 000 100
the next location	00 000 011
the next after that	00 000 001
and the next	00 011 100
and so on	10 000 000
	11 100 100
Stop after this one	00 000 100

Check yourself by reading the contents of the seven locations which start with location 00 000 011. Do they agree with the data above? If they do not agree, try again. Try to find out why you went wrong.

When you are able to store the numbers above and to read them correctly, and while the data is in the computer, then try this experiment. Push Start and release it.

What happens to the display lights? ____________________

Do you think there is a pattern? ____________________

Push Stop. What happens? ____________________

After you have stopped the computer, try this experiment. Set the address to 11 000 000. Enter the number 11 111 111 in the input buttons and push Store five times. Then push Clear which will make the entry number 00 000 000. Push Store five more times. Now set the address to 11 000 000 (the same as before) and read the contents of the ten locations that start with this address.

Did you find that 11 111 111 was stored in the first five locations and that 00 000 000 was stored in the next five locations? ____

If you are storing the same data into consecutive addresses, must you re-enter the number with the input buttons after the first entry? ____

Exercise 3

In the first two exercises we learned how to store numbers in the memory and how to read them. The numbers were binary, not our familiar decimal. Binary numbers use only the digits 0 and 1. A binary digit has been given the special name, "bit". A bit may be a 0 or 1 but it is never a 2, 3, 4, 5, 6, 7, 8, or 9.

In Exercise 2, you were asked to store a series of numbers and, after checking the entry by reading these same numbers, you were asked to push Start. The output lights should have started blinking. The numbers you stored in the memory were instructions or commands to the computer to do some simple operations. This set of instructions was a program. The computer was using this program over and over. As soon as it finished doing the operations once, the computer went back to the beginning of the program and did the operations again.

In this exercise we will load another program which is very similar to the program in Exercise 2. In this program the computer halts after it does the operations once. It will do them once again when you push Start. As often as you push Start, it will do the required operations once and then halt.

Let's load the program and see what is does. Starting with location 00 000 011, store the following numbers in the memory:

Location	Store
00 000 011	00 000 100
next	00 010 011
next	00 000 000
next	00 011 100
next	10 000 000
next	00 000 000
next	00 000 011
next	00 000 001
next	11 100 100
next	00 000 110

Read the numbers you entered. Every bit must be right. If there are any errors, correct them by entering all of the numbers.

Push Start once. The output lights should all be out. If you did not get this result, or if the Run light did not go out, you have probably entered the numbers wrong. Try again, re-enter all of the numbers.

Push Start once. The lights should now be oo ooo oo● .

Push Start once again. The lights should now be oo ooo o●o .

Push Start a few more times, observing how the lights change. Can you tell in advance which lights will turn on and which lights will turn off? Try to make up rules which tell you how the lights will change. Push Start and see if the lights do change in the way that your rule says they will. Try this several times. You may have to change your rules but keep trying until you can always tell what the next pattern in the lights will be.

If you want to start at the beginning again, put the number 00 000 100 into location 00 000 011.

You have been asked to find a rule about how the lights change. If you did find a rule which told you what the next pattern in the lights would be, you have discovered for yourself how to count in binary.

The rule which tells how the lights change might read:

> Find the first light at or from the right hand end that is Off. This light will change to On. All of the lights from the right hand end up to this light, but not including this light, will change to Off. None of the other lights will change.

Put the number 00 000 100 into location 00 000 011. This will start you at the beginning again. Try using the rule above to predict what the next pattern in the lights will be as you push Start.

The rule above also tells us how to count in binary. All that we have to do is to change the words "light" to "bit", "On" to 1, "Off" to 0. Then, given a binary number, the next binary number is determined by the rule:

> Find the first bit at or from the right hand end that is 0. This bit will change to 1. All of the bits from the right hand end up to this bit, but not including this bit, will change to 0. None of the other bits will change.

Let's apply this to some binary numbers.

```
                                This is the first 0 bit from the
This number     00 000 111         right end. Change it to 1.
                                All of these change from 1 to 0.

   counts to    00 001 000

                                This is the first 0 bit from the
This number     00 001 000         right end. Change it to 1.
                                There are no bits to change from
                                   1 to 0.
   counts to    00 001 001

                                This is the first 0 bit from the
This number     00 001 001         right end. Change it to 1.
                                All (in this case, one) of these
                                   change from 1 to 0.
   counts to    00 001 010
```

Put the number 00 000 100 into location 00 000 011. Push Start once. The lights should show you 00 000 000. We have written this number down already in the Table below. Before you push Start again, write down what the next number will be. Push Start and check yourself. Then write down the next number and use Start to check your answer. Repeat these steps until you have filled in the table.

0	0	0	0	0	0	0	0	0
								1
								2
								3
								4
								5
								6
								7
								8
								9
								10
								11
								12
								13
								14
								15
								16
0	0	0	1	0	0	0	1	17

The correct results are given on the next page. Check your numbers against these.

00 000 000	0
00 000 001	1
00 000 010	2
00 000 011	3
00 000 100	4
00 000 101	5
00 000 110	6
00 000 111	7
00 001 000	8
00 001 001	9
00 001 010	10
00 001 011	11
00 001 100	12
00 001 101	13
00 001 110	14
00 001 111	15
00 010 000	16
00 010 001	17
00 010 010	18
00 010 011	19
00 010 100	20
00 010 101	21
00 010 110	22
00 010 111	23
00 011 000	24
00 011 001	25
00 011 010	26
00 011 011	27
00 011 100	28
00 011 101	29
00 011 110	30
00 011 111	31

Table 3.1 First Few Binary Numbers

Exercise 4

Using Table 3.1 from the last Exercise, answer these questions.

What decimal number is beside the binary number 00 000 001?______

What decimal number is beside the binary number 00 000 010?______

What decimal number is beside the binary number 00 000 100?______

What decimal number is beside the binary number 00 001 000?______

What decimal number is beside the binary number 00 010 000?______

Write the sequence of decimal numbers which are the answers to these five questions (start with 1).

__________ __________ __________ __________ __________

What do you think the next decimal number would be?____________

What binary number is the same as this number?_________________

What do all of these binary numbers have in common?____________

The decimal sequence is 1, 2, 4, 8, 16 and the next number in this sequence is 32. This is the binary number 00 100 000. All of these binary numbers have one and only one bit that is a 1.

Now look in the Table at the binary number beside 3. Can you see that it has two bits which are 1's? Can you find the two binary numbers that have single 1's in these positions? Here they are:

```
00 000 001        1
00 000 010        2
00 000 011        3
```

What is the relationship between the number 3 and the other two decimal numbers?

__________ + __________ = __________

Let's try another binary number. Take the binary number opposite 6. Which bits are 1's in it? Find the binary numbers that have single 1's in the same positions as these bits.

What are the decimal numbers? __________ and __________

How are these numbers related to 6?_________________________

Do the same with the binary number 00 001 101. Find the three binary numbers with single ones in these positions.

What decimal numbers are opposite these? ______, ______, ______

What is the sum of these three numbers? ______________________

What decimal number is beside 00 001 101? _________________

Use this same process and find what decimal numbers are equal to the binary numbers:

1001 ___________________________

111 ___________________________

1011 ___________________________

We know in a decimal number such as 2,736 that this does not mean 2 + 7 + 3 + 6. We understand that the 2 is to be multiplied by a 1000, the 7 by 100, the 3 by 10 and the 6 by 1. We call 1, 10, 100, and 1,000 "place values".

Binary numbers have place values also. They are the sequence

1, 2, 4, 8, 16, 32, 64, 128, 256, and so on.

A binary number may be converted to decimal by adding up the place values where the binary number has 1's. Let's do it for a few binary numbers.

Place Values	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>	
Binary Number	1	0	0	1	8 + 1 = 9
Binary Number		1	1	1	4 + 2 + 1 = 7
Binary Number	1	0	1	1	8 + 2 + 1 = 11

Earlier we asked you to convert these same numbers to decimal. Did you get the answers shown here? In the second binary number above, we didn't put any bit under the 8's place value. If we put a bit there, what would we put there?

Below is a small program which is a game you can play against the clock. What you must do is to give the decimal number for a binary number which the computer first displays to you. If you are right, the computer will give you another different binary number. If you are wrong, the computer will give you the same binary number again.

Load the following program. Notice that the addresses are consecutive so you won't have to keep re-entering addresses. Double check your entries by reading. If you have a mistake, you don't need to re-enter all of the numbers. Use the address given and correct the data which is in error.

Location	Data	Location	Data
00 000 000	00 000 000	00 001 101	11 010 001
00 000 001	00 000 000	00 001 110	00 011 100
00 000 010	00 000 000	00 001 111	10 000 000
00 000 011	00 000 110	00 010 000	10 011 100
00 000 100	00 000 000	00 010 001	11 111 111
00 000 101	11 111 111	00 010 010	00 000 000
00 000 110	01 000 011	00 010 011	11 011 001
00 000 111	00 101 101	00 010 100	00 000 011
00 001 000	00 010 011	00 010 101	10 000 010
00 001 001	00 111 000	00 010 110	00 011 100
00 001 010	11 010 100	00 010 111	00 000 100
00 001 011	00 000 001	00 011 000	11 100 100
00 001 100	11 011 001	00 011 001	00 000 100

Push Start. The computer should display oo ooo ●o● in the output lights. This is the binary number for 5. You should push switch number 5, □□ ■□□ □□□. When you do this, the binary number will disappear and the light above the switch will come on. Now push Start. If the entry is correct, the computer will display another, different, binary number. If your entry was wrong, the computer will display the same binary number again. Again make your entry and push Start.

The largest binary number the computer will show you is seven. The smallest is 0. The first time you see the number 0 given by the computer you may think something has gone wrong. It hasn't. Try a few numbers to see how it goes, then get a friend to time you against the clock. How many correct ones can you do in a minute?

Exercise 5

At the end of the last exercise we were converting binary numbers to decimal. In the first part of this exercise we will do the opposite. The computer will give you a decimal number from 0 to 7. You are to enter the binary number which is equal to this decimal number.

The program to use is:

Location	Data	Location	Data
00 000 000	00 000 000	00 010 000	10 011 100
00 000 001	00 000 000	00 010 001	10 000 000
00 000 010	00 000 000	00 010 010	10 011 100
00 000 011	00 000 100	00 010 011	11 111 111
00 000 100	01 000 011	00 010 100	00 000 000
00 000 101	00 101 101	00 010 101	10 000 000
00 000 110	00 010 011	00 010 110	00 000 000
00 000 111	00 111 000	00 010 111	00 010 100
00 001 000	11 010 100	00 011 000	11 111 111
00 001 001	00 000 001	00 011 001	11 011 001
00 001 010	00 011 011	00 011 010	00 001 100
00 001 011	00 000 000	00 011 011	00 001 011
00 001 100	00 000 011	00 011 100	00 100 100
00 001 101	01 000 010	00 011 101	00 000 100
00 001 110	00 011 100	00 011 110	11 100 100
00 001 111	00 010 100	00 011 111	00 000 110

Did you check your entries? Push Start. The computer will display oo ●oo ooo . Since light number 5 is on, you should enter the binary number □□ □□□ ■□■ in the switches. Push Start again. If your input was correct, the computer will display another, different decimal number. If you were wrong, the computer will display the same decimal number again. Try a few times to see how it goes and then have a race against the clock. With practice you can learn to use two and three fingers at the same time to enter the binary number.

There are other number systems besides binary and decimal. One of these is called hexadecimal. There are sixteen digits in hexadecimal numbers. These are the ten digits of decimal numbers and the six letters, A, B, C, D, E, and F. In hexadecimal, F + F = 1E. The hexadecimal system is closely related to binary, but we won't use the hexadecimal system.

Another number system is called octal. The octal system has eight digits: 0, 1, 2, 3, 4, 5, 6, 7. You have been working with the octal system already. For numbers from 0 to 7, the decimal and octal numbers are the same. The conversion problems we have been doing are true for both decimal and octal.

We will use the octal system a lot. It is very similar to binary and it will make our work much easier. Though you may not be aware of it, you already know how to convert a binary number to an octal number and vice versa.

To convert a binary number to octal, divide the binary number into groups of 3 bits starting at the right, for example,

```
001 101 110 010
```

Now write down the 0 to 7 octal digit that is equal to each 3 bit binary group above. In this case we would have

```
 1   5   6   2
```

To convert an octal number to binary, do this in reverse

```
 3   7   4   0
011 111 100 000
```

We won't learn any arithmetic rules for octal. We can convert them to binary and do the arithmetic in binary. However, if there are no carries or borrows, then octal arithmetic is the same as decimal arithmetic. For example, all of these additions and subtractions are true for both octal and decimal numbers:

$$\begin{array}{rr} & 250 \\ + & \underline{5} \\ & 255 \end{array} \qquad \begin{array}{rr} & 267 \\ - & \underline{3} \\ & 264 \end{array} \qquad \begin{array}{rr} & 146 \\ + & \underline{31} \\ & 177 \end{array} \qquad \begin{array}{rr} & 357 \\ - & \underline{113} \\ & 244 \end{array}$$

An octal number such as 250 doesn't represent the same quantity of things as the decimal number 250 does. For example,

Octal 10	is this many things:	●●●●●●●●
Decimal 10	is this many things:	●●●●●●●●●●

There is one arithmetic operation with octal numbers that we use a lot. That is counting. The easiest way of counting in octal is to count in decimal, but to omit all of the decimal numbers which have an 8 or 9. Complete the counts below. (Note that the counts go down the page.)

5	14	46	75	112	035	207
6	____	____	____	____	____	____
7	____	____	____	____	____	____
10	____	____	____	____	____	____
11	____	____	____	____	____	____
12	____	____	____	____	____	____
13	____	____	____	____	____	____

With so many different kinds of numbers, how can one tell what number system is being used? For example, if you see 111 are you to think of seven (in binary) or seventy-three (in octal) or one hundred eleven (in decimal)? We may depend upon the context or the setting in which we see it. As a price in a store, we know it is decimal. In the computer we are working with, the number is usually octal. Sometimes when we could be confused we will do one of the following:

binary 111 or 111_2 or 111_{two}

octal 111 or 111_8 or 111_{eight}

decimal 111 or 111_{10} or 111_{ten}

FOR INTERESTED STUDENTS (OTHERS NOT ALLOWED)

Bi- means two. Binary numbers use two symbols.
Oct- means eight. Octal numbers use eight symbols.
Deci- means ten or tenth. Decimal numbers use ten symbols.
Hex- means six. Hexadecimal (6 + 10) numbers use sixteen symbols.

Can you give some other words that use bi-, oct-, deci-, or or hex- ?____

____

What is odd about October and December ?________________

____

How many sides does a triangle have? ______

What might you call the number system with three symbols?

______

Of all the number systems that are possible, why do you think we use decimal numbers? ______

Could there be a number system with only one symbol? ______

How many symbols does the tally system (𝍸 ||||) use? ______

Does the tally system have place values? ______

In decimal, the place value names are one, ten, hundred, thousand, etc. There are no agreed upon names for the place values in binary, octal, and hexadecimal. We sometimes read the octal number 1,750 as "octal one thousand seven hundred and fifty."

What decimal number is this? ______

Must the binary system use the symbols 0 and 1? ______

Could it use the symbols a and b? ______

Why would you prefer one set of symbols to the other? ______

______

Below we give the addition table for octal numbers. Fill in the empty squares.

+	0	1	2	3	4	5	6	7
0	0	1	2	3				
1	1	2	3					
2	2	3						
3	3							
4								
5								14
6							14	15
7						14	15	16

Exercise 6

In this Exercise, we'll learn how to add two binary numbers. Again, we'll use the computer and a program to help us. The program will add our input number (entered in the input switches) to a number already in the computer. This number in the computer will be stored in an "accumulator".

```
  OLD CONTENTS OF ACCUMULATOR
+ ______OUR INPUT NUMBER______
  NEW CONTENTS OF ACCUMULATOR
```

The addition takes place when we push Start. When the computer stops, it will display the new contents of the accumulator in the lights. As soon as we push Clear or one of the input buttons, the lights will then show the input number. When we want to clear the accumulator (set it to zero), we'll use input switch 7

■□ □□□ □□□ and Start clears the accumulator

Let's load the program and see how it works. Notice that the addresses and data have been given in both binary and octal. Try using the octal when you enter the data. When you read the data to check it, you should use the binary. After a while we will use only octal numbers, but until you are familiar with the octal system we'll also put the binary number down.

Location		Data	
00 000 000	000	000	00 000 000
00 000 001	001	000	00 000 000
00 000 010	002	000	00 000 000
00 000 011	003	021	00 010 001
00 000 100	004	000	00 000 000
00 000 101	005	272	10 111 010
00 000 110	006	377	11 111 111
00 000 111	007	344	11 100 100
00 001 000	010	021	00 010 001
00 001 001	011	004	00 000 100
00 001 010	012	377	11 111 111
00 001 011	013	034	00 011 100
00 001 100	014	200	10 000 000
00 001 101	015	134	01 011 100
00 001 110	016	377	11 111 111
00 001 111	017	344	11 100 100
00 010 000	020	004	00 000 100
00 010 001	021	023	00 010 011
00 010 010	022	000	00 000 000
00 010 011	023	344	11 100 100
00 010 100	024	013	00 001 011

Push Clear. Push Start. The lights should be out.

Push switch 0. Push Start. Repeat these two several times. Do you see the binary count process as we add one each time? Push switch 7. Push Start. The accumulator should now be zero. Add some number to the accumulator. Then push Clear. Push Start. Did the accumulator change? Are we adding zero to the accumulator?

What we want to do now is to find rules for adding binary numbers on paper. The computer will be used to tell whether we are right or wrong. We can try our ideas with it. Don't change the binary numbers to decimal and add. Do all of the work in binary. Try to do this in an orderly way and organize your approach. Start with simple cases and little numbers (like 0 and 1). Test your ideas with the computer. After you have tried awhile, come back and finish reading the rest of this exercise.

There is more than one way to add. You may have found a way that is different from the one to be given here. That doesn't make yours wrong.

When we learned to add decimal numbers, one of the first things we learned was the addition table. With decimal numbers, it was necessary to learn 100 combinations. One such combination was 8 + 6 = 14. How many combinations do you think there would be in binary? With 10 digits for decimal numbers, there were 10 times 10 cases to be learned. With only 2 digits for binary numbers, there are only 2 times 2 cases to be learned. And of these four cases, three of them consist of adding 0 to another number.

The four cases are:

$$\begin{array}{rrrr} 0 & \quad 0 & \quad 1 & \quad 1 \\ +\underline{\;0} & \quad +\underline{\;1} & \quad +\underline{\;0} & \quad +\underline{\;1} \end{array}$$

If you know the answers to these four problems and if you know how to "carry", then you know how to add two binary numbers. Zero plus zero is zero. Zero plus one is one. One plus zero is one. One plus one is two. The number two is written in binary as 10. So we have:

$$\begin{array}{rrrr} 0 & \quad 0 & \quad 1 & \quad 1 \\ +\underline{\;0} & \quad +\underline{\;1} & \quad +\underline{\;0} & \quad +\underline{\;1} \\ 0 & \quad 1 & \quad 1 & \quad 10 \end{array}$$

Adding one to one creates a carry in binary. The carry for binary numbers is used in the same way as for decimal numbers. It adds one more in the next column to the left. Let's take the four cases above and also add in a carry

$$\begin{array}{rcrcrcrcl} 1 & \longleftarrow & 1 & \longleftarrow & 1 & \longleftarrow & 1 & \longleftarrow & \text{carry from the right} \\ 0 & & 0 & & 1 & & 1 & & \\ +\underline{\;0} & & +\underline{\;1} & & +\underline{\;0} & & +\underline{\;1} & & \\ 1 & & 10 & & 10 & & 11 & & \text{carry to the left} \end{array}$$

Notice that the sum, including the carry, is the binary number for the number of 1's being added together. For example,

$$\begin{array}{rl} 1 & \\ 1 & \quad \text{3 1's being added} \\ +\underline{\;1} & \\ 11 & \quad = \text{3 in binary} \end{array}$$

Let's add two binary numbers

$$\begin{array}{rcccccccccl} & 1 & 1 & 1 & & 1 & & & & & \text{carry} \\ & & 1 & 0 & 1 & 0 & 1 & 0 & 1 & & \text{first number} \\ + & & \underline{1} & \underline{1} & \underline{1} & \underline{0} & \underline{1} & \underline{1} & \underline{0} & & \text{second number} \\ & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & & \text{answer} \end{array}$$

Give the sums in the following additions and then check your own work with the computer.

$$\begin{array}{rrrrrr} 1\,0\,0 & \quad 1\,0\,0\,1 & \quad 1\,0\,1\,0\,1 & \quad 1\,1\,0\,1 & \quad 1\,0\,0\,1 & \quad 1\,1 \\ +\underline{1\,0\,0\,0} & \quad +\underline{\;\;1\,0\,0} & \quad +\underline{\;\;\;\;1\,1\,0} & \quad +\underline{\;\;\;\;1\,1} & \quad +\underline{\;\;1\,1\,1} & \quad +\underline{1\,0\,1\,1} \end{array}$$

While the program is in the computer, try these experiments. Put 000 in the accumulator, then keep adding 2. These are the even numbers. Write the first few even numbers in binary.

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

Now put the number 001 in the accumulator, then keep adding 2. These are the odd numbers. Write the first few odd numbers in binary.

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

___ ___   ___ ___ ___   ___ ___ ___

What do all of the even numbers have in common that the odd numbers do not have?______________________________

Do this addition

```
  1 0 1 1 0
+ 1 0 1 1 0
  ---------
```

How does the sum look compared to the numbers being added?

____

Adding a number to itself is the same as multiplying by 2. Does multiplication by 2 for binary numbers have something in common with multiplication by 10 for decimal numbers?______________________

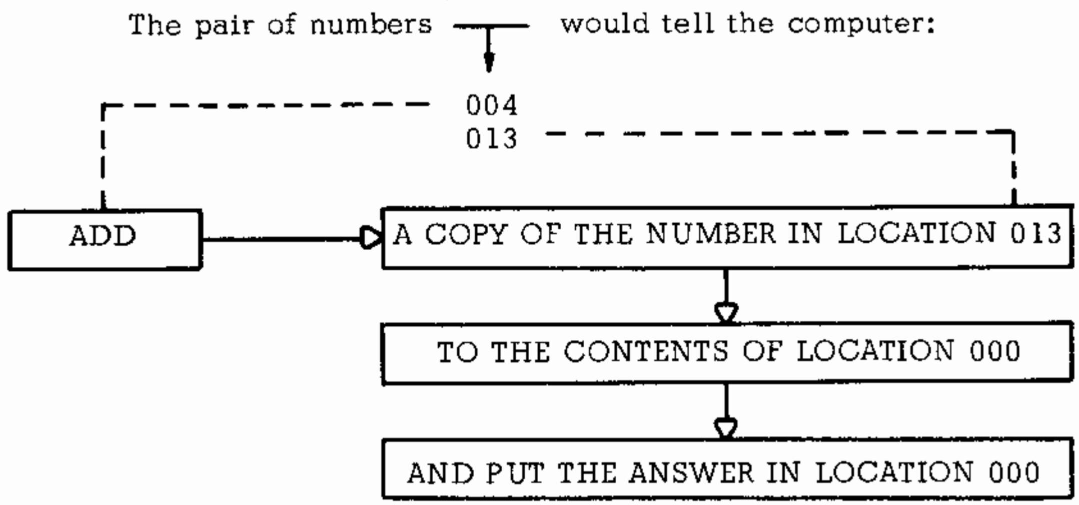
What is the number 2 in binary?____________________

Appendix I may be used to convert octal numbers to decimal or decimal numbers to octal.

Exercise 7

We have been using programs in the computer. A program consists of instructions which tell the computer what to do. An instruction is a number in the computer. Different numbers tell the computer to do different things. In our computer some of the instructions require two memory locations while other kinds of instructions require one memory location.

The pair of numbers would tell the computer:


004
013
ADD
A COPY OF THE NUMBER IN LOCATION 013
TO THE CONTENTS OF LOCATION 000
AND PUT THE ANSWER IN LOCATION 000

If we change the 004 to another number, we would get a different instruction, perhaps a Subtract. The 004 is the operation code. The second number in this pair is an address of a memory location. While in this example it is 013, it could be any address from 000 to 377. With this instruction we must always use the number in location 000 as one of the operands and we must use location 000 to receive the answer. We can't change that. The pair of numbers must be stored in consecutive locations such as 100 and 101. The 004 number, the operation code, must be in the smaller of the two addresses. The memory address, such as 013, is in the larger of the two addresses.

When we put parentheses around an address, for example,

(013)

we read this as "the contents of location 013".

When we use the notation

(000) : (013)

we mean that a copy of the number in location 013 is to be made and this value becomes the contents of location 000. Notice that this kind of statement is read from right to left. We will also use statements like the following

(000) : (000) + (013)

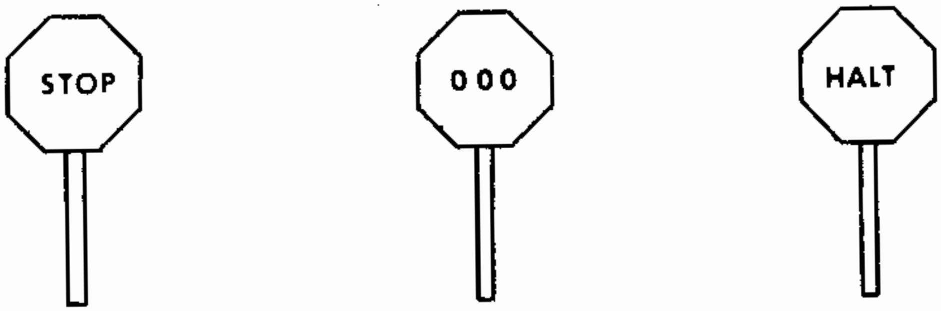
It says, "Add the contents of location 013 to the contents of location 000 and then put the answer in location 000". This statement, of course, describes our Add instruction above.

Since the Add instruction can use any location in memory for the number to add to the number in location 000, we could express the statement as

(000) : (000) + (XXX), XXX = 000, 001,.......,377

The statement on the far right merely says XXX can be any of the memory addresses. Usually we will omit it. It's just understood that it is so.

The single number 000 is an instruction to the computer telling it to stop or halt. We call this an instruction "Halt". Halt is an instruction in the computer and the program tells the computer to halt. We'll use the name "Stop" for the action of telling the computer, from the front panel, to stop. It does not require any instruction. Both Halt and Stop produce the same result. When the computer is halted or stopped, it isn't doing anything. The Run light is out and we can store and read memory locations. When the Run light is on, the computer is doing the steps of a program in the computer. We can't read and store data using the front panel then.



A computer can do only one instruction at a time. After doing one, it will do another, followed by another, and so on. How does the computer know where to begin? And how does it know where the next instruction is?

THE NUMBER IN LOCATION 003 TELLS THE COMPUTER
WHICH MEMORY LOCATION HOLDS THE NEXT INSTRUCTION.

After an Add instruction, the number in location 003 is increased by 2 since the Add instruction is contained in two locations in the memory. After a Halt instruction, the number in location 003 is increased by 1 since the Halt instruction uses only one location in the memory. This adjustment to the value of the number in location 003 is done by the computer automatically. We could express this symbolically,

for ADD,	(003) : (003) + 2
for HALT,	(003) : (003) + 1

WARNING A common mistake that students make is to say that the number in location 003 is the next instruction. This is not correct. The number in location 003 is the address of the next instruction. There is a big difference between these two statements. Be sure you understand the difference.

• **Let**'s try the Add and the Halt instructions in the computer. Load the data in the Before column.

Location	Before	After	Comments
000	250	□	
003	100	□	
100	004	□	ADD (000) : (000) + (164)
101	164	□	(003) : (003) + 2
102	000	□	HALT (003) : (003) + 1
164	005	□	

Push Start. The Run light should go out. Read the memory locations and fill in the blanks in the After column above.

Did the contents of location 000 change?

By how much did they change?

What location holds this number?

Did the contents of location 164 change?

Did the contents of location 003 change?

What do the contents of location 003 now tell the computer?

Did (100) or (101) or (102) change?

Now here is a problem for you to program. Write a four instruction program to add together the numbers in location 301, in 302, and in 303. You can choose your own numbers to put in these locations. Pick small numbers. We've written out a few of the locations and data for you. Try your program in the computer. When the computer stops, you'll have to read the contents of location 000 to get your answer.

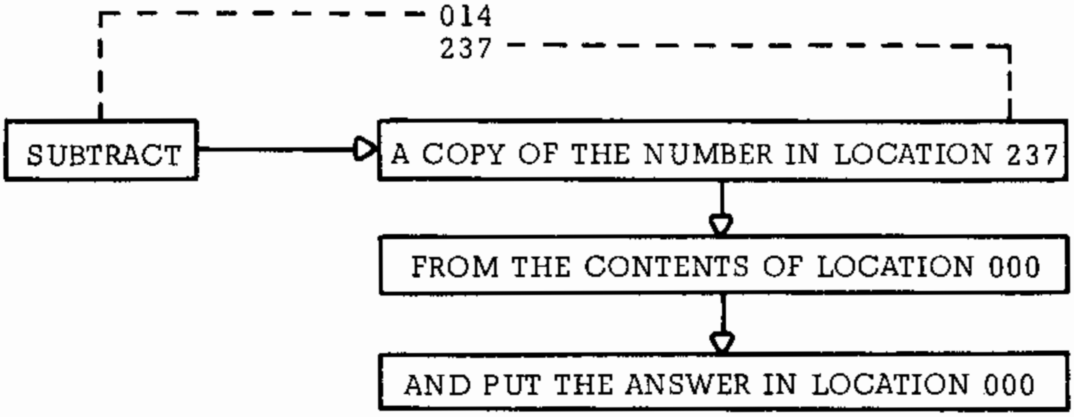
Location	Data	Comments
000	000	Start with zero in location 000
003		
100		Put first instruction here
101		
102		
103		
104		
105		
106		HALT
301		First number
302		Second number
303		Third number

Try a different set of numbers in 301, 302, and 303. Don't forget to put the proper starting values in locations 000 and 003.

Exercise 8

We'll learn some more instructions in this Exercise.

This pair of numbers would tell the computer:


014
237
SUBTRACT
A COPY OF THE NUMBER IN LOCATION 237
FROM THE CONTENTS OF LOCATION 000
AND PUT THE ANSWER IN LOCATION 000

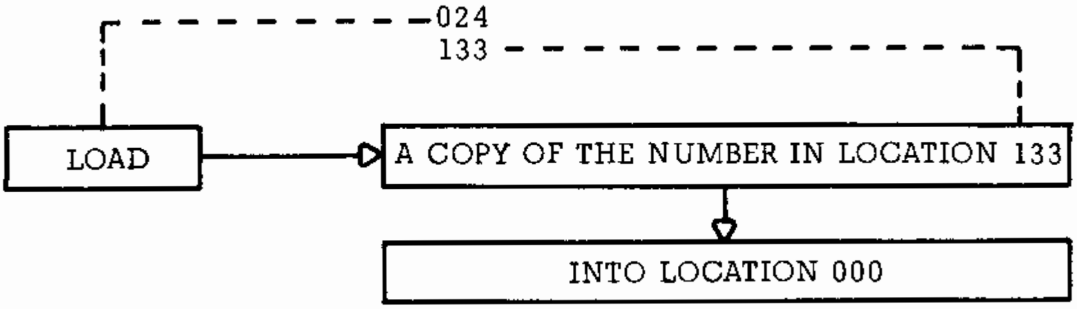
We can also give the results produced by this instruction in this way,

(000) : (000) - (XXX)

(003) : (003) + 2 SUBTRACT

Operation code is 014

And this pair of numbers would tell the computer:



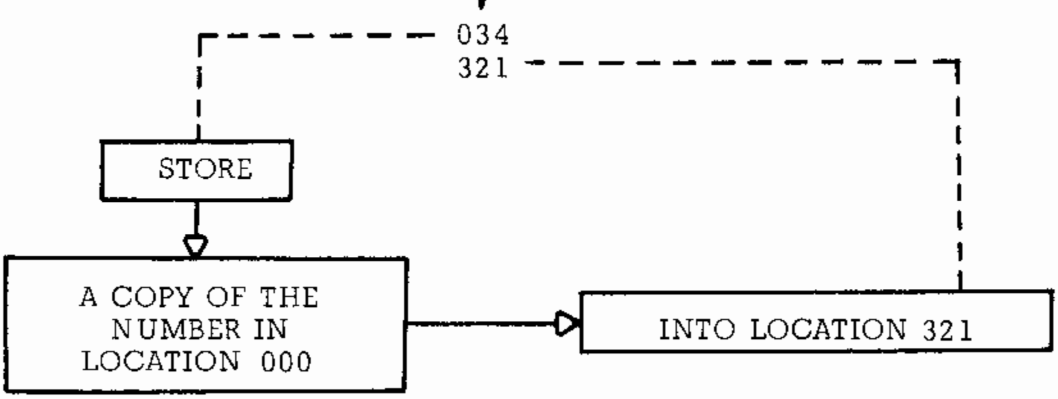
LOAD is a transfer of data to location 000 from another location. The data is not changed in any way. Our shorthand notation for Load is,

(000) : (XXX)

(003) : (003) + 2 LOAD

Operation code is 024

The pair of numbers would tell the computer:



STORE is a transfer of data from location 000 to another location. The data is not changed. You may want to compare this instruction with LOAD. We can express the Store instruction,

(XXX) : (000)

(003) : (003) + 2 STORE

Operation code is 034

The Add, Subtract, Load, and Store instructions all use memory location 000. It's beginning to look as if location 000 is special. It is. It is so special that

MEMORY LOCATION 000 IS CALLED THE "A REGISTER"
and THE (000) ARE CALLED "A"

The A Register is important in the Arithmetic unit.

Location 003 is another very special location. In order that its feelings won't be hurt, we have given it a name also.

MEMORY LOCATION 003 IS CALLED THE "P REGISTER"
and THE (003) ARE CALLED "P"

The contents of the P Register point to the next instruction in the program.

In our shorthand notation we could say, for addition,

A : A + (XXX) , P : P + 2

Whether these are any better or any worse than

(000) : (000) + (XXX) , (003) : (003) + 2

we leave to your judgement.

Let's write another small program. Add the numbers in location 140 and 141 and from this sum subtract the number in 142. Before doing these arithmetic operations, A is to be set to 000 by the program. After the arithmetic operation, store A in location 200. This will cause the answer to appear in the output lights. Finally, we will halt.

Notice how this last paragraph could be condensed,

```
1.     A : 0
2.     A : A + (140)
3.     A : A + (141)
4.     A : A - (142)
5. (200) : A
6. HALT
```

Each of these six statements can be done with one instruction.

Write a program (see below) from the six statements above. Choose your own data and load them with the program and try it.

Location	Data	Comments
003	320	P = 320
∿	∿	∿
320		
321		
322		
323		
324		
325		
326		
327		
330		
331		
332		
∿	∿	∿
140		First number
141		Second number
142		Third number
143	000	Zero

After you have written your program you will probably have a 000 in location 332. This 000 will be used as a Halt instruction. We suggested also putting 000 in location 143. This 000 was intended for use as data to clear the A Register. Would one 000 be enough? Yes, we could use only one. But it is poor practice to needlessly mix the instructions and data and make one number serve for both purposes. Try to avoid it.

Exercise 9

First let's learn one more instruction. The number 200, as an instruction, tells the computer to do nothing. Don't laugh, the Do Nothing instruction (also called No Operation or NOOP) is very useful. Symbolically the instruction causes

P : P + 1 NOOP

Operation Code is 200

That is, the contents of the P Register advance by 1. This action in itself can be important to us.

We are going to use a new feature on the computer which we haven't used before. From the front panel we are going to make the computer start, do one instruction, and stop. We already know that the <u>program</u> can have the computer stop with a Halt instruction. From the front panel <u>we</u> can make the computer execute one instruction and stop regardless of what this instruction is. This is called "Single Instruction" and when we tell you to do a single instruction you should do this:

Single Instruction

1. Push <u>and hold</u> Stop
2. Push Start
3. Release Start
4. Now release Stop

Try this a few times to learn the pattern of the actions that are required. The single instruction feature is very helpful when you are studying what is happening in the middle of a program. You guessed it, that is what we are going to do.

Load this program. Do not push Start. Answer the questions below. If you have trouble answering the questions, read the last two exercises again.

Location	Data		Comments
003	240		P Register
200	000	______	Clear output lights
240	200	______	NOOP
241	024		LOAD A (060)
242	060	______	
243	200	______	NOOP
244	344		? ? ? ? ?
245	241	______	
246	000	______	HALT

What is the value of the P Register? ______________________

In what location is the first instruction? ______________________

What is the name of this instruction? ______________________

How many locations does the first instruction occupy? ________

What should the value of P be after this instruction is done? ___

Do a single instruction (see the first page of this Exercise).

Read P (the contents of location 003). ______________________

What is the name of the next instruction? ______________________

How many locations does it require? ______________________

What should P be after this instruction is done? ______________

Do a single instruction. The P Register should be equal to 243. Check it.

Do a single instruction. Now P should be equal to 244. Check it. (If it isn't, keep doing single instructions until P = 244.)

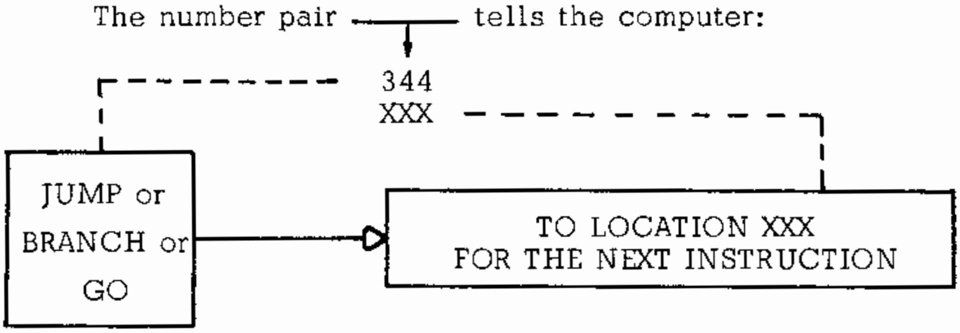
Do a single instruction.

What values does the P Register have now? __________________

Were you expecting something like this? ____________________

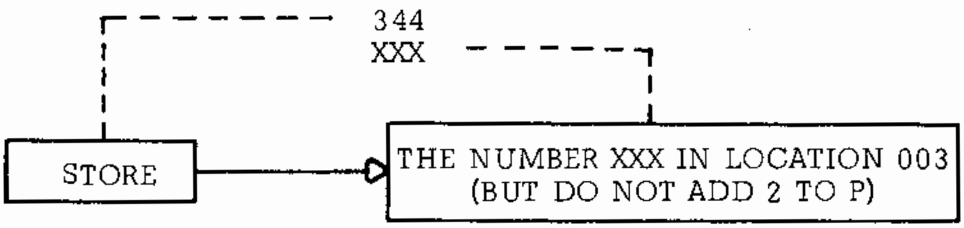
From where, do you think, this value of P came? (Hint, look in the data above.) ____

Do you think the P Register can be changed in ways that we haven't talked about? ____



The normal next sequence for locating the instructions is changed by this instruction. The location XXX can be any memory location where it is desired to have the computer go for its next instruction. Not only does the computer go to location XXX for its next instruction, but the P Register is also changed to this value.

We could look at this Jump instruction in this way:



When put this way, it emphasizes that P is changed. These two descriptions are the same though we will use the language of the first description.

In our shorthand we could say:

P : XXX

Operation Code is 344 JUMP

When given in this way, the description seems so short that it is hard to believe that it could be a very useful instruction. It is a very useful instruction.

Do you see why an instruction of this type could be called Jump or Branch or Go To? We prefer the name "Jump".

Let's have some more fun with the computer. Load this program.

Location	Data	Comments
000	000	A
003	150	P
150	004	
151	154	
152	344	
153	150	
154	001	

Fill out line 1 below by reading the contents of the locations. Have the computer do a single instruction. Keep repeating this pattern until all the lines are filled in.

1. A = ____________ P = ____________
2. A = ____________ P = ____________
3. A = ____________ P = ____________
4. A = ____________ P = ____________
5. A = ____________ P = ____________
6. A = ____________ P = ____________
7. A = ____________ P = ____________
8. A = ____________ P = ____________

Let's stop here, it could go on forever. From the values for A and P, can you tell what the program is doing? Write a short statement telling what the program does.

____

____

____

Here's another complete program to try. (The program is called Chase Your Tail.)

Location	Data	Comments
003	100	P
100	344	
101	100	

Do single instructions and look at the P values. Push Start and let the computer run. Push Stop. What does this program do?

____

____

____

____

Does it seem very useful? ____________________

Exercise 10

In the last exercise we were studying the P Register. The contents of the P Register are the address of the next instruction. We saw P advance in a normal next sequence. Also we saw how we could cause the computer to break the sequence of taking instructions from an increasing series of addresses. The Jump instruction could be used to make the computer go to any location for its next instruction.

In this exercise we will continue working with Jump instructions. Let's start right off by loading a program.

Location	Data	Comments
000	000	A Register
003	100	P Register
100	024	LOAD A
101	360	
102	044	? ? ?
103	140	
104	000	HALT
105	344	JUMP TO 100
106	100	
140	000	HALT
141	344	JUMP TO 100
142	100	
200	000	
360	001	

Push Start. The computer should stop. (Run light out).

Read the contents of location 003.________________________

Push Start again. The computer should stop again.

Read the contents of location 003.________________________

Change the number in location 360 to 000. Push Start again. The computer should stop.

Read the contents of location 003.________________________

Push Start again. The computer should stop.

Read the contents of location 003.________________________

You should have different answers for the P Register values. The first two times they should be the same. The last two times they should be the same but different from the first two times.

Does this have anything to do with the number in 360?______

Change the number in 360 back to 001. Push Start. The computer should stop.

What are the contents of location 003?________________

Is the number in 360 an instruction (Hint: See below)?_____

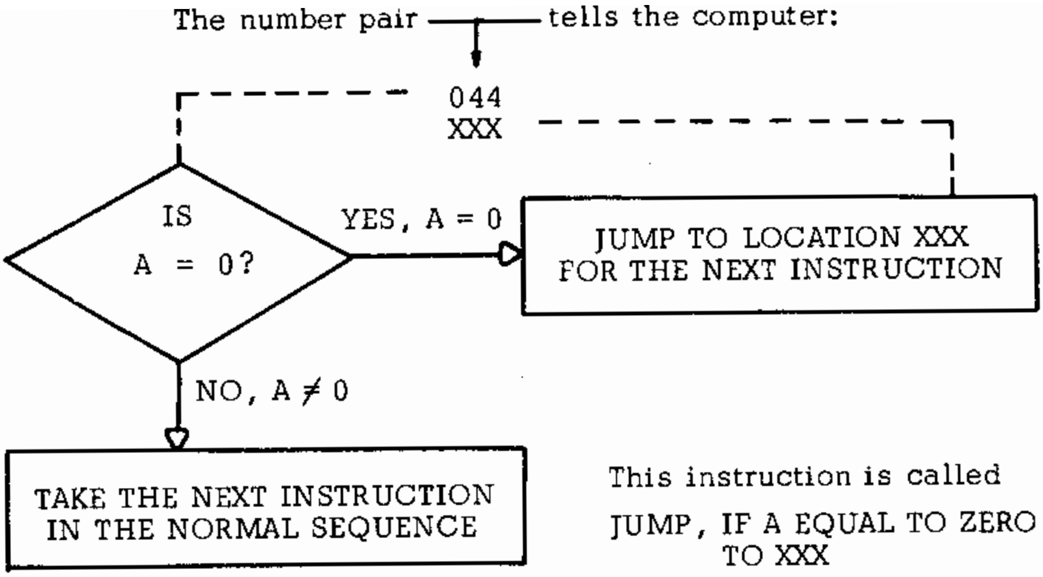
If you don't know the answer to that last question, do single instructions and look at the values of the P Register until you repeat a number. You can write your P Register values here:

_____ _____ _____ _____ _____

Did the P Register ever indicate that the number in location 360 was to be used as an instruction?_________________________

The instruction in locations 102 and 103 sometimes causes a Jump to location 140. Sometimes it doesn't cause a Jump. We see that the number in location 360 may be involved. By changing this number, which isn't an instruction, we can cause the computer to jump or not to jump. Just prior to the Jump instruction in location 102 and 103, the instruction in 100 and 101 load A with the contents of location 360.

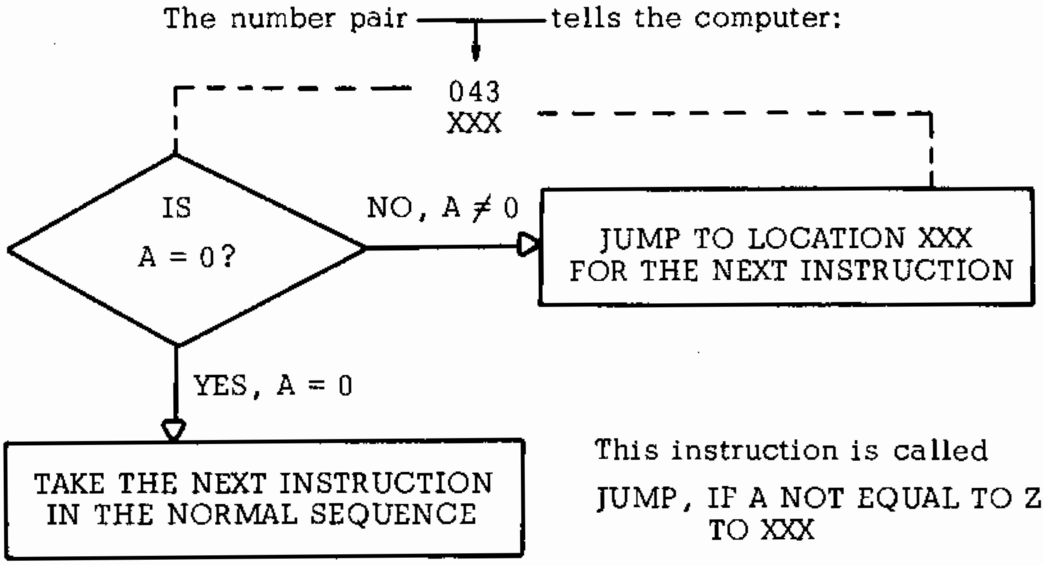
Don't you think the important thing may be what the A Register contains?____



This instruction is called

JUMP, IF A EQUAL TO ZERO,
TO XXX

This instruction has two possible outcomes depending on whether the A Register is equal to zero or not. If the A Register is equal to zero, the computer will jump to location XXX for the next instruction. Location XXX can be any location we desire. If the A Register is not equal to zero, no Jump is made and the following instruction is used as the next instruction.



This instruction is called

JUMP, IF A NOT EQUAL TO ZERO,
TO XXX

Do you see the difference? Compare the words until you find the difference.

Write an explanation for what this instruction does (see the one above).

____

____

____

So far we have learned these instructions.

Symbolic Instruction			Numeric	Shorthand
ADD	A	(XXX)	004 XXX	A : A + (XXX) P : P + 2
SUBTRACT	A	(XXX)	014 XXX	A : A − (XXX) P : P + 2
LOAD	A	(XXX)	024 XXX	A : (XXX) P : P + 2
STORE	A	(XXX)	034 XXX	(XXX) : A P : P + 2
JUMP		XXX	344 XXX	P : XXX

JUMP	A=0	XXX	044
			XXX
				if A = 0, P : XXX
				if A ≠ 0, P : P + 2

JUMP	A≠0	XXX	043
			XXX
				if A ≠ 0, P : XXX
				if A = 0, P : P + 2

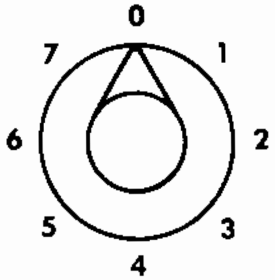
HALT			000	P : P + 1 HALT
NOOP			200	P : P + 1

(XXX = 000, 001,.......,377)

Exercise 11 (Double length)

In this Exercise we'll analyze a problem and write a program to solve it.

The Problem



We want the computer to be a combination lock. We'll use only the 0 to 7 digits and we'll require only a two number sequence to open it.

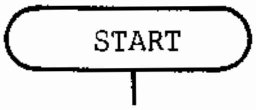
The lock would be a better lock if the sequence were longer, but we want to keep it simple and we want to "crack the safe". It's fun to crack the safe.

If the code were 57, then the entry of a 5 followed immediately by a 7 will open it. (We'll ignore clockwise and counter-clockwise turns). The program should allow the codes to be changed easily.

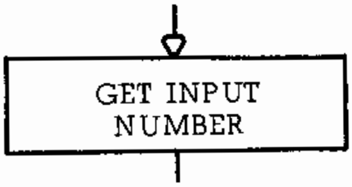
For input, we'll use a single button such as □□ ■□□ □□□ for 5. We could use binary numbers but that would be more buttons to push. When the lock is open let's have the computer turn on all the output lights. We need two variables that describe the first and second codes. Let these two be X1 and X2. We'll store them in the same form as the input.

Flow Charting

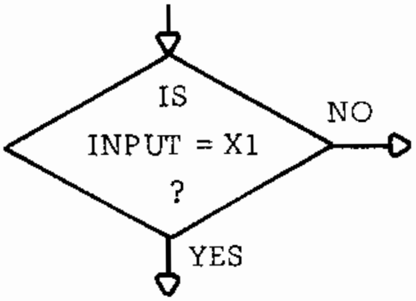
Right now we are not going to describe any rules for flow charting. We won't even define a flow chart. We'll just do some things that come naturally to us. First, we have to start somewhere so let's put that down.



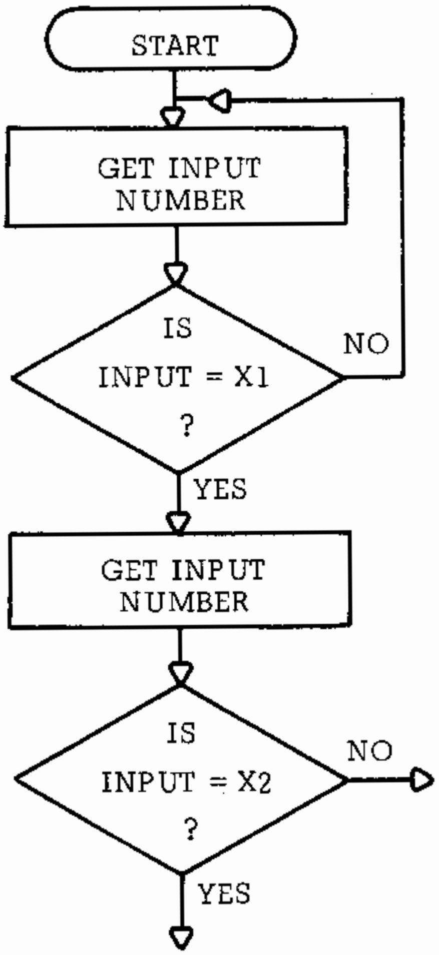
We must get an input number. For the moment let's skip the fol-do-rah about how we get the input number.



When we have the number we want to know whether this is equal to X1. Since this is a question and not a statement of action, let's use a different enclosure.



We've drawn two lines leaving the diamond, one for NO and one for YES. In case the answer is NO we should start over andget another input number. If the input number is equal to X1, we must remember the code is half-broken and we must now compare the input against X2.



Looking at the NO branch of this X2 test, the second input character did not equal X2. The code isn't broken. But this input character might be equal to X1 so we should test for that.

Why? Consider this:

45 Combination that opens the lock

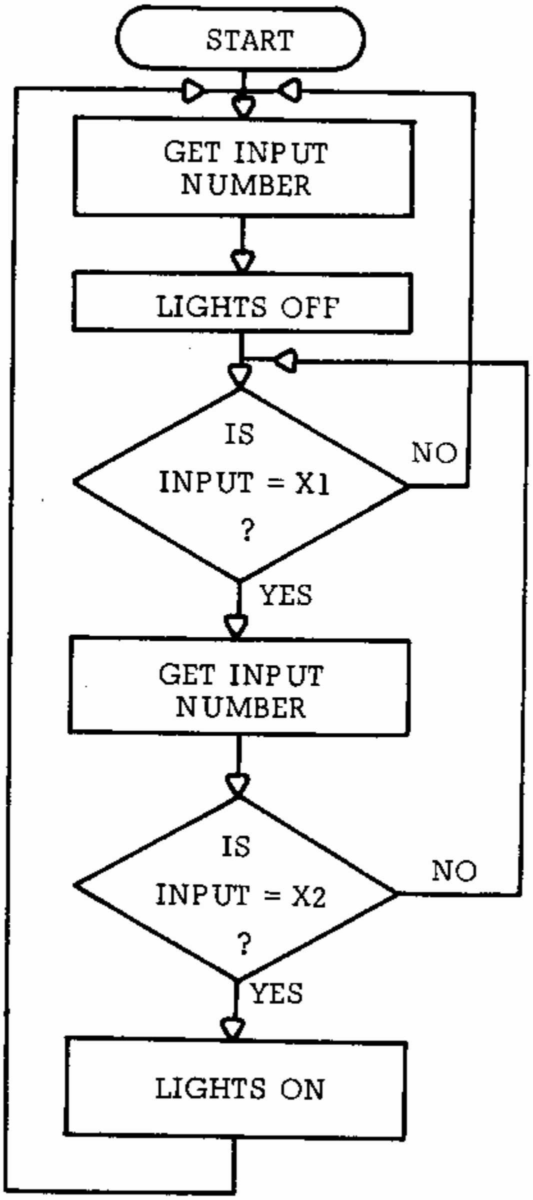
445 Portion of input sequence

The 4 here passes the X1 test

This 4 fails the X2 test,

but this second 4 plus this 5 should open the lock.

Looking at the YES branch, the code is broken and we should turn the lights on. After turning the lights on, let's go back and be ready to play again.



We see that we should make provision for turning the lights off. So we added one more box to the flow chart to remind us to do that.

Input and Output

Let's take a little time to talk about input and output in our computer. Whatever number we enter with the input switches, including Clear, appears in location 377. This location is used to assemble or to gather together the bits of a number which we are entering. This happens when the computer is running or when the computer is stopped.

When the program is ready for another input, we will have it halt. The person enters his input and starts the computer again. With the computer running again, the program can take the contents of location 377 as the input.

Before halting for the input, we will have the program store the number 000 in location 377. This is the same as if the person pushes Clear. We do this to save a person from having to push Clear. Our input sequence will be

Store the number 000 in location 377

Halt

When the program starts again

The input number is in location 377

For output, the program will store a number in location 200. When the computer is in Run, the contents of location 200 control the display lights. When the computer stops, the contents of location 200 continue to be displayed until the operator makes another choice for the display.

The Program

Let's now use the flow chart and the comments we have made about input and output to write a sequence of instructions for the computer. Let's begin with the box on the flow chart that says START. The first thing to do is to get an input number. We'll have the program clear location 377 and halt. We'll put the first instruction that we write in location 100.

```
Location    Data   Comments

     100    024    Load A with the number 000
     101    376            Location 376 is to hold 000
     102    034    Store A in location 377
     103    377
     104    000    Halt
```

When the computer starts again, the input will be in location 377. Before we use this input we'll clear the output in location 200. We see from the instructions above, that A is 000. So

```
     105    034    Store A in location 200
     106    200
```

To determine whether the input is equal to the X1 code, we can use these instructions

```
     107    024    Load A with the input
     110    377            Input is in location 377
     111    014    Subtract from A the X1 code
     112    375            Have X1 in location 375
```

Don't let the word "code" confuse you. It's just a number.

The Jump instruction which jumps on A not equal to zero will complete the action required by the first diamond in the flow chart

```
     113    043    Jump, A ≠ 0, to 100
     114    100
```

Taking the YES path from the first diamond, we have to get another input number. We'll use the same pattern of three instructions as we used before

```
Location   Data   Comments

  115      024    Load A with the number 000
  116      376              We had 000 in location 376
  117      034    Store A in location 377
  120      377
  121      000    Halt
```

When the program starts again we will determine whether the input is equal to the X2 code.

```
  122      024    Load A with the input
  123      377              Input is in location 377
  124      014    Subtract from A the X2 code
  125      374              Have X2 in location 374
```

The NO branch from this second diamond is to be taken if A is not zero

```
  126      043    Jump, A ≠ 0, to location 107
  127      107
```

We determined that 107 was the address to use by comparison to the flow chart.

If A was equal to zero after the last subtraction, the lock has been opened. To turn the lights on

```
  130      024    Load A with the number 377
  131      373              Put 377 in location 373
  132      034    Store A in location 200
  133      200
```

Now the program should go back and start over again

```
  134      344    Jump to location 100
  135      100
```

We have omitted one other location that is important. Do you know which one? Location 003 which holds P should have the value 100 since the first instruction that the computer was to execute was in that location.

Let's put all of the instructions, data and constants together. We have

Location	Data	
003	100	
100	024	
101	376	
102	034	
103	377	
104	000	
105	034	
106	200	
107	024	
110	377	
111	014	
112	375	
113	043	
114	100	X1 and X2 are chosen from these codes
115	024	
116	376	
117	034	
120	377	
121	000	
122	024	
123	377	
124	014	
125	374	
126	043	
127	107	
130	024	
131	373	
132	034	
133	200	
134	344	
135	100	
373	377	
374	X2	Determined by the operator
375	X1	
376	000	

For Input Digit	X1 and X2 Code
0	00 000 001
1	00 000 010
2	00 000 100
3	00 001 000
4	00 010 000
5	00 100 000
6	01 000 000
7	10 000 000

First, try the program with codes of your choosing. See if the program works. Then see if you can get a friend to load X1 and X2 codes. Can you open the lock then?

Exercise 12

The final program that we developed for the lock program is hard to "read" and understand. It is very desirable that we be able to understand our own programs. When we first write a program, it probably has some errors in it. Perhaps we used an Add instruction when we should have used a Load instruction. Or the program jumps to the wrong instruction. Before we can make changes to a program (and we will do a lot of changing), we must understand what we have done already. A list of locations and their data content is a poor way of expressing our thinking. It is true that this is exactly what the computer requires. It is the only thing that it understands. But *we* need something more than three numbers. While "something more" sounds like extra work, it will save us effort and time. Let's see what this "something more" is. Here is an example of it.

```
Symbolic   Contents of
 Address   Location

       A   000
       P   BEGIN
     ONE   001
  OUTPUT   000

   BEGIN   ADD A ONE

           STORE A OUTPUT

           JUMP BEGIN
```

Can you read this as a program?

Scanning this quickly, we can see three instructions at the bottom. But instead of using numerical addresses, they use the words ONE, OUTPUT, and BEGIN. All of these words appear also in the Symbolic Address column. Opposite ONE we see the contents of the location are 001.

Thus we begin to form the idea that

Symbolic Address	Contents
ONE	001
	ADD A ONE

may in some way mean an example like this

Location	Data
135	001
	ADD A (135)

where the 135 could just as well be any other numerical address. Thus the use of ONE would be similar to the use of x in algebra.

ONE is a symbolic address. It stands for or represents an address which is not yet known. When we use symbolic addresses it is understood that we don't use parentheses that normally would be required.

When we use the label ONE with the constant 001, it is obvious that we were thinking about the contents. It is a symbolic address which is descriptive of the contents of the location.

What does the program do? The first instruction adds one to the A Register. The second instruction stores A in OUTPUT which will control the display lights. The third instruction tells the program to start over. The program cycles continuously adding one to the A Register and putting these values out to the display lights. The display lights will show the binary counting process.

Notice that we can write the program, read it and talk about it even though the instructions and the constant ONE have not been assigned memory locations. The way the computer was designed determines the locations for A, P, and OUTPUT, but it is not important at this stage in the development of the program to remember just what these are. We do have to understand the functions of the A Register, P Register, and the Output location.

When we write programs, we will first write a symbolic program. We won't even talk about memory locations until we think that we have a correct program. Only then will we assign locations and specify the complete data content of these locations. This final step is simple and easy to do. But we will do it last.

How do we translate the symbolic program to numeric codes? Look at the example below.

Loc	Data	Symbolic Address	Contents
000		A	000
003		P	BEGIN
200		OUTPUT	000
		BEGIN	ADD A ONE
			STORE A OUTPUT
			JUMP BEGIN
		ONE	001

We started by putting A, P, and OUTPUT on the top lines. The locations assigned to these are known and can't be changed. So we put them down and got them out of the way. Below these we put the instructions. We spaced the instructions so that there would be as many memory locations as each instruction required. Some will require two and some take only one location (here all instructions require two locations). After the instructions we put the contant 001. We could have used a different order. We could have put the constant before the instructions. Be we couldn't put the constant ONE between the ADD and STORE instructions.

Next we complete the assignment of locations. Except for the reserved locations 000, 001, 002, 003, and 200, 201, 202, 203 and 377, we can make most any choice for the first instruction. Location 004 is a very good choice. With this determined, the locations of the rest of our instructions are also determined. We'll assign ONE beginning at 204. Then if we add more instructions to the program, we won't have to move ONE.

Loc	Data	Symbolic Address	Contents
000		A	000
003		P	BEGIN
200		OUTPUT	000
004		BEGIN	ADD A ONE
005			
006			STORE A OUTPUT
007			
010			JUMP BEGIN
011			
204		ONE	001

With the locations now assigned, we can fill in the Data column. Some of the entries in the Data column don't depend upon the location assignment and could have been filled in earlier.

For location 000, its contents are defined to be 000.

For location 003, we defined (by BEGIN) that it should be the address of the location which is labeled BEGIN. We see that this is 004.

For location 200, its contents are defined to be 000.

For location 004, we use the operation code for ADD (004). (This is just a coincidence that location 004 contains 004.)

For location 005, the instruction refers to a location named ONE. The address of ONE is 204.

For location 006, the operation code is 034.

For location 007, the address assigned to OUTPUT is 200.

For location 010, the operation code is 344.

For location 011, the address for BEGIN is 004.

For location 204, it is defined to be 001.

Hence we have,

Loc	Data	Symbolic Address	Contents
000	000	A	000
003	004	P	BEGIN
200	000	OUTPUT	000
004	004	BEGIN	ADD A ONE
005	204		
006	034		STORE A OUTPUT
007	200		
010	344		JUMP BEGIN
011	004		
204	001	ONE	001

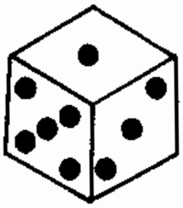
Here is a symbolic program for you to convert to numbers for the computer to use.

Loc	Data	Symbolic Address	Contents
		A	(Doesn't Matter)
		P	START
		OUTPUT	000
		INPUT	(Doesn't Matter)
		START	LOAD A ZERO
			STORE A INPUT
			HALT
			LOAD A INPUT
			STORE A NUM
			ADD A NUM
			STORE A OUTPUT
			JUMP START
		ZERO	000
		NUM	(Doesn't Matter)

After you convert the program, try it in the computer. Push Start. The computer will halt. Enter some number with the switches. Push Start. The computer should halt again. The number displayed should be twice your entry. You can repeat this.

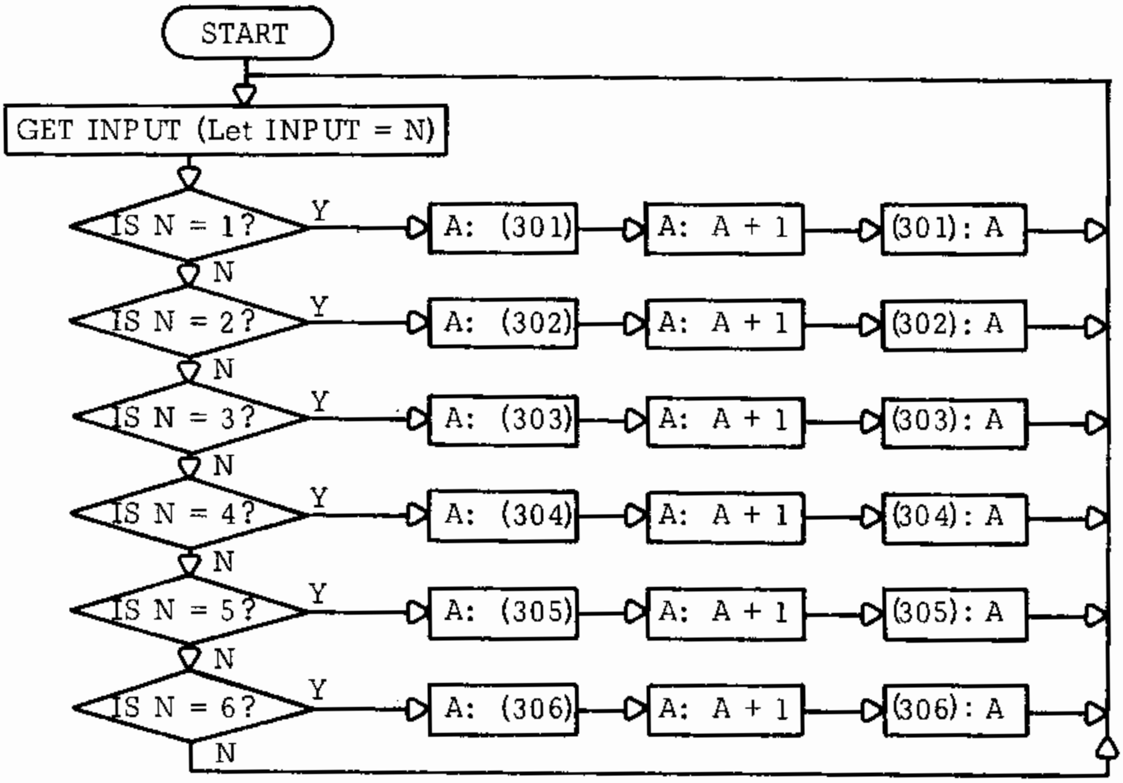
Exercise 13

Jack had a die (one die, two dice) which he thought might not be fair. It seemed to him that some numbers came up more often than they should. We agreed to use the computer to help Jack test his die. Jack would roll it and call the number: 1, 2, 3, 4, 5, or 6. We would keep a tally in the computer for each of these six numbers. For example, when Jack rolled the number 3 we would increase the count for 3 by one. When Jack had rolled the die a large number of times, we would compare the counts for the six numbers.



We'll use six locations in the memory to hold the counts, one for each of the six possible numbers that Jack might roll. To start, each of these locations will contain 000. Let's choose these locations: Location 301 for the tally on number 1, 302 for the tally on number 2, 303 for number 3, 304 for number 4, 305 for number 5, and 306 for number 6.

One very straight forward solution is given in the flowchart below:



The instructions to get the input and to test whether it is equal to 1, 2, 3, 4, 5, or 6 are simple. And the steps to increase the tally count for one of the numbers are easy enough. Let's see what it would take to increase the tally for 3 by one.

			LOAD A (303)
			ADD A ONE
			STORE A (303)
			JUMP START
		ONE	001

To increase the tally for 4 by one, the instructions would be:

			LOAD A (304)
			ADD A ONE
			STORE A (304)
			JUMP START

There would be a similar set of instructions for each of the six counts.

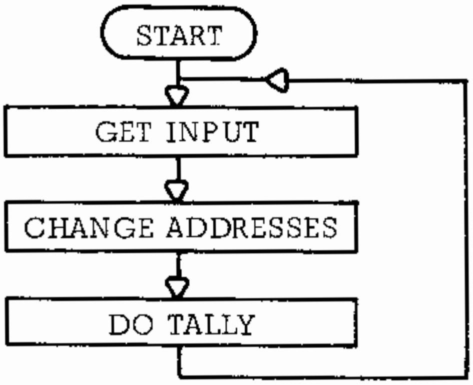
Is there a way we can write the program so that fewer instructions are required? There is. We can have the computer change its own instructions.

Note that our typical tally sequence is

	024		LOAD A (30X)
	30X	ADDR 1	
	004		ADD A ONE
	034		STORE A (30X)
	30X	ADDR 2	

where X is 1, 2, 3, 4, 5, or 6 depending on the number that Jack rolled.

We will have the program change the value of X in the two locations, ADDR 1 and ADDR 2. Then these three instructions can be used to tally for all of the numbers. Very broadly, the program will do these things:



To do the tally, we will use the instructions in the typical tally sequence above. To change the addresses, we can use these instructions:

			LOAD A K300
			ADD A (377)
			STORE A ADDR 1
			STORE A ADDR 2
		K300	300

The first of the instructions loads A with the number 300 (stored in symbolic location K300). The input number in location 377 is added to this. The A Register then contains 301, 302, 303, 304, 305, or 306. This number is stored in symbolic locations ADDR 1 and ADDR 2. We need only one tally sequence.

Our complete program, in symbolic form, is given on the next page.

ADD A	004
SUB A	014
LOAD A	024
STORE A	034
HALT	000
JUMP	344

Loc	Data	Symbolic Address	Contents	Comments
000		A	- - -	
003		P	START	
377		INPUT	- - -	
200		OUTPUT	000	
		START	LOAD A ZERO	Get input
			STORE A INPUT	
			HALT	
			LOAD A K300	Form address
			ADD A INPUT	
			STORE A ADDR 1	
			STORE A ADDR 2	
			LOAD A (30X)	Do tally
		ADDR 1		
			ADD A ONE	
			STORE A (30X)	
		ADDR 2		
			STORE A (200)	To display tally
			JUMP START	
		ZERO	000	
		K300	300	
		ONE	001	
301			000	Tally for 1
302			000	Tally for 2
303			000	Tally for 3
304			000	Tally for 4
305			000	Tally for 5
306			000	Tally for 6

EX 13-4

We added one instruction to the symbolic program which we didn't discuss. We stored the new tally count in the output register.

The symbolic program has a very serious flaw. It does work correctly, but it will accept any number (from 000 to 377) as input. To show a problem which can arise, suppose you put your first instruction in location 340. Then suppose you use the number 040 as input. The program will change the instruction in 340 instead of one of the valid tallies. No matter where you put the instructions, a similar situation can occur. The program can destroy itself. Good programs are not sensitive (can't be damaged) by bad input data.

Here is an interesting experiment. Complete the symbolic program and load it. Take a "random" page in the telephone directory. Use only the last digit of each telephone number. Use only the 1, 2, 3, 4, 5, and 6 digits and skip the numbers ending in a 0, 7, 8, or 9 digit. Use the computer to tally a couple of hundred digits. Do you think they occur with equal probability? Here are the results of one such test:

number of 1's	-	27	(decimal)
number of 2's	-	22	
number of 3's	-	23	
number of 4's	-	20	
number of 5's	-	18	
number of 6's	-	13	
Total		123	

In this example we should conclude that these digits are not distributed evenly. The probability that a telephone number ends in a 1 is much higher than the probability that it ends in a 6.

Exercise 14

In the last Exercise we stored our variable data, the tallies, in a very orderly way. We took advantage of this to have the program change the addresses it used. Again and again we will see the advantage of storing the data in a systematic way. We'll see it again in this Exercise. We'll also see a new principle.

We want to look at the problem of adding 50 small numbers together. These might be the number of runs that a baseball team scored in 50 games. Let's assume we have these 50 numbers in consecutive locations in the memory starting at location 204 and ending with 265 (both octal).

We could use a set of instructions like the following to add the 50 numbers together:

Loc	Data	Symbolic Address	Contents
			LOAD A ZERO
			ADD A (204)
			ADD A (205)
and so on until			
			ADD A (265)

How many instructions would be required? ________________

How may locations in memory would be required to hold these instructions? ________________________________

Let's look at another way. After the computer has done the second instruction above, do you think it could add 1 to the address in this instruction? ________________________________

It could, and the instruction would now be

			ADD A (205)

If the computer now did this instruction, it would add in the second of the fifty numbers. Let's examine a set of instructions which would do this.

100	024	BEGIN	LOAD A ZERO
101	302		
102	004	LOOP	ADD A (204)
103	204	ADDR	
104	034		STORE A SAVESUM
105	300		
106	024		LOAD A ADDR
107	103		
110	004		ADD A ONE
111	301		
112	034		STORE A ADDR
113	103		
114	024		LOAD A SAVESUM
115	300		
116	344		JUMP LOOP
117	102		
300	---	SAVESUM	---
301	001	ONE	001
302	000	ZERO	000

Let's examine these in some detail. Start with the instruction in location 100 and 101. Answer the statements below as true or false.

After the 100/101 instruction, A is 000. . . . . T F

After the 102/103 instruction, A is the first

of the 50 numbers. . . . . T F

Storing A in SAVESUM saves the number in the A Register. We want to use the A Register for another purpose now.

After the 106/107 instruction, A contains 204. . . . T F

After the 110/111 instruction, A contains 205. . . . T F

After the 112/113 instruction, ADDR contains 205. . T F

After the 114/115 instruction, A is restored
to its value after the 102/103 instruction. . . . . T F

The next instruction after the one in 116/117
is the instruction in 102/103. . . . . T F

After doing the instruction in 102/103, A will
contain the sum of the first and second numbers. . . . . T F

Eventually the computer will stop. . . . . T F

These statements are all true except the last one. Unfortunately, it is false. These program steps have no end. The fifty numbers will be added together but then the program will continue to add everything else in the memory to this sum. And it will stay in this cycle forever.

Let's look at how we can stop the program from looping back after it has added the fifty numbers together. There is an easy answer. If the number we store in ADDR is 266 (by instruction 112/113), then we have added the fifty numbers together, no more, no less. This test can be made by inserting two new instructions after the ones in 112/113.

			SUB A K266
			JUMP A=0 END
		K266	266

where END is the beginning of the instructions to finish the job. They might be

		END	LOAD A SAVESUM
			STORE A OUTPUT
			HALT

which picks up the answer in SAVESUM and puts it in the display lights. The program then stops.

If we wanted to re-use the program, we would have to set the P Register back to BEGIN and we would have to restore the number in ADDR to 204. We could have the program do these things.

We include these features in the symbolic program on page 14-6.

In this program how many locations are required for instructions, constants, and variables (but not for the 50 numbers)? ________________

Is this a saving, compared to our first method? ____________

By the time we completed the program there were 51 numbers to add together. What one location (other than the location for the 51st number) must be changed in the program to do this? __________________

What should the new value be? ________________________

When you try the program in the computer, here are 50 numbers to add together

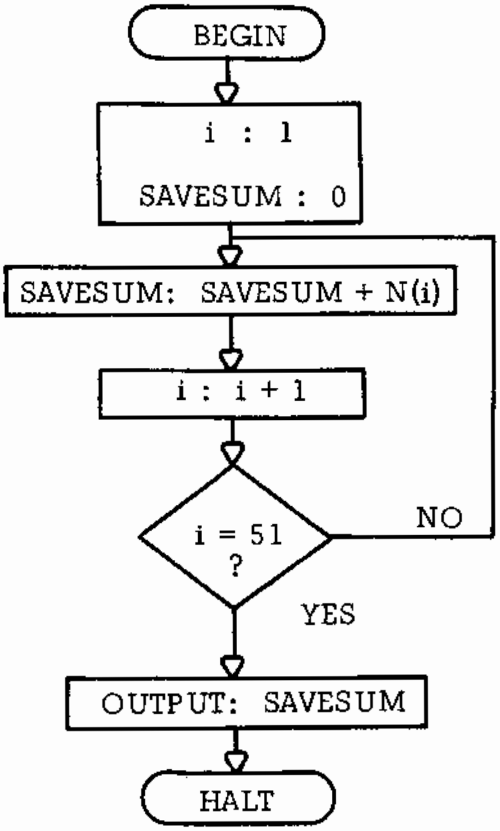
00000 11111 22222 33333 44444 55555 66666 77777 88888 99999

The correct answer is 225 (= 341 octal). Now change the five 0's to 9's. The correct answer is now 270 which is equal to octal 416.

What answer does the computer give you? ________________

Can you show the octal number 416 at one time in the lights? ____

Below we give a flowchart for our program. The terminology is a little different. N(i) is the ith number. We start by having i equal to 1 and we add the first number. We increase i by 1 and add the second number. When the value of i is 51, we stop looping.



ADD A	004
SUB A	014
LOAD A	024
STORE A	034
JUMP A=0	044
JUMP	344
HALT	000

Loc	Data	Symbolic Address	Contents	Comments
		A		
		P	BEGIN	
		OUTPUT		
		BEGIN	LOAD A K204	Initial conditions
			STORE A ADDR	
			LOAD A ZERO	
		LOOP	ADD A (204)	Basic add
		ADDR		
			STORE A SAVESUM	Save A
			LOAD A ADDR	Form next address
			ADD A ONE	
			STORE A ADDR	
			SUB A K266	Last Address?
			JUMP A=0 END	Yes, if jump
			LOAD A SAVESUM	Restore A
			JUMP LOOP	Loop back
		END	LOAD A SAVESUM	Display Sum
			STORE A OUTPUT	
			HALT	
			JUMP BEGIN	To start over
		K204	204	
		ZERO	000	
		SAVESUM	---	
		ONE	001	
		K266	266	

Exercise 15

Have you been wondering about negative numbers? Perhaps you have been hoping that a device as real as a computer didn't use negative numbers. Sorry about that, but negative numbers are just as real as positive numbers and the computer can use both kinds.

Let's do a little exploring with the computer. Load this program.

000	---	A	---
003	004	P	START
377	---	INPUT	---
200	000	OUTPUT	000
004	024	START	LOAD A ZERO
005	017		
006	034		STORE A INPUT
007	377		
010	000		HALT
011	014		SUB A INPUT
012	377		
013	034		STORE A OUTPUT
014	200		
015	344		JUMP START
016	004		
017	000	ZERO	000

This program loads A with 000 and then subtracts the number in the input (location 377) from A. The result is displayed in the lights. We'll subtract some small positive numbers from zero and see what answers the computer gives us. To use the program push Start and enter the number when the computer stops. Then push Start and the lights will show the result of subtracting this number from zero. Complete the table on the next page.

From 00 000 000 subtract		and we get	
0	00 000 000	00 000 000	0
+1	00 000 001	11 111 111	-1
+2	00 000 010		-2
+3	00 000 011		-3
+4	00 000 100		-4
+5	00 000 101		-5
+6	00 000 110		-6
+7	00 000 111		-7
+8	00 001 000		-8
+9	00 001 001		-9

The numbers in the right hand column are what the computer suggests for the first few negative numbers. If you have done your work correctly, you should be able to take + and - entry on the same line and add them together and get 00 000 000. For example:

```
+7     00 000 111
-7     11 111 001
     1 00 000 000
```

When we do this, we get a carry out of the left most column. In the computer there is nowhere to put this bit. It is dropped and only the eight bits on the right are retained. Therefore, we see that what the computer generates for the negative numbers meets this test:

$$x + (-x) = 0$$

Can we count with these negative numbers? Counting up is adding one. Here is an example:

```
-4     11 111 100
+1     00 000 001
-3     11 111 101
```

That was the correct answer. Here are two for you to do and check against the table:

```
-8     11 111 000          -3     11 111 101
+1     00 000 001          +1     00 000 001
-7                         -2
```

If you can count, you can add. Adding x + y is nothing more than starting with x and counting up from there y times.

Can we count down with these negative numbers? Counting down is subtracting one. For example:

$$\begin{array}{rr} -4 & 11\ 111\ 100 \\ \underline{-\ (+1)} & -\ \underline{00\ 000\ 001} \\ -5 & 11\ 111\ 011 \end{array}$$

which is equal to minus five. If you can count down, you can subtract.

How can we tell a positive number from a negative number? If we were to extend the table we started earlier, we would see the positive numbers have a 0 in the left most bit and the negative numbers have a 1. Thus the left most bit becomes the sign where

0 is +

1 is -

How can you read a negative number? Here are two methods:

Method 1. Subtract it from 00 000 000 and this gives you the positive magnitude of the number. For example, given 11 101 011

$$\begin{array}{rl} & 00\ 000\ 000 \\ - & \underline{11\ 101\ 011} \\ & 00\ 010\ 101 \quad = \quad 25 \end{array}$$

so the original number was -25 octal or -21 decimal.

Method 2. Interchange 0's and 1's in the negative number, read it as a binary number and then add 1 to the answer. For example, given 11 111 011, read this as 00 000 100, which is 4, and then add 1. Hence the number is -5. This is a handy method for very small numbers.

In this Exercise we have stated that a number whose most significant bit was a 1 was a negative number. For example,

341 (= 11 100 001)

would be -37 octal. In earlier Exercises we have talked about the number 341 as though it were the positive number 341. Which is the correct answer? Both are. The computer works equally well with both interpretations. It is our choice whether we wish to interpret the bit pattern 11 100 001 as +341 or as -37 (both octal).

If the nature of the problem allows it, we may consider the numbers as positive whole numbers. Then

our smallest number is 00 000 000 (= 0)
and our largest number is 11 111 111 (= +255 decimal)

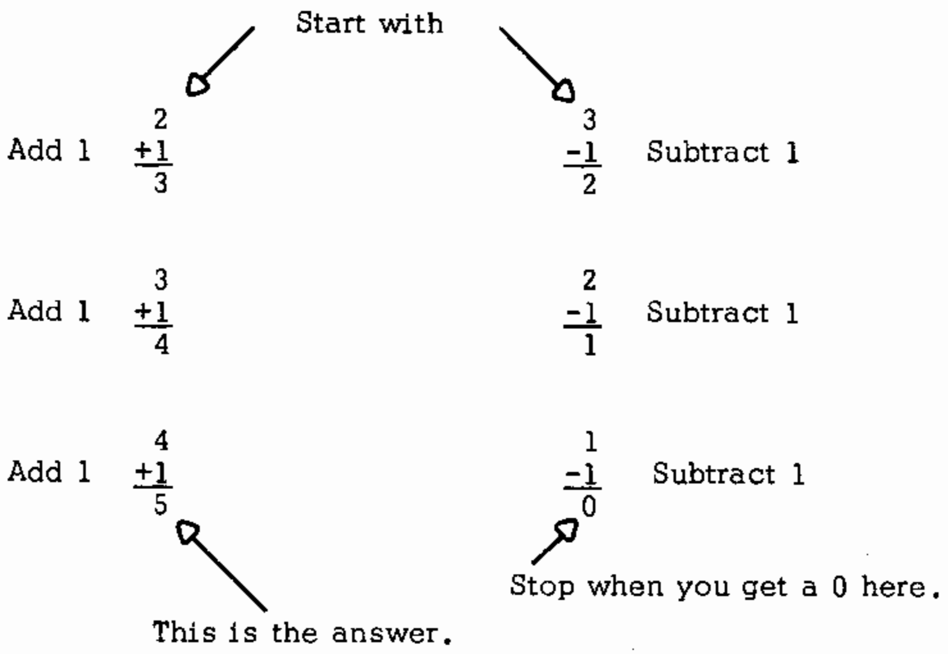
When we are doing address computations (as we were in the last Exercise), this is appropriate because the addresses are always considered to be positive.

If the nature of the problem requires positive and negative numbers, then the number representation that we introduced in this Exercise is appropriate. In this case,

the smallest negative number is	10 000 000	(= -128 decimal)
the largest negative number is	11 111 111	(= - 1)
zero is	00 000 000	(= 0)
the smallest positive non-zero number is	00 000 001	(= + 1)
the largest positive number is	01 111 111	(= +127 decimal)

In both of these interpretations, the decimal point is as the far right. Again, it is our choice where we wish to put the decimal point (or binary point). We could have it in the middle or at the far left or somewhere outside the byte. The computer doesn't care. Of course, when you have the computer add two numbers you must consider the decimal point in both of these numbers to be in the same place.

We now pose a problem for you to program a solution. Write a program to add x and y (both positive numbers) by only adding and subtracting one. We illustrate the technique with a small example of adding 2 + 3.



After you have done this, you can change one instruction and then the program will do x -y. Try this with values of x and y that would generate negative numbers. About a dozen instructions are required. On the next page we give a work form you can use.

ADD A
004

SUB A
014

LOAD A
024

STORE A
034

HALT
000

JUMP
344

JUMP A=0
044

JUMP A≠0
043

Loc	Data	Symbolic Address	Contents	Comments

EX 15-6

Exercise 16

Suppose that we had the problem of writing a program to find the larger of two numbers. It can be done with the instructions that we have learned. You might like to take some time and try this. We give a short analysis below.

If we have two non-zero numbers, U and V, and we want to determine which is the larger, we could subtract the number 1 from each of them. After the subtraction, if the number U has been reduced to 0 but V hasn't, then V is the larger number. If both are zero after the subtraction, then they are equal. If neither one is zero, then we could subtract 1's again and repeat the test. Here are two examples:

```
      1                             2
   ______                        ______

   U     V                       U     V

   3     2                       2     2
  -1    -1                      -1    -1
  --    --                      --    --
   2     1   Subtract again      1     1   Subtract again

   2     1                       1     1
  -1    -1                      -1    -1
  --    --                      --    --
   1     0   U is larger         0     0   Equal
```

So while we can determine which of two numbers is larger without any new instructions, we will see that it becomes much easier with some new Jump instructions that we will learn now.

In addition to jumping on A zero or jumping on A not zero, we also have

Jump if A is negative
Jump if A is zero or positive
Jump if A is positive (and not equal to zero)

These instructions use the sign convention that we learned in the last Exercise. A is negative if the most significant bit (the left hand one) is a 1. A is positive, but possibly zero, if the most significant bit is a 0. A is zero only if all eight bits are zero.

Let's use the following program, to study our five conditional Jump instructions.

Loc	Data	Symbolic Address	Contents	Comments
000	---	A	---	
003	010	P	START	
377	---	INPUT	---	
200	000	OUTPUT	000	
004	024	YES	LOAD A K200	200 = "yes"
005	027			
006	034	NO	STORE A OUTPUT	
007	200			
010	024	START	LOAD A ZERO	Get input
011	025			
012	034		STORE A INPUT	
013	377			
014	000		HALT	
015	024		LOAD A INPUT	Input to A
016	377			
017	043		JUMP (X) YES	Jump instruction being tested
020	004			
021	024		LOAD A ONE	001 = "no"
022	026			
023	344		JUMP UNC NO	
024	006			
025	000	ZERO	000	
026	001	ONE	001	
027	200	K200	200	

We will change the code in location 017 to try the different Jump instructions. The original value is for a Jump If A Not Zero instruction. To use the program, push Start and when the computer stops enter a number in the input. Push Start and the computer will use your input number with the Jump instruction. If it makes a jump, it will display ●o ooo ooo and halt. If it does not make the jump, it will display oo ooo oo● and halt. You can try different input numbers and see whether the computer makes the jump or not.

In the table on the next page, fill in the right column by indicating whether the jump was made or not. (Light 7 on the left is "Yes" and light 0 on the right is "No". Just remember, yes and no, left to right.) For each box in the right column use an example of input data as given in the center column. The mark "-" means that bit doesn't matter. Choose anything you wish.

For each different kind of jump (there are 5), change the code in location 017. The codes are given in the left column of the table. If you are not familiar with the notation, we give a definition here:

$a = b$	a equals b
$a \neq b$	a is not equal to b
$a < b$	a is less than b
$a \geq b$	a is greater than or equal to b
$a > b$	a is greater than b

Instruction	Input (Data to be tested)	Was Jump Made?
JUMP $A \neq 0$ XXX 043 XXX	00 000 000	
	At least one bit is a 1	
JUMP $A = 0$ XXX 044 XXX	At least one bit is a 1	
	00 000 000	
JUMP $A < 0$ XXX 045 XXX	0- --- ---	
	1- --- ---	
JUMP $A \geqslant 0$ XXX 046 XXX	1- --- ---	
	00 000 000	
	0(At least one bit = 1)	
JUMP $A > 0$ XXX 047 XXX	1- --- ---	
	00 000 000	
	0(At least one bit = 1)	

Our three new Jump instructions are based on the assumption that the number being tested, in the A register, uses the negative number convention that we learned in the last Exercise. If we are thinking of the number as positive only, then we must be careful. For example, if we had the address 341 in A we would think of it as positive. Our three new Jump instructions would treat this as a negative number.

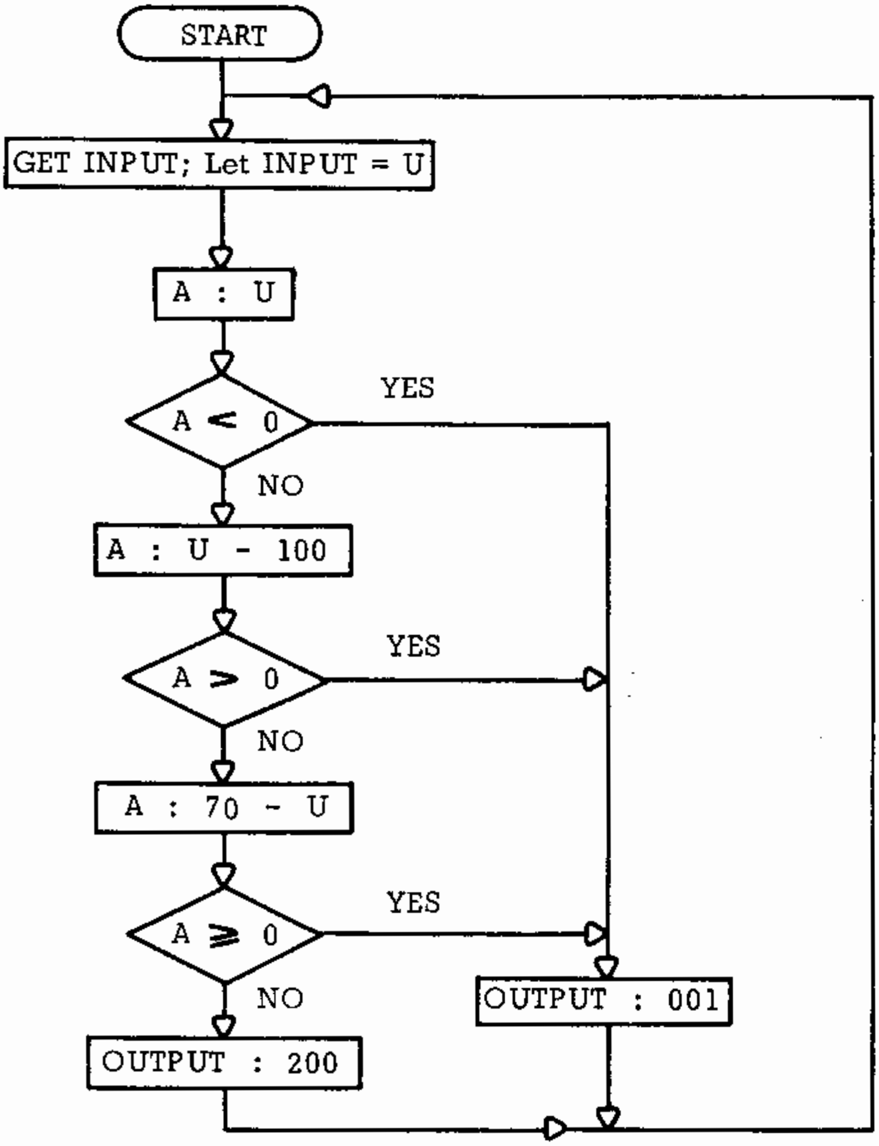
Here is a problem for you to program. Given a number U (let this be your input number), is it larger than 70 but not greater than 100 (all octal numbers)? Stated another way,

is U equal to 71, 72, 73, 74, 75, 76, 77, or 100?

Or using the notation we just learned,

is it true that $71 \leqslant U \leqslant 100$?

We give a flowchart below and a worksheet on the next page. Try U equal to 0, 70, 71, 100, 101, and 270.



ADD A	004
SUB A	014
LOAD A	024
STORE A	034
JUMP $A \neq 0$	043
JUMP $A = 0$	044
JUMP $A < 0$	045
JUMP $A \geq 0$	046
JUMP $A > 0$	047
JUMP	344
HALT	000

Loc	Data	Symbolic Address	Contents	Comments

Exercise 17

In the last Exercise we learned three new instructions. They did not allow us to do anything that we couldn't do before, but they made it easier to do some things. In this Exercise we will learn more features about the computer that make it easier to do some other things. However, the new features don't allow us to do anything that we couldn't do already.

Besides the A register, the computer also has a B register and an X register. The B register is location 001 and the X register has address 002. These registers are called the programming registers. A programmer has full control of these registers. There are instructions to Load, Store, Add, and Subtract (from) the A, B and X registers. The five conditional Jump instructions can also be applied to these registers.

The codes for these new instructions are obtained by changing the most significant octal digit (of the first byte) from 0 to 1 for the B register and from 0 to 2 for the X register. We summarize in the table below.

	A	B	X
Address	000	001	002
ADD	004 XXX	104 XXX	204 XXX
SUB	014 XXX	114 XXX	214 XXX
LOAD	024 XXX	124 XXX	224 XXX
STORE	034 XXX	134 XXX	234 XXX

Below are the test conditions for the Jump instructions

≠0	043 XXX	143 XXX	243 XXX
=0	044 XXX	144 XXX	244 XXX
<0	045 XXX	145 XXX	245 XXX
≥0	046 XXX	146 XXX	246 XXX
>0	047 XXX	147 XXX	247 XXX

As the table shows, the new codes are easily learned. You must only remember the rule about substituting 1 or 2 for the 0.

If we use the instruction ADD X INPUT, we do not change the value of the A or B registers. Only the X register is involved. Likewise, when we use a conditional Jump instruction, the test is made on the specified register only.

Since the A, B and X registers are memory locations and do have addresses, they can be used in another way. We can use them as we would use memory locations to supply or to receive our data. We can add the contents of the A register to the contents of the B register and put the answer in the B register. The symbolic form of this instruction would be ADD B A. Now go back and read the last two paragraphs again except do not do what this sentence says next time.

Below we give a few symbolic instructions that use A, B or X as a symbolic address. Give the octal code for each and an algebraic or shorthand statement for each one to describe what it does.

ADD A B ________________, ______________________________

STORE X A ________________, ______________________________

LOAD A X ________________, ______________________________

ADD A A ________________, ______________________________

SUB B B ________________, ______________________________

What is the value of the extra registers? In several of the problems which we have programmed, we would load A with a variable from the memory, do something with it, then place it back in the memory. Then we would load A with another variable, do something else with it, and put it back in the memory. With more programming registers we can keep our variables in the registers andnot have to transfer them back and forth from the memory. It will save instructions and time.

The problem in Exercise 14 is a good example. Our problem there was to add up 50 numbers. The program alternates between loading A with the sum and loading A with the address that was being changed. (You should review Exercise 14.) Let's look at a program which keeps the address in the X register and keeps the sum in the A register. We give it in symbolic form on the next page.

The instructions labeled 1 initialize the first values of A and X. The instruction labeled 2 sets ADDR, the first time to 204. Instruction number 3 increases X by 1. We see if this new address goes beyond the last number to be added with instruction 4. If it does, the program jumps out with the instruction in 5 to the END routine. Otherwise, we add back in, via instruction 6, the amount we subtracted with instruction 4. With instruction 7 we add the next number in the list to the sum. With 8 we loop back and repeat these steps. When we loop back, notice that X is increased by one.

Loc	Data	Symbolic Address	Contents	Comments
		START	LOAD A ZERO	1
			LOAD X K204	1
		LOOP	STORE X ADDR	2
			ADD X ONE	3
			SUB X K266	4
			JUMP X=0 END	5
			ADD X K266	6
			ADD A (204)	7
		ADDR		Changes
			JUMP LOOP	8
		END	STORE A OUTPUT	
			HALT	

There is one obvious simplification though it is a minor one. The two instructions, 3 and 4, can be replaced by one instruction.

```
ADD  X  ONE
SUB  X  K266
```

can be replaced by

```
SUB  X  K265
```

where K265, K266, and ONE contain the numbers 265, 266, and 001 respectively.

Here is a problem for you to program. In a list of unknown numbers stored in locations 204 to 265, how many times does the number 252 occur? It may be as few as 0 or as many as 50 (decimal). Use all three of the programming registers. It is not difficult and has many similarities to the problem we discussed in this Exercise. A worksheet for your answer is on the next page.

A
0--

B
1--

X
2--

ADD
-04

SUB
-14

LOAD
-24

STORE
-34

JUMP

≠0
-43

=0
-44

<0
-45

≥0
-46

>0
-47

Loc	Data	Symbolic Address	Contents	Comments

Exercise 18

External to the computer, in our world, we use decimal numbers rather than octal or binary numbers. How can we use decimal numbers in the computer? There is a binary number equal to each decimal digit

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

A binary number of four bits larger than 1001 would involve two decimal digits so we won't use 1010, 1011, 1100, 1101, 1110, or 1111. If we have a decimal number we'll substitute the equivalent binary number for each digit. For example, for 1984 we could write

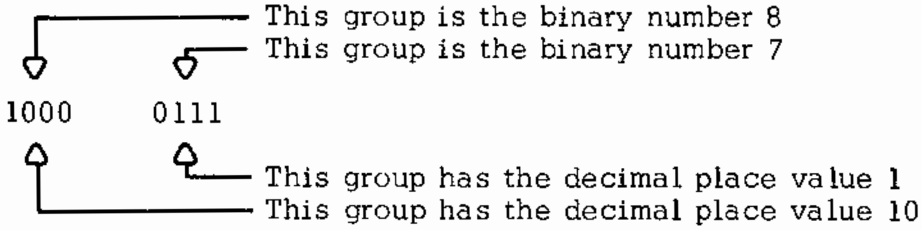
0001 1001 1000 0100

This requires us to memorize the ten codes but we already know eight of them from our binary to octal conversions.

For the time being, let's only talk about the decimal numbers 00 to 99. We can store a number in this range in one byte. The four most significant bits, the left half of the byte, would be the tens digit and the four least significant bits, the right half of the byte would be the units digit. For example,

1000 0111 would be 87 (decimal)

To distinguish between the different number systems we will call this representation binary coded decimal (BCD). In some ways these numbers are like decimals and in other ways they are like binary numbers.


This group is the binary number 8
This group is the binary number 7
1000
0111
This group has the decimal place value 1
This group has the decimal place value 10

Here are some binary coded decimals for you to translate to decimal:

0000 0011 ____________ 1000 0100 ____________

0001 0000 ____________ 0101 0110 ____________

0000 1000 ____________ 0010 0011 ____________

0111 0111 ____________ 0100 0001 ____________

0001 0010 ____________ 0100 1001 ____________

Which of the following binary coded decimals are valid numbers?

0000 1100 ____________

0001 0000 ____________

1010 0000 ____________

1001 0111 ____________

Translate these decimal numbers to binary coded decimal numbers.

50 _________ _________ 46 _________ _________

05 _________ _________ 37 _________ _________

23 _________ _________ 28 _________ _________

78 _________ _________ 19 _________ _________

The computer does its operations in binary. If it were to add two binary coded decimal numbers, the answer may not be correct. Let's look at such a case.

$$\begin{array}{rll} & 0010\ \ 0011 & = 23 \text{ in BCD} \\ + & \underline{0100\ \ 1000} & = 48 \text{ in BCD} \\ & 0110\ \ 1011 & \end{array}$$

which does not equal 71 in BCD. The decimal units digit is not even a valid code.

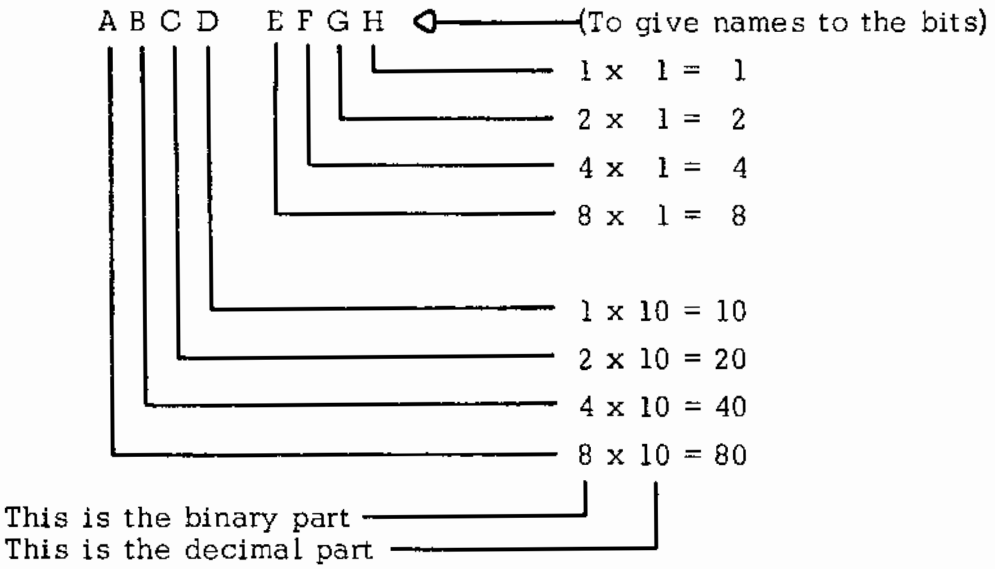
There are two general approaches to this problem.

1. We can use BCD for the input and output but the computer can convert these to and from binary. Internally, the numbers will be "straight" binary.

2. We can keep the BCD representation internally and use special programming techniques. If invalid BCD codes are generated, the program will correct these codes.

At this time we will discuss the conversion of a two digit BCD number to binary. This is one-half of the approach described in 1. above.

Let's look at the place value of each bit in a two digit BCD number:



For the right half byte, the place values are the same as for a binary number. To be valid though, these four bits cannot represent a number larger than 9. For the left half byte, the same binary place values apply but the decimal place value is 10. Hence all of the place values of the left hand byte are multiplied by ten.

Let's look at some simpler BCD cases first.

This BCD number

0000 EFGH

is already converted to binary (or octal) because the tens digit is zero. Therefore

0000 EFGH (in BCD) is equal to 00 00E FGH (in octal)

We have just regrouped the bits, nothing more.

Now take this BCD number

```
                    0001  EFGH
```

The 1 in the left hand byte has a place value of ten. The number ten as a binary number is 1010. Therefore,

```
0001  EFGH  (in BCD)  is equal to  0000  EFGH
                                  +______1010
                                              (in binary)
```

what we have done is to take a 1 from the tens position and added ten to the units position. Let's look at a couple of examples.

```
  0001  0100          = 14 in BCD
- 0001                Subtract 1 in the 10's position
+ _____ 1010          Add 10 in the 1's position
  0000  1110          This is the binary number for 14.
```

Here's another example

```
  0001  1001          = 19 in BCD
- 0001                Subtract 1 in the 10's position
+ _____ 1010          Add 10 in the 1's position
  0001  0011          This is the binary number for 19.
```

Note that there was a carry from bit position E to D

Let's make a general rule for any two digit BCD number, 00 to 99.

1) Subtract 1 in the 10's position

2) Add 1010, in another memory location, each time we do 1)

3) Stop when the BCD number is 0000 EFGH

4) Add the number 0000 EFGH from 3) to the number we were forming in 2) above.

We'll apply this to an example for the BCD number 32.

```
        0011  0010    = BCD 32        00  000  000
I)    - 0001                        + 00  001  010
        0010  0010    = BCD 22        00  001  010    = Binary 10
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        0010  0010    = BCD 22        00  001  010    = Binary 10
II)   - 0001                        +          1  010
        0001  0010    = BCD 12        00  010  100    = Binary 20
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        0001  0010    = BCD 12        00  010  100    = Binary 20
III)  - 0001                        +          1  010
        0000  0010    = BCD  2        00  011  110    = Binary 30
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        0000  0010    = BCD  2
IV)   - 0001
        1111  0010    Whoops. We should not have subtracted again.
                      We can fix it up by adding it back in.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        1111  0010
      + 0001
        0000  0010    = BCD 2 = Binary 2
```

Now we take what is left here and add it to our binary number

```
      00  011  110     Binary 30
    + 00  000  010     Binary  2
      00  100  000     Binary 32
```

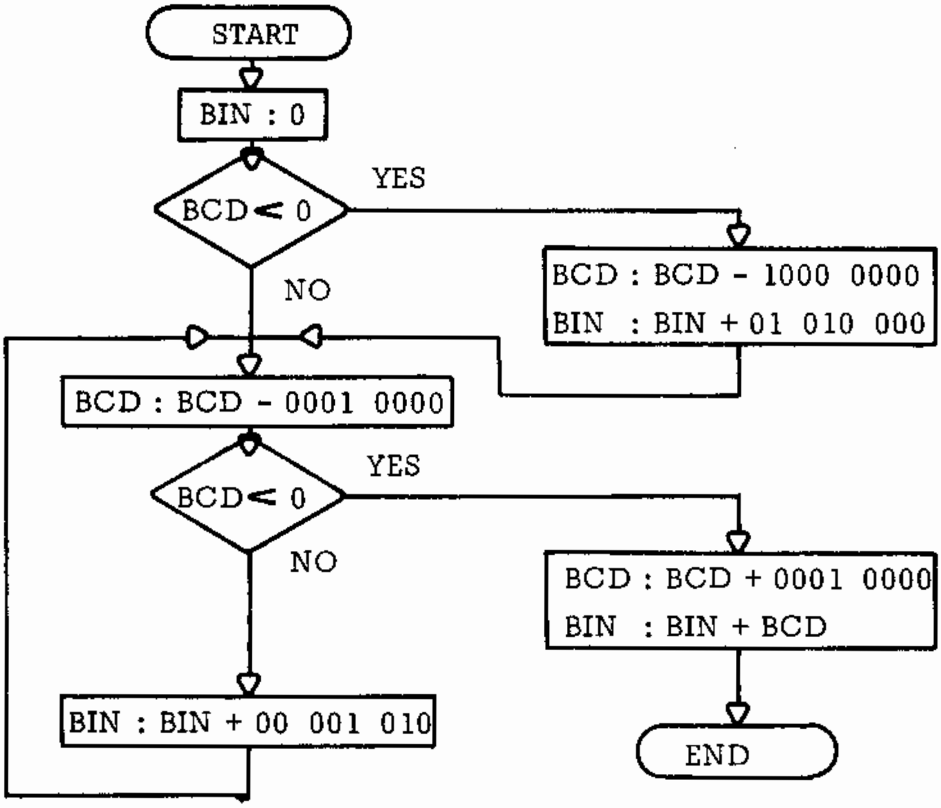
In step IV above, the "over-subtraction" was deliberate. We'll use this method to detect when we have reached 0000 EFGH in the program we'll write shortly. If the answer after subtraction is negative (the most significant bit is a 1), then we have subtracted 0001 0000 one time too many and we should add it back in.

We need first to look at a special case which may arise. Suppose we had the BCD number 95 or in general any number in the 90's. Look at the result of subtracting 0001 0000:

$$\begin{array}{r} 1001\ \ 0101 \\ -\ \underline{0001\qquad\quad} \\ 1000\ \ 0101 \end{array}$$

This answer looks like a negative number and if we used our rules above we would have the wrong answer. Why does this problem come up? It is because the Jump A<0 and the Jump A≥0 and the Jump A>0 instructions are based on a signed representation for positive and negative numbers. Our input of a BCD number is considered to consist of only positive numbers.

It is not difficult to overcome this. Below we give a flowchart to convert a two digit BCD number, 00 to 99, to binary.



ADD A	004
ADD B	104
SUB A	014
SUB B	114
LOAD A	024
LOAD B	124
JUMP	
A<0	045
UNC	344
HALT	000

Loc	Data	Symbolic Address	Contents	Comments
		START	LOAD B ZERO	
			JUMP A<0 FIXUP	
		LOOP	SUB A K020	
			JUMP A<0 FINISH	
			ADD B K012	
			JUMP UNC LOOP	
		FIXUP	SUB A K200	
			ADD B K120	
			JUMP UNC LOOP	
		FINISH	ADD A K020	
			ADD B A	
		END	HALT	
		P	START	

This program starts with a BCD number in A and finishes with the equivalent binary number in B.

Exercise 19

Many times we have seen instructions like these

```
LOAD A ZERO
 ADD B K020
```

Each of these uses three bytes, two for the instruction and one for the data. Instead of having the second byte of the instruction as the address of the data, we can let the second byte be the data. There would be two advantages in doing this - it saves memory space and running time.

This type of instruction, which does not use addressing but contains the data itself, is called constant or immediate. Perhaps "immediate" is the better name because the data is not always constant. It may vary.

We can use the Load, Store, Add, and Subtract instructions in the immediate mode. In the Store instruction the designated register is stored into the second half of the instruction. In the Load, Add, or Subtract instructions the second byte of the instruction is the data or the operand.

Symbolically we write these instructions

```
 ADD A C=000
LOAD A C=020
```

or as

```
  SUB X C=ADDR
STORE A C=
```

or even as

```
LOAD A C=TABLE -1
```

The notation, C= , indicates it is an immediate type of instruction. If a numerical quantity follows, then this is to be the value of the second byte of the instruction. If a symbolic address follows, the numeric address to which the symbolic address is assigned is to be used. Arithmetic combinations of symbolic addresses and constants are allowable. In the Store instruction, no value is given for C since the value to be stored is the value of the register. The notation, C= , is retained to indicate the type of instruction.

The codes for the immediate instructions are easily derived. The right hand octal digit of the first byte becomes a 3. So far, we always have had this digit as a 4.

Let's look at some examples of immediate instructions in the computer. Load this

Loc	Data	Symbolic Address	Contents
003	004	P	
004	023		LOAD A C = 000
005	000		
006	003		ADD A C = 001
007	001		
010	033		STORE A C =
011	000		
012	000		HALT

We'll do single instructions to study this program. The initial value of P is 004. Do a single instruction (see Exercise 9).

The value of P is now ______________________________

The value of A is now ______________________________

From what byte in the memory did this last value come? _____

Do another single instruction.

The value of P is now ______________________________

The value of A is now ______________________________

How did we get this last value? ______________________

Do another single instruction.

The value of P is now ______________________________

The value of A is now ______________________________

The number in location 011 is now ___________________

Prior to this Exercise we would have used other bytes in the memory to store 000, 001, and to save the value of the A register.

How many bytes have we saved? ____________________

In the following program what is displayed in the output lights when the computer halts? ________________________________

Loc	Data	Symbolic Address	Contents
		HERE	LOAD A C=HERE
			ADD A C=006
			STORE A OUTPUT
			HALT

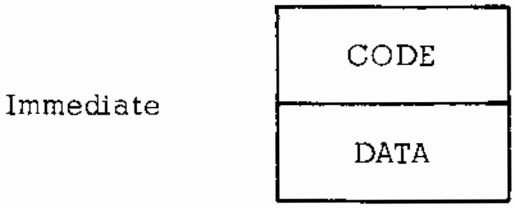
At the end of the last Exercise we had a program to convert a BCD number to binary. That program can use immediate instructions to good advantage. In the worksheet on the next page, write it using as many immediate instructions as you can.

IMMED	--3
MEMORY	--4
ADD	-0-
SUB	-1-
LOAD	-2-
STORE	-3-
A	0--
B	1--
X	2--
≠0	-43
=0	-44
<0	-45
≥0	-46
>0	-47

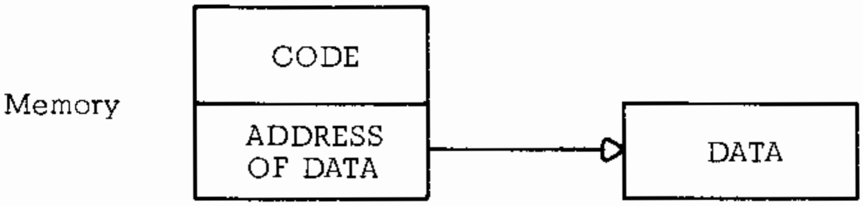
Loc	Data	Symbolic Address	Contents	Comments

Exercise 20

We have learned two forms for our Load, Store, Add, and Subtract instructions. To the first type we will give the name "Memory". To the second type we gave the name "Immediate". Letting a box like this mean memory location, we could represent an immediate type of instruction in this way.

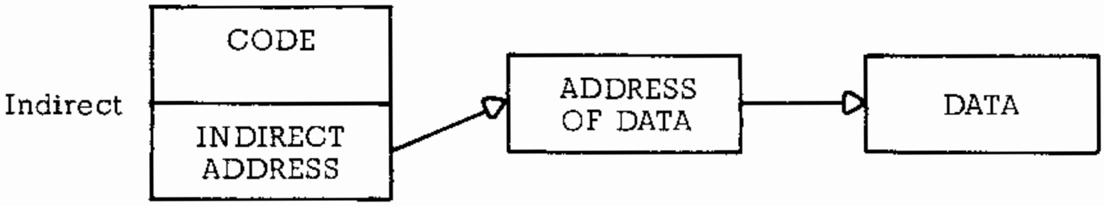


We could represent a memory type of instruction in this way.



In a memory type of instruction, the second byte of the instruction is the address of the location in the memory which contains the data. It "points" to the data.

These diagrams suggest other ways that we might specify the location of the data. One other way, which we can use in the computer, is called "indirect" and is shown below.



The second byte of the instruction contains the address of the location which contains the address of the data. The address in the instruction does not point directly to the data. In an indirect way, it does. Hence this type of data addressing is named "Indirect".

To obtain the code for the indirect mode, the right octal digit of the first byte is changed to a 5. Load, Store, Add, and Subtract can use the indirect address.

An example of the symbolic notation for this class of instruction is

LOAD A (TABLE)

The parentheses around the symbolic address tell us that the indirect address mode is being used. The symbolic address TABLE applies to the location which contains the address of the data. Let's take an example.

Loc	Data	Symbolic Address	Contents
100	035		STORE A (TALLY)
101	177		
102	000		HALT
177	303	TALLY	
301	000	TALLY1	
302	000	TALLY2	
303	000	TALLY3	
304		TALLY4	
305		TALLY5	
306		TALLY6	

The instruction in location 100/101 is a Store indirect. The second byte of this instruction points to location 177. Location 177 in turn points to location 303. The contents of the A register will be stored into location 303.

Why would such a round about method have any value? Like the other addressing methods, there are times when indirect addressing saves memory locations or work.

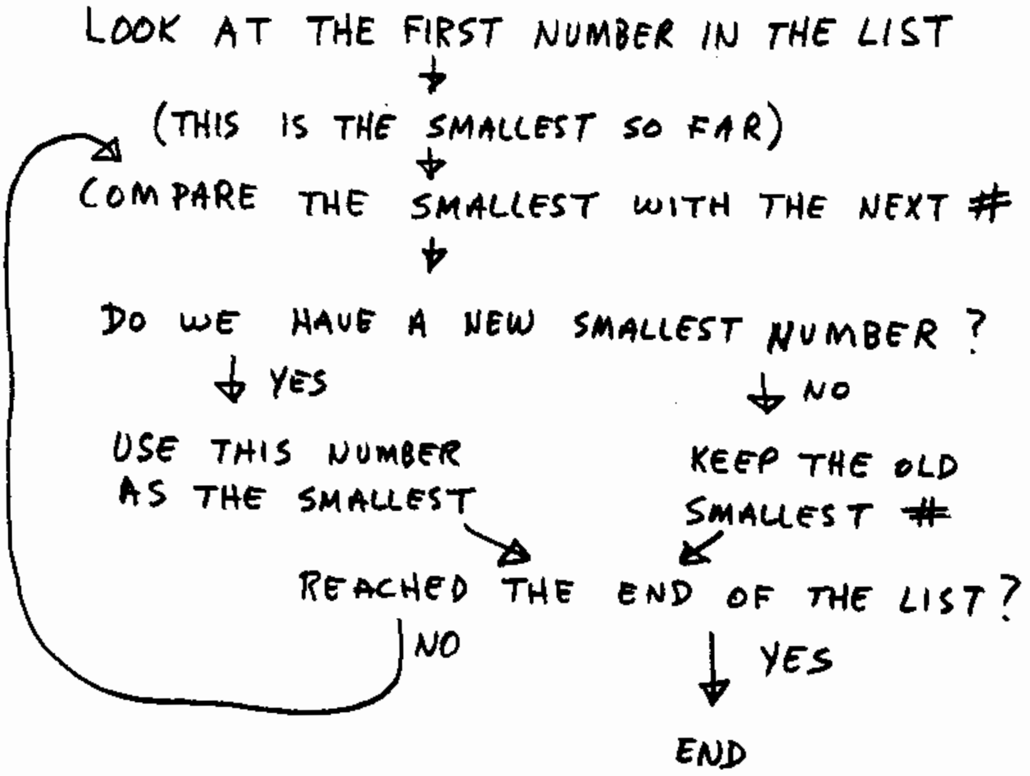
We'll now look at the problem of finding the smallest number in a list of numbers. If the list of numbers were 2, 5, 7, 3, 3, 2, 6, 9, 4, then the smallest number is 2. The fact that it occurs twice has no significance.

Our numbers will be octal in the range 0 to 177. They will be stored in a table or a list which starts with the symbolic address TABLE and ends with the symbolic address LAST. Thus, if we had three entries, we might have

```
TABLE  002
       003
LAST   001
```

Of course the table may be much longer than this. We'll write the program, symbolically at least, without knowing exactly how long the table is or where it is located in memory. When we do know we'll substitute the actual values for TABLE and LAST.

In our analysis we will show how one person approached it including several of his omissions and errors. We wish to exphasize that very few people can write a perfect program from scratch. Our programmer first wrote a very informal and general flowchart.


LOOK AT THE FIRST NUMBER IN THE LIST
(THIS IS THE SMALLEST SO FAR)
COMPARE THE SMALLEST WITH THE NEXT #
DO WE HAVE A NEW SMALLEST NUMBER ?
YES
NO
USE THIS NUMBER AS THE SMALLEST
KEEP THE OLD SMALLEST #
REACHED THE END OF THE LIST?
NO
YES
END

To obtain the code for the indirect mode, the right octal digit of the first byte is changed to a 5. Load, Store, Add, and Subtract can use the indirect address.

An example of the symbolic notation for this class of instruction is

LOAD A (TABLE)

The parentheses around the symbolic address tell us that the indirect address mode is being used. The symbolic address TABLE applies to the location which contains the address of the data. Let's take an example.

Loc	Data	Symbolic Address	Contents
100	035		STORE A (TALLY)
101	177		
102	000		HALT
177	303	TALLY	
301	000	TALLY1	
302	000	TALLY2	
303	000	TALLY3	
304		TALLY4	
305		TALLY5	
306		TALLY6	

The instruction in location 100/101 is a Store indirect. The second byte of this instruction points to location 177. Location 177 in turn points to location 303. The contents of the A register will be stored into location 303.

Why would such a round about method have any value? Like the other addressing methods, there are times when indirect addressing saves memory locations or work.

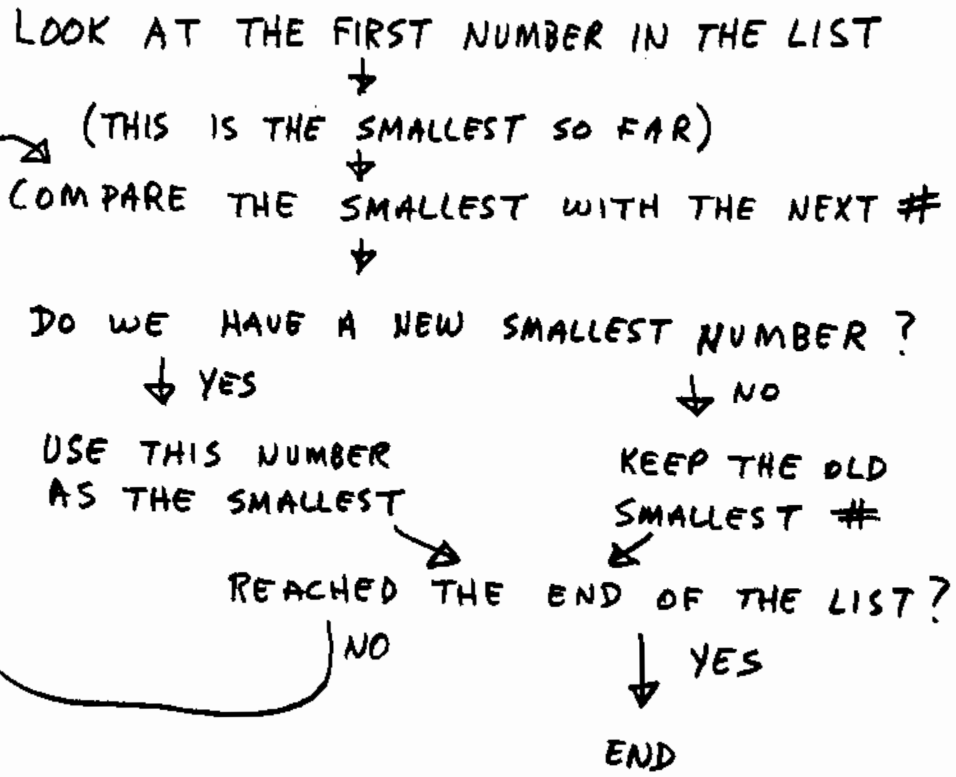
We'll now look at the problem of finding the smallest number in a list of numbers. If the list of numbers were 2, 5, 7, 3, 3, 2, 6, 9, 4, then the smallest number is 2. The fact that it occurs twice has no significance.

Our numbers will be octal in the range 0 to 177. They will be stored in a table or a list which starts with the symbolic address TABLE and ends with the symbolic address LAST. Thus, if we had three entries, we might have

```
TABLE  002
       003
LAST   001
```

Of course the table may be much longer than this. We'll write the program, symbolically at least, without knowing exactly how long the table is or where it is located in memory. When we do know we'll substitute the actual values for TABLE and LAST.

In our analysis we will show how one person approached it including several of his omissions and errors. We wish to exphasize that very few people can write a perfect program from scratch. Our programmer first wrote a very informal and general flowchart.


LOOK AT THE FIRST NUMBER IN THE LIST
(THIS IS THE SMALLEST SO FAR)
COMPARE THE SMALLEST WITH THE NEXT #
DO WE HAVE A NEW SMALLEST NUMBER ?
YES
NO
USE THIS NUMBER AS THE SMALLEST
KEEP THE OLD SMALLEST #
REACHED THE END OF THE LIST?
NO
YES
END

Next our programmer decided he could use two registers to good advantage. He let

B CONTAIN THE ADDRESS OF THE NUMBER
IN THE LIST BEING EXAMINED

A USE FOR COMPARISONS

Starting to write the symbolic program he put down

```
LOAD   B   C= TABLE
LOAD   A   (B)
ADD    B   C= 1
SUB    A   (B)
```

The first of these instructions sets the value of B for the first number in the list. The A register is loaded indirectly through the address in B. For example, if this were the case

Loc	Data	Symbolic Address	Contents
204	002	TABLE	FIRST NUMBER
205	003		SECOND NUMBER
206	001	LAST	THIRD NUMBER

then B contains 204 and A will contain 002 after the LOAD A instruction above. So far this is the smallest number. Next the number in B is increased by 1 and the next entry in the list is compared to the smallest number by subtracting the two. Right away our programmer realizes he has destroyed the copy of the smallest number, so he inserts an instruction to save it. Also if the result of the subtraction is larger than zero, the new number from the list is smaller than the previous smallest number.

```
             LOAD   B     C= TABLE
                                            STORE A C=
HERE  <--    LOAD   A     (B) -----{SMALLEST
             ADD    B     C= 1
             SUB    A     (B)
             JUMP   A>0   HERE
```

The programmer has noted that if A is greater than 0 and if the program jumps to the instruction marked HERE, the the new smallest number will be picked up from the table, stored or saved in SMALLEST, and the process will continue with a comparison to the next number in the list. On the other hand, if the subtraction leaves A zero or negative, the new number is not smaller. So the programmer adds the two instructions below.

```
             LOAD   B     C= TABLE
                                            STORE A C=
HERE  <--    LOAD   A     (B) -----{SMALLEST
THERE        ADD    B     C= 1
             SUB    A     (B)
             JUMP   A>0   HERE
             LOAD   A     SMALLEST
             JUMP         THERE
```

This provides the main logic of the comparison. What is missing now is the test to determine if the end of the table has been reached and the ending routine. The best place for testing whether the end of the list has been reached seems to be after the addition of one to the B register.

```
        LOAD   B     C=TABLE
HERE  ← LOAD   A     (B)       { STORE A C=
                                 SMALLEST
THERE   ADD    B     C=1
      ← SUB    A     (B)
        JUMP   A>0   HERE
        LOAD   A     SMALLEST
        JUMP         THERE
END     As required
                             SUB   B    C=LAST+1
                             JUMP  B=0  END
                             ADD   B    C=LAST+1
```

Now our programmer might observe that the two successive instructions

```
THERE   ADD   B   C=1
        SUB   B   C=LAST+1
```

could be replaced by the one

```
THERE   ADD   B   C=LAST
```

We could criticize our programmer on the choice of HERE and THERE as symbolic addresses. It would be more meaningful if HERE were replaced by NEWSMALL and THERE by OLDSMALL.

You can complete the program, choose a table location and entries for it, and test the program. In the end routine, have the smallest number displayed in the output lights. A worksheet is given on the next page.

A
0--

B
1--

X
2--

ADD
-0-

SUB
-1-

LOAD
-2-

STORE
-3-

IMMED
--3

MEMORY
--4

INDIRECT
--5

<u>JUMP</u>

≠0
-43

=0
-44

<0
-45

≥0
-46

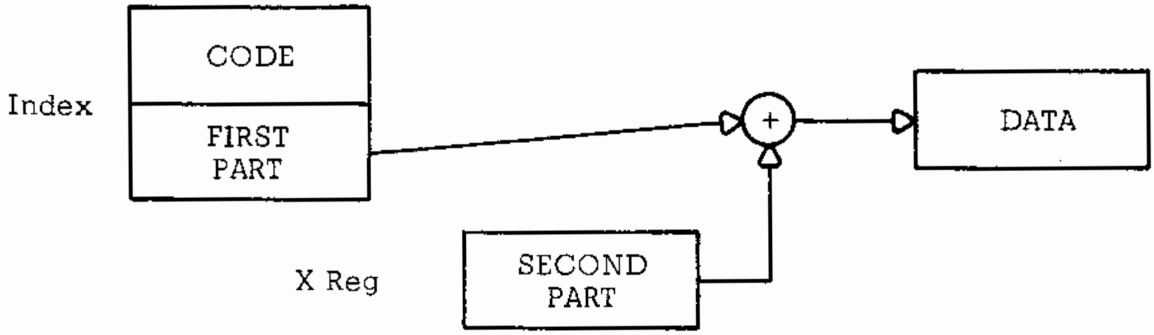
>0
-47

Loc	Data	Symbolic Address	Contents	Comments

Exercise 21

The times when one should use the immediate addressing mode are usually obvious. The memory addressing mode is the standard and usually it is the one which is used. It takes experience to appreciate the indirect addressing mode. We will learn two other addressing methods in this Exercise.

In indexed addressing the contents of the second byte of the instruction are added to the contents of the X register. The sum is the address of the data.



If the second byte of the instruction is a 200 and the contents of the X register are 004, then the data location is 204. The same result would have occurred with 004 in the instruction and with 200 in the X register or with 100 in the instruction and 104 in the X register.

The code for the indexed mode of addressing is obtained by changing the right hand octal digit of the instruction to a 6. The symbolic form of an indexed instruction is

```
ADD B TABLE, X
```

The index mode is very handy when one is working with a list of numbers or a table. Usually, the component from the instruction itself will be one of the end points of the table. It will be the first or last address in the table. The component contributed by the X register will be a relative location within the table. For example, if we have

300	TABLE	1st entry
301		2nd entry
302		3rd entry
303	LAST	4th entry

then

1. The second byte of the instruction may be 300, and X varies from 0 to 3, or

2. The second byte of the instruction may be 277, and X varies from 1 to 4, or

3. The second byte of the instruction may be 303, and X varies from -3 to 0, or

4. The second byte of the instruction may be 304, and X varies from -4 to -1.

And there are more possibilities than these.

Let's apply the index mode to the simple problem of Exercise 14 which was the addition of 50 small numbers. We had the 50 numbers in locations 204 through 265 inclusive.

It could be done this way

```
            LOAD  X  C=061      = 49 decimal
            LOAD  A  C=0
LOOP        ADD  A  TABLE,X
            SUB  X  C=1
            JUMP X ≥ 0 LOOP
 END        ---
TABLE       FIRST NUMBER

 LAST       50th NUMBER
```

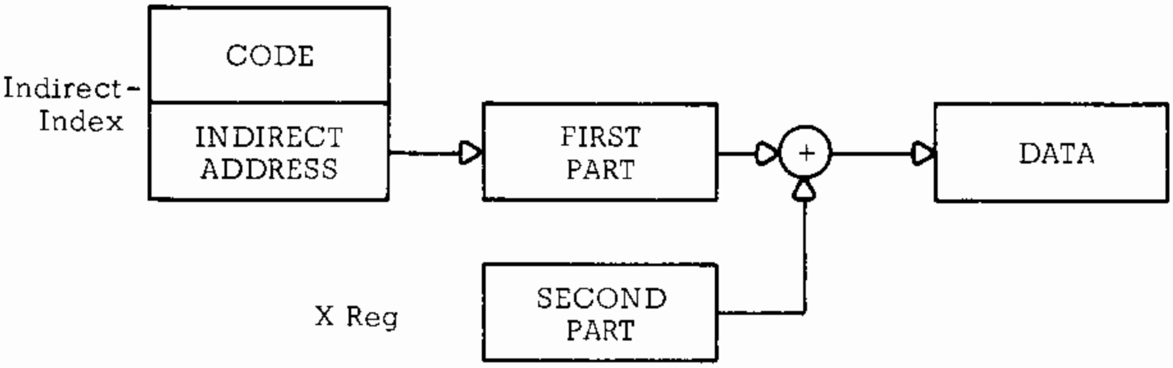
Here we add the 50th number first and the first number last. The reason for doing it this way is that we can test the X register very easily when it goes from zero to a negative number. It would not be as easy to test when X went from 49 to 50.

You should compare the number of instructions above with the number we used in Exercise 14. The new addressing modes have allowed us to save half of the number of bytes we used then. Again we emphasize that we can accomplish nothing new or different. It is just an easier way.

We'll mention the last addressing method which is called indirect-indexed. It combines the features of the indirect mode, which occurs first, and the features of the index mode which occurs second. The second byte of the instruction is the address of another byte. The contents of this second byte are added to the X register and the sum becomes the data location. Its symbolic form is

STORE A (LIST),X

This instruction code is obtained by making the right digit a 7. Our diagram for this mode is



The following fun type of problem illustrates several things including

1. Index addressing
2. Table look up
3. Delay

The problem is easy though it is harder to describe. Basically we will create the illusion of movement in the output lights from a series of non-moving displays. Perhaps the best illustration is a sign on which the news moves across. Some score boards operate this way and the basic principle is the same as movies or TV. We are limited with our few lights but still we can have fun.

In a table in the memory, we will store individual frames or "snap shots". The program will cycle among the bytes in the table. When it comes to a new byte it will place it in the output lights. Unless we slowed the computer down it would run too fast. So we'll add some delay.

Let's suppose the table contains

```
00000001
00000010
00000100
00001000
00010000
00100000
01000000
10000000
01000000
00100000
00010000
00001000
00000100
00000010
```

With the proper choice of delay, the display resulting from this table will be that of a lighted dot which bounces back and forth, left and right. Everyone can make up his own table and they will vary in size. All of the tables can start at the same location, say 300. Each person who makes up a table must put the address of the first byte after his last entry into the A register.

The program is quite simple.

Loc	Data	Symbolic Address	Contents	Comments
		P	START	
		A		"Last" address
		START	LOAD X C=0	1
		REPEAT	LOAD B TABLE,X	2
			STORE B OUTPUT	3
		LOOP	LOAD B C=040	4
		DELAY TIME		
			SUB B C=001	5
			JUMP B≠0 LOOP	6
			ADD X C=1	7
			SUB X A	8
			JUMP X=0 REPEAT	9
			ADD X A	10
			JUMP REPEAT	11
300		TABLE		Table starts here

Let's examine it. Instruction number 1 sets X equal to 0. Instruction 2 loads B from the table. The value of X determines the byte within the table. The number we have "looked-up" goes to the output lights. Instructions 4, 5, and 6 create a delay. The technique is to load a register (B in this case) with a number and to count it down to zero. This takes time which creates a delay. Changing the number in symbolic location DELAY TIME changes the delay. (You may want to experiment with the delay time).

The instruction in 7 increases the value of X by one. Next we test to see if we have reached the last address which we had decided to put in the A register. If X is equal to zero, we jump back to REPEAT and start the cycle again. If X is not equal to zero, we add the amount subtracted out for the test and then jump back to fetch the entry in the table.

A worksheet on the next page can be used to complete the program. If you try it with your own patterns, remember the best illusion of motion occurs when the difference between "frames" is relatively small.

A
0--

B
1--

X
2--

ADD
-0-

SUB
-1-

LOAD
-2-

STORE
-3-

IMMED
--3

MEMORY
--4

INDIRECT
--5

INDEX
--6

IND-IND
--7

JUMP

≠0
-43

=0
-44

<0
-45

≥0
-46

>0
-47

Loc	Data	Symbolic Address	Contents	Comments

Exercise 22

We have completed the addressing modes. In this and the following Exercises we will learn some new instructions. The three that we will learn in this Exercise apply only to the A register. They cannot be used with the B or X registers. They have the five addressing modes and address codes that we learned.

Many times one byte will contain more than one item of data. We had an example of this when we had two BCD digits in one byte. How can we separate the individual items within a byte? For example, suppose we have a byte with three items of data, A, B, and C.

```
                  AA   BBB   BCC
```

which we wish to separate into

```
                  AA   000   000
            and   00   BBB   B00
            and   00   000   0CC
```

One of our new instructions, AND, permits us to do this.

The opposite problem occurs when we want to merge data. We can combine separate items of data such as A, B, and C above into one byte. In some cases we can use an Add instruction. We could in the example just given. In other cases the use of the Add instruction many not work because of carries between bit positions. Here is a case which would not work. We have a byte of data,

```
                  0X   000 000
```

in which we want to set the X bit to a 1. However, we don't know whether X is now a 0 or a 1. If we used the Add instruction, we could get two results.

```
        If X = 0                 If X = 1
         /                        /
      00   000   000           01   000   000
    + 01   000   000         + 01   000   000
      ---------------          ---------------
      01   000   000           10   000   000
```

We get the result we want in the first case but not in the second case. While there are ways one could do this, the second of our new instructions, OR, allows us to do this easily.

In the AND instruction the answer consists of 8 bits (one byte). Each bit is determined from the two bits in the data and in the A register which are in the same position. A bit in the answer, which is stored in the A register, is a 1 if and only if the two corresponding bits in the data and in the original A register were both 1's. For example,

	00	110	011	Original A
AND	01	010	101	Data
	00	010	001	Answer (in A)

In the OR instruction the answer consists of 8 bits also. Again each bit is determined by the two bits in the same position of the A register and the data. A bit in the answer, which is stored in the A register, is a 1 if either or both of the corresponding bits in the data and the A register are 1's. For example, with the same two operands as above,

	00	110	011	Original A
OR	01	010	101	Data
	01	110	111	Answer (in A)

The codes for the AND or OR instructions are

AND	32M	XXX
OR	30M	XXX

where M is the addressing mode and XXX is the second byte of the instruction.

The AND and OR instructions perform logical operations. We don't wish to imply that Add and Subtract are illogical operations; we speak of them as arithmetic operations.

Our third new instruction, LNEG, is an arithmetic operation. It loads the A register with the negative value of the data. If the data is already negative, then A will be loaded with a positive number. The instruction

```
LNEG  A  DATA
```

produces the same result as

```
LOAD  A  ZERO
SUB  A  DATA
```

The code for LNEG (Load Negative) is

```
LNEG     33M   XXX
```

where M is the addressing mode and XXX is the second byte of the instruction.

If one wanted to change all the 0 bits in a byte to 1's and all of the 1 bits in a byte to 0's, for example,

```
        01  110  011
  to    10  001  100
```

it could be done this way

```
LNEG A  DATA
ADD  A  C=377
```

To show an example, let the DATA content be 01 110 011. Using the LNEG instruction we would get

```
       this            10  001  101
and then adding        11  111  111
                       ------------
      we get           10  001  100
```

By now you know how to demonstrate the operation of individual instructions with the computer. It is recommended that you do so for each of the instructions. When you try the LNEG instruction, use the data 10 000 000. You will see that an incorrect result is obtained. The reason is that the number 10 000 000 represents -128. In the signed representation of numbers for the computer, there is no +128.

If DATA contains 10 000 000, and the following operation is performed

```
LNEG A  DATA
```

what result is obtained in the A register? ______________________________

Here is a problem for you. There are six conditions, U, V, W, X, Y, and Z. Each condition is true or false as indicated by the least significant bit in each of six locations. We'll give the locations the same name as the condition. We'll let a 1 bit indicate the condition is true and a 0 bit indicate it is false. For example, if U contains 00 000 000, then U is false. If U contains 00 000 001, then U is true.

Program a solution to the following question:

Is it true that

1) U and V and W are all true,

or 2) X and Y are both true,

or 3) Z is not true?

Here are some test conditions to try (we give the correct answer also)

	U	V	W	X	Y	Z	Question	
Test 1	0	0	0	0	0	0	TRUE	by 3) above
2	0	0	0	0	0	1	FALSE	
3	1	1	0	1	0	1	FALSE	
4	1	1	1	1	0	1	TRUE	by 1) above
5	1	1	1	1	1	1	TRUE	by 1) and 2) above

A worksheet is given on the next page.

The code for LNEG (Load Negative) is

LNEG   33M XXX

where M is the addressing mode and XXX is the second byte of the instruction.

If one wanted to change all the 0 bits in a byte to 1's and all of the 1 bits in a byte to 0's, for example,

01 110 011
to 10 001 100

it could be done this way

LNEG A DATA
ADD A C=377

To show an example, let the DATA content be 01 110 011. Using the LNEG instruction we would get

this 10 001 101
and then adding 11 111 111
we get 10 001 100

By now you know how to demonstrate the operation of individual instructions with the computer. It is recommended that you do so for each of the instructions. When you try the LNEG instruction, use the data 10 000 000. You will see that an incorrect result is obtained. The reason is that the number 10 000 000 represents -128. In the signed representation of numbers for the computer, there is no +128.

If DATA contains 10 000 000, and the following operation is performed

LNEG A DATA

what result is obtained in the A register? ______________________________

Here is a problem for you. There are six conditions, U, V, W, X, Y, and Z. Each condition is true or false as indicated by the least significant bit in each of six locations. We'll give the locations the same name as the condition. We'll let a 1 bit indicate the condition is true and a 0 bit indicate it is false. For example, if U contains 00 000 000, then U is false. If U contains 00 000 001, then U is true.

Program a solution to the following question:

Is it true that

1) U and V and W are all true,

or 2) X and Y are both true,

or 3) Z is not true?

Here are some test conditions to try (we give the correct answer also)

	U	V	W	X	Y	Z	Question	
Test 1	0	0	0	0	0	0	TRUE	by 3) above
2	0	0	0	0	0	1	FALSE	
3	1	1	0	1	0	1	FALSE	
4	1	1	1	1	0	1	TRUE	by 1) above
5	1	1	1	1	1	1	TRUE	by 1) and 2) above

A worksheet is given on the next page.

OR
30-

AND
32-

LNEG
33-

ADD
-0-

SUB
-1-

LOAD
-2-

STORE
-3-

A
0--

B
1--

X
2--

IMMED
--3

MEMORY
--4

INDIRECT
--5

INDEX
--6

IND-IND
--7

Loc	Data	Symbolic Address	Contents	Comments

Exercise 23

In the last Exercise we learned of ways to manipulate the individual bits in a byte. Through use of AND and OR instructions we could set individual bits to 0 or to 1. In this Exercise we'll learn of another way to control and test individual bits. In some cases the new way is preferable while in other cases the AND and OR instructions are better.

The new instructions have a different format and code structure. They still have two bytes and the second byte is an address of a byte in the memory. Only memory addressing is allowed.

The first instruction, Set 0, sets a bit in the memory to a 0. We may give the location and the bit position within the byte. The code is

```
0B2  XXX
```

B is an octal digit which specifies the bit position. Bit positions are numbered 7, 6, 5, 4, 3, 2, 1, 0 corresponding to the front panel labeling. XXX is the address of the location to be changed. For example,

```
072
200
```

sets the most significant bit of the output location to a 0.

A similar instruction, Set 1, sets a bit in the memory to a 1. Its code structure is

```
1B2 XXX
```

The instruction

```
102
200
```

sets the least significant bit of the output location to a 1.

Since the A, B, and X registers have addresses, these instructions can be used to set bits in A, B, or X. However, a major advantage of the Set 0 and Set 1 instructions is that they can be used to control bits outside of A, B, or X without changing A, B, or X.

The Set 0 and Set 1 instructions manipulate or change data. Two other instructions, with some points of similarity, permit individual bits to be tested.

The Skip 0 instruction will cause the P register to advance an extra two locations if the bit specified by the instruction is a 0. A bit is specified in the same way as for a Set 0 or Set 1 instruction. The second byte of the instruction is a memory address. The bit within this byte of data is specified by the second octal digit of the first byte of the instruction. The code for Skip 0 is

2B2 XXX

The code for the Skip 1 instruction is

3B3 XXX

The result produced by these instructions is easier to remember if you read the name of the name of the instruction as

Skip on the specified bit equal to 0 (or 1)

If the specified bit is not equal to the condition given, then the next instruction is the instruction following this instruction. In this case the result is much like a two byte "do-nothing" instruction. If the specified bit is equal to the condition given, then the two byte instruction following this Skip (or the two one byte instructions) are omitted or not executed. The P register skips over these two locations.

Let's demonstrate these operations with the computer. Load this program:

Loc	Data	Symbolic Address	Contents
000	001	A	001
003	004	P	004
200	000	OUTPUT	000
004	202		SKIP 0 b0 A
005	000		
006	172		SET 1 b7 (200)
007	200		
010	000		HALT

First, from a study of the program predict what the output lights will be after you push Start. ______________________________

Push Start. Were you correct? ________________________

Now change P back to 003. Change A to 000. Do not change OUTPUT.

What do you predict the output lights will be after you push Start? ______________________________

Push Start. Were you correct? ______________________________

Set P equal to 003, A to 000, and OUTPUT to 000.

What do you predict the output lights will be after you push Start? ______________________________

Push Start. Were you correct? ______________________________

At the end of Exercise 22 we gave you a problem to program using And, Or, and Lneg instructions. Solve the same problem using only the Set, Skip, and Jump instructions. There is a worksheet on the next page.

SET 0
0B2

SET 1
1B2

SKIP 0
2B2

SKIP 1
3B2

JUMP

UNC
344

≠0
-43

=0
-44

<0
-45

≥0
-46

>0
-47

Loc	Data	Symbolic Address	Contents	Comments

Exercise 24

We now come to a group of related instructions for which there may not seem to be much need. After you have some programming experience, you'll see more value in them. Basically the instructions move data within a byte and usually they don't generate any new bits. And sometimes they throw bits away. Let's describe the instructions first and then we'll see how to use them. The four types of instructions are

Shift Left
Rotate Left
Shift Right
Rotate Right

These are one byte instructions. They operate only on the A or B registers as specified but not on both at the same time. The amount of shifting or rotating may be specified as 1, 2, 3, or 4 bit positions. We'll describe all of the operations for only one bit position. For three bit positions, the result is the same as doing it for one bit position three times in a row.

A Rotate Left 1 instruction moves all bits in the byte one place to the left. The bit that was at the left end (the most significant bit) is carried back around to the right end to become the least significant bit. For example,

Bit position	7	6	5	4	3	2	1	0
Original bits	a	b	c	d	e	f	g	h
Final bits	b	c	d	e	f	g	h	a

A Rotate Right 1 instruction moves the bits one place to the right. The bit that was at the right end (the least significant bit) is carried around to the left end to become the most significant bit. An example is

Bit position	7	6	5	4	3	2	1	0
Original bits	a	b	c	d	e	f	g	h
Final bits	h	a	b	c	d	e	f	g

The Shifts differ from the Rotates in two ways. No bits are carried around the end. The most significant bit or the least significant bit is treated in a special way. In a Shift Left 1, these are the results

Bit position	7	6	5	4	3	2	1	0
Original bits	a	b	c	d	e	f	g	h
Final bits	b	c	d	e	f	g	h	0

If you were to add a byte of data to itself, you would get

$$\begin{array}{r} a\ b\ c\ d\ e\ f\ g\ h \\ +\underline{a\ b\ c\ d\ e\ f\ g\ h} \\ b\ c\ d\ e\ f\ g\ h\ 0 \end{array}$$

which is the same result as a Shift Left 1. That the addition does yield the result claimed may be shown by considering that multiplication by 2 in binary is the same as multiplication by ten in decimal. All the digits move one position to the left and a zero is added at the right. Therefore a Shift Left 1 is equal to multiplication by 2. This is only true if the answer is not larger than what can be represented in one byte. If we start with the number 5

	00	000	101	
after one Left Shift we have	00	001	010	= 10 decimal
after another Left Shift	00	010	100	= 20 decimal
after another Left Shift	00	101	000	= 40 decimal
after another Left Shift	01	010	000	= 80 decimal
after another Left Shift	10	100	000	= 160 decimal
			or	= -96 decimal

If you are using a signed convention for your numbers, overflow occurred on the last shift. If you are using a positive only convention, overflow would occur on the next shift.

Similar results are produced with negative numbers. Let's take -1

11 111 111

After two Shift Left 1 instruction we would have

11 111 100

which is equal to -4 (-1 x 2 x 2 = -4). If we continued this we would eventually get overflow. The Shift Left 1 instruction is multiplication by 2 if no overflow occurs. When overflow occurs the result is greatly in error.

The Shift Right instruction is equal to division by 2 with the remainder being thrown away. The sign is treated in a special way. For example,

Bit position	7	6	5	4	3	2	1	0
Original bits	a	b	c	d	e	f	g	h
Final bits	a	a	b	c	d	e	f	g

Look also at a Shift Right 4

Original bits	a	b	c	d	e	f	g	h
Final bits	a	a	a	a	a	b	c	d

It is done in this way to preserve the sign of the original number. For example, a Shift Right 2 of -4

	11	111	100	-4
yields	11	111	111	-1

Give the results of the following operations applied to the data.

Rotate Right 1   10 111 000   ____________________

Rotate Right 3   01 011 100   ____________________ a

Rotate Right 4   10 111 000   ____________________ b

Answers a and b should be the same. Do you see why?

Rotate Left 1   01 011 100   ____________________

Rotate Left 4   10 111 000   ____________________ c

Answers b and c should be the same. Can you make up a rule here?

Shift Left 1   11 000 110   ____________________

Shift Left 4   11 000 110   ____________________ d

Shift Right 4   11 000 110   ____________________ e

Answers d and e should not be the same.

For signed representation of positive and negative numbers, does overflow occur in these cases?

Shift Left 1   00 101 101   ____________________

Shift Left 2   00 101 101   ____________________

Shift Left 1   11 010 010   ____________________

Shift Left 2   11 010 010   ____________________

If you do a Shift Right 4 followed by a Shift Right 4 what are the possible outcomes?____

The codes for these instructions, when applied to the A register are

Right Shift	0N1
Right Rotate	1N1
Left Shift	2N1
Left Rotate	3N1

Where N	
1	1 place
2	2 places
3	3 places
0	4 places

For the B register add 4 to the value of N. These instructions have only one byte.

You should try these instructions in the computer. You can check your answers above and try other data values also.

The Rotate instructions are useful for moving bits around. Perhaps you might want to test whether bit 6 in the A register is a 0 or a 1. You could do this in several ways, one of which is to perform a Rotate Left 1 followed by a conditional Jump instruction.

The Shift instructions are arithmetic operations. They are a fast way to multiply or divide by 2, 4, 8, or 16.

Exercise 25

We often have the same sequence of instructions repeated at different places in a program. The repeated sequence might perform a function such as conversion of a BCD number to binary or getting an input number. A flow chart for a situation of this type might look like Figure 25.1. At three different places in the program, we might use the same instructions

```
LOAD  A  C = 0
STORE  A  INPUT
HALT
```

In this Exercise we'll show a way to make one set of instructions serve for all three places. The general idea goes by the name "subroutine".

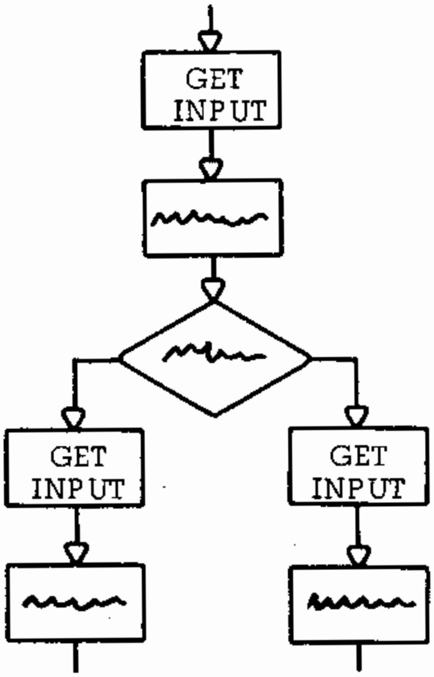


FIGURE 25.1

Subroutines are different from looping. It is true that a program which loops does use the same set of instructions more than once but these occur at only one place in the program. Subroutines would apply to the same sequence of instructions which are used at different places in a program. A program loop may have a subroutine and a subroutine may have a loop.

Load these instructions in the computer and push Start.

003	004	P	
004	200		NO-OP
005	024		LOAD A P
006	003		
007	000		HALT

When the computer halts, what does A contain? ____________

The NO-OP instruction may be ignored. The LOAD instruction puts the value of the P register into the A register. At the moment that the transfer is actually made, the P register has the value which is the address of the LOAD A P instruction.

Let's extend the program in the computer and divide it into three parts

003	004	P		Part 1
004	200		NO-OP	
005	024		LOAD A P	
006	003			
007	344		JUMP 300	
010	300			
011	200		NO-OP	
012	000		HALT	

100	024		LOAD A P	Part 2
101	003			
102	344		JUMP 300	
103	300			
104	200		NO-OP	
105	000		HALT	

300	034		STORE A OUTPUT	Part 3
301	200			
302	000		HALT	

First, set P equal to 004. Push Start.

After halting, what does A contain? ____________________

Next, set P equal to 100. Push Start.

Now what does A contain? ____________________________

If the program comes to part 3 from part 1, A contains 005. If the program comes to part 3 from part 2, A contains 100. Now suppose we wanted to go back, from part 3, to the part we came from. If we made part 3 into the following

300	034		STORE A OUTPUT
301	200		
302	000		HALT
303	003		ADD A C = 4
304	004		
305	034		STORE A JP ADDR
306	310		
307	344		JUMP XXX
310	---	JP ADDR	

we will return to the part from which we came to part 3.

This illustrates the subroutine concept. Part 3 is the subroutine (not a very useful one in this case). Part 1 and part 2 are two different sections of the program which can both use part 3, the subroutine. Load the complete part 3, do single instructions and observe the contents of the registers when you start with values of P equal to 004 and 100.

There are other ways that we achieve the same thing. The common feature is to create a record (an address) which indicates to the subroutine what other part of the program is using the subroutine. This record "marks" where we are to return after the subroutine is completed. In our example above, this record was maintained in the A register.

A new instruction, Jump and Mark, makes this process much easier. The Jump and Mark set of instructions has all of the characteristics of the Jump instructions which we have learned. It may be conditional based on one of five conditions for A, B, or X or the Jump and Mark may be unconditional.

If the Jump is made, with the Jump and Mark instruction, the address of the Jump and Mark instruction plus two is stored in the location whose address is in the second byte of the instruction. The next instruction is taken from the location following this Mark record. Let's look at an example to help straighten out the confusion. Load this.

003	100	P	

100	364		JUMP & MARK 300
101	300		

300	---		
301	000		HALT

The Jump and Mark above is unconditional.
Push Start.

P has the value ______________________________

The contents of 300 are ________________________

The Mark address is stored in location 300. In this case the Mark address is 102 which is two more than the address of the (first) byte of the Jump and Mark instruction. When the subroutine (not shown here) is completed, it would probably "return control" to the instruction in location 102.

The Jump Indirect instruction is an easy way to return control from the subroutine to the program segment which used the subroutine. The word "indirect" has exactly the same meaning as it does for indirect addressing. The second byte of a Jump Indirect instruction contains the address of the location which contains the address of the next instruction to be executed. Let's expand our previous demonstration program to include an unconditional Jump Indirect instruction.

003	100	P	

100	364		JUMP & MARK 300
101	300		
102	000		HALT

300	000		
301	000		HALT
302	354		JUMP INDIRECT (300)
303	300		

Do single instructions and record the values of P

1. P is 100
2. P is ______
3. P is ______
4. P is ______

Location 300 contains ______

Initially (300) are 000. After the Jump and Mark instruction, location 300 contains 102. When the Jump Indirect instruction is executed, the computer uses the 300, from the second byte, to find the address in 300 of the next instruction.

In our little example, the subroutine is so simple as to be ridiculous. In general, subroutines will be more complex than shown here.

There is also a Jump and Mark Indirect instruction. The indirect feature occurs first followed by the Mark feature. You can study this instruction by yourself.

continued . . .

The names, codes, symbolic representations, and the conditions for all of the Jump instructions are summarized below.

NAME	CODE		SYMBOLIC		
JUMP or JUMP DIRECT	U 4 V	XXX	JUMP or JPD	W	XXX
JUMP INDIRECT	U 5 V	XXX	JPI	W	(XXX)
JUMP & MARK DIRECT	U 6 V	XXX	JMD	W	XXX
JUMP & MARK INDIRECT	U 7 V	XXX	JMI	W	(XXX)

U	V
0 A register	3 $\neq$ 0
1 B register	4 = 0
2 X register	5 $<$ 0
3 Unconditional	6 $\geq$ 0
	7 $>$ 0

Must be one of these values

W

(A, B, or X) combined with ($\neq$ 0, = 0, $<$ 0, $\geq$ 0, $>$ 0)
or UNC

Previously, we have used the name JUMP. With more types of Jump instructions, we need to refine our description. By including "Direct" and "Indirect" in the name we indicate the addressing mode. The name "Mark" tells when this feature is present. In the symbolic form we abbreviate these as D, I, and M. With no Mark feature, we use the letter P.

Exercise 26

We'll continue our work with subroutines and we'll comment about computers and programming. As we have been learning new features and new instructions, we have kept repeating that we couldn't do anything that we couldn't do with our first set of instructions. The new addressing modes and the new instructions made some things easier to do.

* * *

A computer must only be able to do only a few simple kinds of operations.

More complex operations may be built up from combinations of simpler operations.

The speed and accuracy of a computer make it reasonable to have very complicated combinations of elementary operations.

As programmers, our task is to find combinations of operations for solving our problems. This task is not always easy.

* * *

The Add instruction is not a necessity. We could omit it. We'll explain how by developing a subroutine for additions.

The addition of a + b can be performed in this way:

$$a + b = a - (-b)$$

If we have a Subtract instruction, we don't need an Add instruction. It is easier if we do have an Add instruction, but for the time being let's assume that we don't.

When we need to add a number (in NUM) to a number already in A, we can use these instructions

```
                 STORE   A   C =
  TEMPORARY 1
                 LOAD    A   C = 0
                 SUB     A   NUM
                 STORE   A   C =
  TEMPORARY 2
                 LOAD    A   TEMPORARY 1
                 SUB     A   TEMPORARY 2
```

Since addition is a common requirement we might choose to make this into a subroutine

```
        ADD           ---
                      STORE   A     C =
    TEMPORARY 1       LOAD    A     C = 0
                      SUB     A     NUM
                      STORE   A     C =
    TEMPORARY 2       LOAD    A     TEMPORARY 1
                      SUB     A     TEMPORARY 2
                      JPI     UNC   ADD
```

If the main part of the program requires addition we will jump to the ADD subroutine which will do the addition. When this is done, the subroutine will return us to where we came from the main program.

However, we have one remaining problem. Each part of the main program that uses the subroutine will generally need to add a different number. As we have written the subroutine we always use the number in NUM.

When the main part of the program uses the subroutine, it could do this

```
                      LOAD    B     DESIRED NUMBER
                      STORE   B     NUM
                      JMD     UNC   ADD
```

Before using the subroutine, the main part of the program places the number to be added to A in NUM. Then our subroutine above would work.

Another way is as follows. The B register will contain the number to be added. Then the main part of the program "calls" the subroutine by this sequence

```
                      LOAD    B     DESIRED NUMBER
                      JMD     UNC   ADD
```

but we have to change this instruction in the subroutine

```
                      SUB     A     NUM
                  to  SUB     A     B
```

Another way is as follows. The B register will contain the address of the number to be added. Then the subroutine is called by this sequence

```
                LOAD     B      C = Address of desired number
                JMD      UNC    ADD
```

and in the original subroutine

```
                SUB      A      NUM
is replaced by  SUB      A      (B)
```

We are trying to illustrate that a set of conventions usually applies to a subroutine. We must be sure that we understand these. What do we have to do to call the subroutine? Where will be the answer when we return from the subroutine? Are there any restrictions? What registers does it use?

Just because we want to divide two numbers and there is a subroutine called DIVIDE, it doesn't mean we can use this subroutine. Perhaps the subroutine assumes the answer will be less than one and we have an answer greater than one.

Here is a two part problem for you. First, write a subroutine to multiply two small numbers together. (We say "small" so the answer won't exceed the capacity of one byte.) If U and V are the two small numbers, the multiplication can be performed by adding V together U times.

For the second part of the problem write a program which will find the volume of a box whose sides are J, K, and L. Use your subroutine to find J x K and then multiply this result by L. Find the volume of this box: 4 x 5 x 6.

There is a worksheet on the next page.

	Loc	Data	Symbolic Address	Contents	Comments
Jump					
RTC					
R=0, A					
R=1, B					
R=2, X					
R=3, UNC					
T=4, JPD					
T=5, JPI					
T=6, JMD					
T=7, JMI					
C=3, ≠0					
C=4, =0					
C=5, <0					
C=6, ≥0					
C=7, >0					
Other					
RKA					
K=0, Add					
K=1, Sub					
K=2, Load					
K=3, Store					
A=3, Immed					
A=4, Mem					
A=5, Indirect					
A=6, Index					
A=7, Ind/Ind					

See next page also.

Loc	Data	Symbolic Address	Contents	Comments

Exercise 27

Every time that we add or subtract to the A, B, or X register, the computer determines and stores a carry bit and an overflow bit. If we were nit-pickers, we would note that the carry bit should be called a borrow bit for subtractions. We're not, so we use the name "carry" for both addition and subtraction.

First, let's say where these bits are stored and then we will say what they are. For the A register, the carry and overflow are stored in location 201, for B in 202, and for X in 203. The overflow bit is the least significant bit (b0). (Note: Can you remember Overflow and b0?) The carry bit is in the adjacent position, b1.

Whenever an Add or Subtract instruction is used with the A register, the carry and overflow bits in location 201 are updated. This is the only time these bits are changed except for things like Store B (201). The bits in location 202 are updated when the addition or subtraction is made to B register, in location 203 for addition and subtraction to the X register. The other six bits in these locations are always set to 0.

The carry bit is the easiest to explain. If you add two eight bit numbers, the carry out of the left most position (b7) is the carry bit which we have defined above. In these examples, you are to tell what the carry bit will be.

```
  10 001 100        10 001 100
+ 01 101 110      + 11 101 110


  10 001 100        10 001 011
- 10 001 100      - 10 001 100
```

Are you able to check your answer in the computer?________

What good is the carry bit? Here are a couple of uses. For numbers which are considered to be positive, with a range from 000 to 377 octal, the carry indicates that overflow has occurred. We have added two numbers whose sum is larger than 377 or we have subtracted one number from a smaller number.

The second use for the carry is in multiple precision addition and subtraction. A double precision number (for us) is 16 bits long and is stored in two locations. If we add two double precision numbers, for example

```
                 Possible carry  (in this example, the carry is a 1)
     10 011 101 | 11 110 100
   + 01 000 100 | 01 101 001
```

there may be a carry from the least significant half of the addition into the most significant half. This would be the carry that we detect (or the borrow in subtraction).

The overflow bit pertains to a signed representation of numbers. If our convention is that the number range is -128 to +127 decimal (or -200 to +177 octal), then the overflow bit is a 1 if the correct answer to the addition or subtraction is outside this range. Otherwise the overflow bit will be a 0.

Tell whether the cases below generate overflow or not. The numbers are octal and you may have to convert them to binary to do the arithmetic.

```
      + 120
  (+) +  65                 Does overflow occur? ____________

      - 112
  (+) -  73                 Does overflow occur? ____________

      +  43
  (-) - 162                 Does overflow occur? ____________
```

Double check your answer by using the computer and a couple of instructions.

Here is a problem for you to program. Write a subroutine for double precision addition. The first number will be in B (most significant half) and in A (the least significant half). The number to be added will be stored in two consecutive locations in the memory. The least significant half will be the location which has the smaller address. This address will be in the X register.

One person came up with this solution which has an error in it. Can you find and correct his error?

```
ADD  DP ---
          ADD      A    (X)
          ADD      X    C = 1
          ADD      B    (X)
          SKIP 0   b1   (201)
          SUB      B    C  =  1
          JPI     UNC  ADD  DP
```

His error was ______________________________

which should be______________________________

Correct, complete, and test the subroutine with the computer. A worksheet is given below.

Loc	Data	Symbolic Address	Contents	Comments

Exercise 28

In this Exercise we'll solve a problem involving sorting. Along the way we'll take some time to talk about general problems.

Our problem starts with these conditions.

1. We have a table of numbers, with at least two numbers, stored in the memory. The table begins with symbolic address TABLE and ends with LAST.
2. A number in the table is positive, from 0 to 100 (decimal), These numbers, in octal, are stored one per location.
3. There may be repeated numbers, i.e., 37 may appear three times. Some numbers may not be present.
4. The original order of the numbers may be described as random.

We are to write a program to re-arrange these numbers within the table so that they will be in ascending order or at least a non-descending order. In short, we are to put the numbers in order. For example, these numbers are not sorted: 1, 5, 6, 3, 2, 0, 7, 1. When sorted the sequence becomes 0, 1, 1, 2, 3, 5, 6, 7.

*   *   *

Many people have a difficult time at this point. We speak now about the general case of problem solving and not just about this particular problem. First, they may not understand the problem. Second, they may not know how to proceed toward finding a solution. The following ideas are sometimes helpful.

Usually it is a good idea to forget that a computer is involved. Can we re-state the problem in more familiar terms? Can we find a "manual" solution that we could do ourselves? Answers to these questions may give us some insights and clues.

In our particular problem, it is the same as the problem of sorting cards which have the number written on them. When given to us, the cards are all scrambled up. We have to sort them in order.

Still forgetting that we have a computer, we can ask ourselves how we would solve the problem manually. Think of an many ways as you can and see what the advantages and disadvantages are. Then try to translate a selected method into computer terms. How would the computer do it?

At this point you should try this process yourself before reading on.

* * *

Using the cards as our model, these ways suggest themselves (you may have thought of more):

1. Look through the cards for all of them with the number 0. Put these at the front. Next look for all of the cards with the number 1. Put these behind the 0 cards. Continue in this way until all of the numbers have been examined.

2. Sort the cards into ten piles by their unit's digit. Pick up the pile of cards ending in 9, then pick up the cards ending in 8 and put them behind the 9's. Continue this pattern until the cards ending in 0 are at the back. Next go through the deck from the front and sort them again into 10 piles but use the ten's digit this time. Thus ten new piles are formed, one of each ten's digit. Pick up the pile for 00 to 09, next put the pile for 10 to 19 behind these, and so on. The cards will be sorted when you have done this. (If you don't believe it, try it).*

3. Imagine the cards are spread in your hand like playing cards. Compare two adjacent cards and put the larger one on the right and the smaller one on the left. A pair may already be in order. If they aren't, exchange the position of the pair of cards. Keep comparing and interchanging cards until no more exchanges are possible. The cards are then sorted.

The first and third methods are simple but may involve a lot of card moving or comparisons. The second method has the least amount of card handling. At the most, in this method, you'll look at a card twice. The second method is more complicated to perform within the computer. We'll use the third method since it seems to be the simplest to do.

* A slight extension of this method is required in our problem because there are three digit numbers.

Usually a good approach for writing a program is to identify the heart of it and to work with that for a while. This tends to clear up ideas. Then it will be obvious usually how the beginning and the end should relate to this.

Let's put into words what we might do. We'll compare the first and the second number. If necessary, we'll interchange their positions in the table so that the first of the pair is not larger than the second number. Next we'll compare the second and the third number, and interchange them if necessary. We continue in this way until we have compared the next-to-last and the last number. This is one pass. We'll repeat making passes until we have a pass without any interchange. Then we are done.

Many times this is a good point at which to commit tentative ideas to a flowchart. In Figure 28.1 we give a flowchart in very broad statements which is incomplete. We must have a method to detect whether an interchange was made during the last pass.

There is a technique of using "flag" bits that we can use. Actually we need only one bit. We'll reset this bit (make equal to 0) before we start a pass. If we do interchange a pair of numbers, we'll set this bit to a 1.

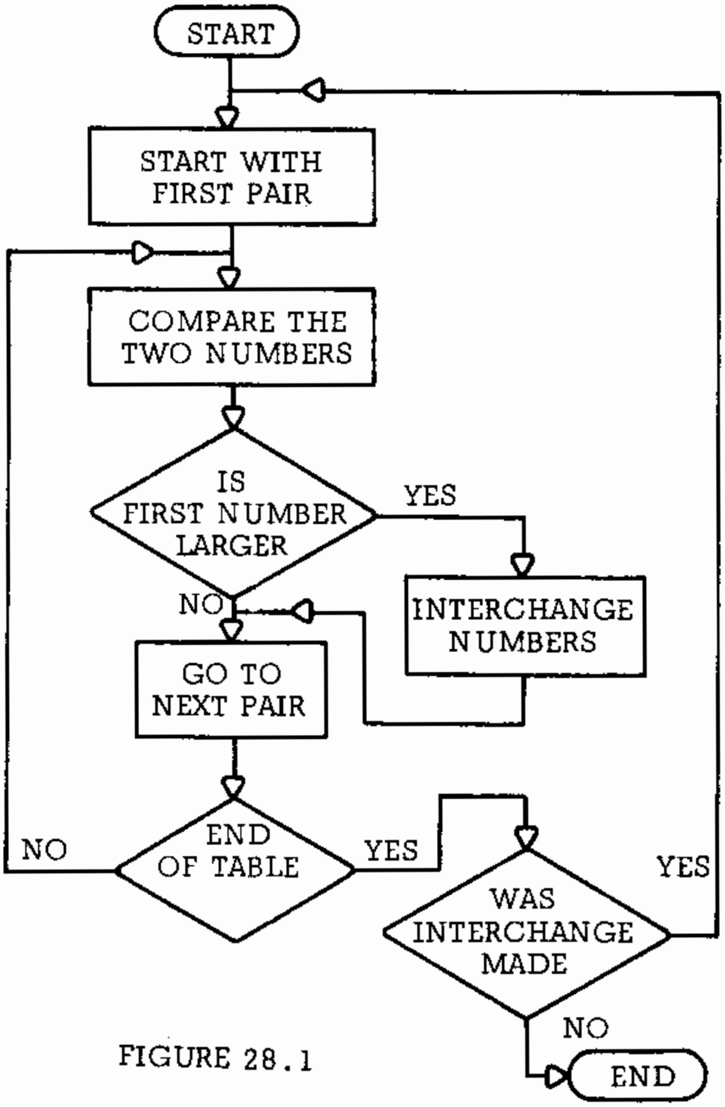


FIGURE 28.1

At the end of a pass we can look at whether this bit is 0 or 1 to see if we are done or if we need to make another pass.

We have revised and completed the original flowchart to include this, Figure 28.2.

When a flowchart has enough detail that it is obvious how to complete the solution, then no more should be added. Too much detail clutters it up. Sometimes secondary or auxiliary flowcharts are better. What constitutes enough detail depends upon who uses it. If someone besides the author is using it, it seldom is as clear as it was to the author.

A flowchart breaks a larger problem into smaller components and relates these parts to each other. With experience, people can understand larger concepts. Eventually their flowcharts may only say SORT when a sort is intended. Let's assume that you understand the component parts in Figure 28.2 and proceed from there.

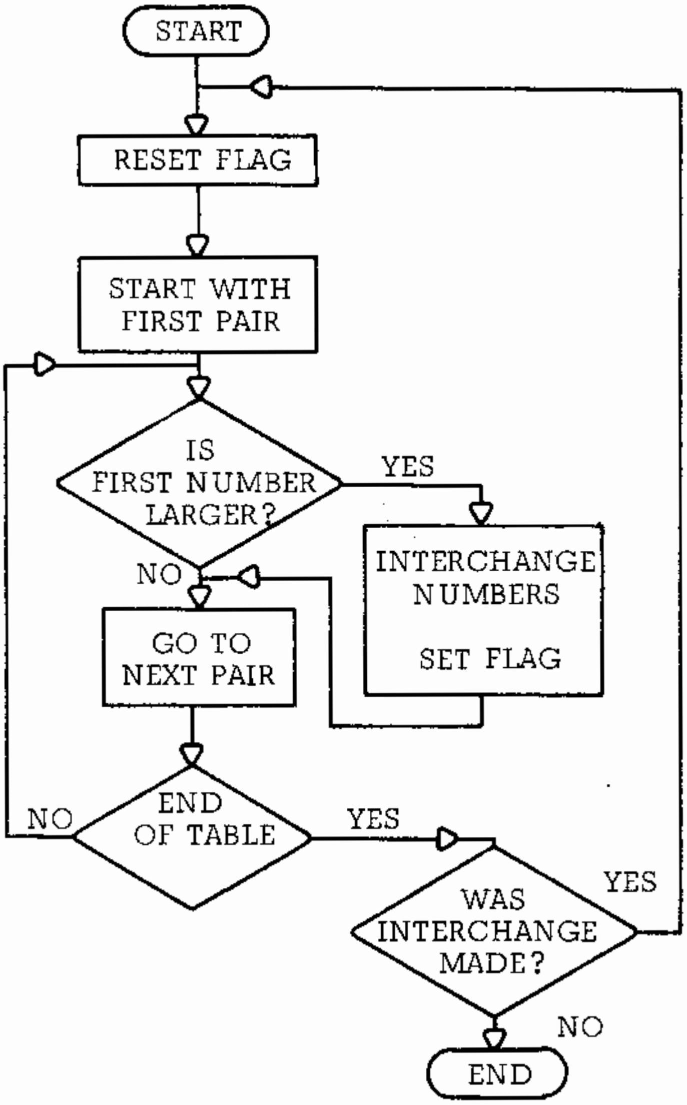


FIGURE 28.2

As you write a program it is a good habit to add comments with the instructions. These may tie the instructions in with the flowchart, or explain very generally what is being done, or explain a "tricky" point. On the next page we give the symbolic instructions for our sort program. From the flowchart and the comments you should be able to interpret the individual instructions.

A:0--
B:1--
X:2--

ADD:-0-
SUB:-1-
LOAD:-2-
STORE:-3-

Immed:--3
Mem:--4
Index:--6

JPD:-4-

A<0:--5
UNC:3-4
=0:--4

SET0:0-2
SET1:1-2
SKIP0:2-2

Loc	Data	Symbolic Address	Contents	Comments
		P	START	
		FLAG	---	
		START	LOAD X C=0	For first number
		RESET FLAG	SET 0 b7 FLAG	
		NEXT PAIR	LOAD A TABLE,X	First number
			ADD X C=1	To get second number
			SUB A TABLE,X	Subtract second number
			JPD A<0 OK	Jump if order is OK
			JPD A=0 OK	
			LOAD A TABLE,X	
			SUB X C=1	
			LOAD B TABLE,X	To interchange the two numbers
			STORE A TABLE,X	
			ADD X C=1	
			STORE B TABLE,X	
				Remember that interchange was made
			SET 1 b7 FLAG	
		OK	SUB X C=LAST-TABLE	Test for end of pass
			JPD X=0 END PASS	Jump if end of pass
			ADD X C=LAST-TABLE	Correction
			JPD UNC NEXT PAIR	Do next pair
				Test for interchanges, skip to ENDJOB if none
		END PASS	SKP 0 b7 FLAG	
			JPD UNC RESET FLAG	Do another pass
		END JOB	HALT	Done
		TABLE		Data to be sorted
		LAST		

We've carried the analysis far enough as a joint effort.

There is a bad point about the program. We can't complete the program until we know what address that LAST will be. We would like it to be more general. One technique is to store a number in the table (at the end) which is not valid as data. In our case this might be a number in the range 200 to 377. If we read a number of these characteristics, we would know that we are at the end of the table.

Here is a set of data to be sorted by the three methods that we discussed earlier

1456720379
0000321071
5620000324
2134728347

Would method 1 be a good method to sort these four numbers? _____

Why? ____

Would method 2 (when extended for the larger number) be a good method? ____

Do you think the characteristics of the data to be sorted would affect the efficiency of the method? ______________________________

Exercise 29

There's a four letter expression, GIGO, which stands for "Garbage In, Garbage Out". The inventor of the phrase meant to convey the idea that the output of a computer was no better than the data input. Actually, this need not always be the case. One of the functions of a program should be to test the validity of the input data. If the input data is bad, the computer should not use it.

In this Exercise, we will use a program which tests input data, has a subroutine, and has a new example of BCD to binary conversion. Our problem is to determine whether a date, which is our input, is valid or not. The input will be in three parts in this sequence:

2 BCD digits for the month
2 BCD digits for the day of the month
2 BCD digits (the last two) for the year.

Only 1 to 12 will be a valid month. The number of days per month depends on the month and whether it is a leap year or not. The year can be any two BCD digits.

Which years are leap years? These are the rules:

A. If the year is divisible by 400 (decimal) evenly (with no remainder), it is a leap year.
B. Years divisible by 100 evenly but not by 400 are not leap years.
C. Years divisible by 4 evenly but not by 100 are leap years.

Our program will check dates from 1601 to 1999. The dates 1600 and 2000 are leap years. The years 1700, 1800, and 1900 were not leap years. Our program will consider a year input of 00 not to be a leap year. During leap years, February has 29 days. Otherwise it has 28 days.

We give the complete program including a set of comments. We'll discuss some of the features of it starting now with the INPUT subroutine. This subroutine is used to get the month, day, and year inputs and to convert the entry to binary. The output display is used to indicate which of these three should be entered next. These output codes are

Month	00 100 000
Day	00 010 000
Year	00 001 000

The part of the main program which uses the subroutine will load the A register with the proper output code. The INPUT subroutine stores A in the output location 200. Coming into the subroutine, the B register may contain information which should be saved.

The BCD to binary conversion starts with LOOP within the INPUT subroutine. We follow a procedure similar to the one used earlier in Exercise 18 except that we detect too many subtractions from B by the carry (or borrow) which is stored in location 202. The correction for the A register is a combination of two things:

1. We have added 012 one too many times and this should be subtracted out.
2. In adding the units digit in B, we will be adding 1111UUUU since we oversubtracted on B. The correction for this is to add 020.

The combined correction from 1) and 2) is to add 006.

If the number, after conversion, has b7 as a 1, this would represent a number larger than +127 decimal. Since this would not be a valid month, day, or year in our convention, this would be a bad input. In this case we jump to the program for bad input and we do not make a normal return from the subroutine. If the input passes this first test, we restore the B register to its initial value and make a normal return from the subroutine.

Looking now at the main part of the program, it commences at START where the A register is loaded with the code that indicates a month should be entered. We then use the subroutine to get the input and convert it to binary. On the return from this subroutine the converted value is in the A register. If this number is zero, it is a bad input and it is rejected. If the month is larger than 12 decimal, it is also rejected.

```
START       LOAD   A     C = 040     Code for month

MONTH       JPM    UNC   INPUT       Get BCD input and
                                        convert to binary
            JPD    A=0   BAD         No zero month

            STORE  A     X           Keep month in X

            SUB    X     C = 015     (=13 decimal)

            JPD    X≥0   BAD         Month larger than 12

            ADD    X     C = 015     Restore

DAY         LOAD   A     C = 020     Code for day

            JPM    UNC   INPUT       Get BCD input and
                                        convert to binary
            JPD    A=0   BAD         No zero day

            STORE  A     B           Keep day in B

YEAR        LOAD   A     C = 010     Code for year

            JPM    UNC   INPUT       Get BCD input and
                                        convert to binary
            JPD    A=0   NOT LP YR

            SUB    A     C = 143     (=99 decimal)

            JPD    A>0   BAD         Larger than 99

            ADD    A     C = 143     Restore

            AND    A     C = 003     Leap year test

            JPD    A≠0   NOT LP YR

LP YR       SET1   b0    FEB         February has 29 days
                                        in leap years
NOT LP YR   SUB    B     JAN-1,X     Test for number of days
                                        per month
            JPD    B>0   BAD         Too many days for the
                                        month
            LOAD   A     C = 240     Codes for "good" and
                                        "month"
            JPD    UNC   OUT

BAD         LOAD   A     C = 041     Codes for "bad" and
                                        "month"
OUT         SET0   b0    FEB         Restore Feb to 28 days

            JPD    UNC   MONTH       Start over
```

```
INPUT        ---
             STORE   A     (200)        Output

             STORE   B     C =          Save B
BSAVE
             LOAD    A     C = 0        To clear input

             STORE   A     (377)        Clear input

             HALT                       Stop for input
             LOAD    B     (377)        Pick up input

LOOP         SUB     B     C = 020      Subtract 1 in 10's

             ADD     A     C = 012      Add 10 in 1's

             SKP1    b1    (202)        Skip if carry from B

             JPD     UNC   LOOP         Repeat

             ADD     A     C = 006      Correct A for overaddition
                                          and next addition
             ADD     A     B            Add units digit

             JPD     A<0   BAD          Number too large

             LOAD    B     BSAVE        Pick up B again

             JPI     UNC   INPUT        Return from subroutine



JAN          037                        = 31 decimal
FEB          034                        = 28 decimal
             037
             036                        = 30 decimal
             037
             036
             037
             037
             036
             037
             036
             037
```

To get the day input, we load A with the code for day and use the subroutine again. Again when we return, a zero value is not valid. The day is saved in the B register.

For the year input, we load A with the year code and jump to the INPUT subroutine. On returning, a zero value is valid and it is not a leap year (1900). Next we test for values greater than 99 decimal and take the exit to BAD if the year does exceed 99. If it is valid, we restore the original value and extract the two least significant bits. Since we are in the binary system, these two bits would be the remainder after division by 4. If they have the value 1, 2, or 3, it is not a leap year.

For the leap years we set the least significant bit of the number of days in February to 1. There is a table of 12 entries, one for each month, which is the maximum numbers of days for that month. Twenty-eight in octal is 034 and twenty-nine is 035 in octal.

To test whether we have more days than the month allows, we subtract the table entry from the number of days in B. The subtraction uses indexed addressing where X contains the month. An answer greater than zero is bad. For a good answer, we load A with 240 which is a combination of the codes for "good" and "month".

Good	10 000 000
Month	00 100 000

The program then goes to OUT.

For a bad input we load A with the codes for "bad" and "month".

Bad	00 000 001
Month	00 100 000

In the OUT routine, we restore February to 28 days and then jump back to MONTH. This takes us to the INPUT subroutine where A will be stored in the output lights.

Here are some variations on this problem for you to program.

1. In most countries the data sequence is day, month, year. There is more logic for doing it this way. Revise the program for this input sequence.

2. Either of these BCD inputs

```
0000 1010
or 0001 0000
```

yields an octal 012 (= 10 decimal). The first of these BCD inputs uses a non-valid BCD code. An even better input validity check would include tests for non-valid BCD codes even though the non-valid BCD codes lead to valid answers. Re-write the INPUT subroutine to eliminate non-valid BCD codes.

3. Expand and modify the program so that the program will indicate on which day of the week a valid date occurs. Do only 1900 to 1999. The output display could be in this form

```
Sunday      ●○ ○○○ ○○○
Monday      ○● ○○○ ○○○
  etc.
Saturday    ○○ ○○○ ○●○
Not valid   ○○ ○○○ ○○●
```

Hint: January 1, 1900, fell on a Monday.

A news item noted that a grandmother completed her high school work. The education department's computer rejected her application for a diploma because it wasn't programmed to handle forms for persons of her age. In processing her application, the computer subtracted her date of birth, 1887, from the current year and arrived at an age of minus 16. What can you tell about how the program was written.

Exercise 30

Computers are deterministic. We mean that what a computer does is well defined and, except for failures, it will always do the same thing when it starts from identical sets of conditions. Nothing is left to the "imagination" of the computer. It is a slave which does "its thing" without any variation.

Many problems can best be solved by the use of random numbers. Generating a random number in a deterministic machine is not easy. These techniques have been used:

1. Store previously generated random numbers in the memory. (One or more books have been published which consist of nothing but random numbers.)

2. Generate pseudo-random numbers in the computer. These numbers, though deterministic, will pass the most common tests of random numbers.

3. In our computer we will describe a third method which depends upon an influence external to the computer, namely the operator. This is a simple method and the "quality" of the random numbers is fair to good.

What is a random number? We shall try to approximate the following. A bowl contains 256 balls which are numbered 000 to 377 octally. A ball is selected by some impartial method and the number on it becomes the next random number. This ball is placed back into the bowl and they are mixed up again. Again a ball is selected. At any drawing of a ball, all balls are equally likely to be drawn regardless of the previous history of drawing.

We shall use a somewhat different technique. While the computer is idling, with nothing else to do, we'll have it add one to a number. When we push switch 7, it will stop counting and it will rotate the number three bits to the left. When we release the switch, the computer will resume adding one of the now rotated number. When we push switch 7 again it will rotate the number 3 bits to the left again. And again when we release switch 7, it will resume adding 1. The third time that we push switch 7, the computer will give us this number as the random number.

A random element is present because the time at which we push the switch varies. In addition, requiring three pushes to obtain one eight bit number increases the chances for variations. If we required only a one bit random number, one operator interaction would probably be enough.

We are going to give you a complete program and explain it along with some techniques for using the computer. At the end of the Exercise we will suggest some problems for you to examine.

In this program the computer never stops running. Usually we have the computer stop for input. Here, when the program has nothing else to do, it remains in a loop of three instructions:

```
COUNT      ADD   B     C = 1
           SKP1  b7    (377)
           JPD   UNC   COUNT
```

The B register holds the number which will become our random number. In this loop of three instructions, we add one to B each time through. If switch 7 is pushed then b7 of location 377 will be a 1, and the Skip will be taken which would cause the program to bypass the Jump instruction. When switch 7 has not been pushed, the Skip is not taken and the Jump instruction keeps the program in this loop.

Our requirement was for three pushes of the switch to generate one number. On leaving the loop above, we will subtract 1 in the X register to count the number of times we have pushed switch 7. When X is zero, we have pushed switch 7 three times and we take the number in B as our random number.

```
SUB    X     C = 1
JPD    X=0   HAVE NUMBER
ROTL   B3
```

If X is not yet zero, we rotated B three bits to the left.

When the random number has been generated, our program stores it in the output location. We then Jump to START to repeat the process. If some other use were to be made of the number, this would be a logical place to exit to that program. The return should be made to START to re-initialize the X register.

The next problem is to detect whether switch 7 has been released. We do it this way

```
CLEAR       SET 0   b7     (377)
            SKP 0   b7     (377)
            JPD     UNC    CLEAR
            JPD     UNC    COUNT
```

The program resets bit 7 in location 377. If the switch is still dep ssed, this bit reappears in location 377 and the Skip is not effective. We jump back to CLEAR and keep trying this. When switch 7 is eventually released, bit 7 will not reappear in location 377 and the Skip will be taken. This puts the program counter at the Jump instruction which in turn returns to COUNT.

We give the complete program below, all ready to use. Try it for a while. Do you think the numbers are random?

Loc	Data	Symbolic Address	Contents	Comments
000	---	A		
003	033	P	START	
200	000	OUTPUT		
004	103	COUNT	ADD B C=1	Idle loop
005	001			
006	372		SKP 1 b7 (377)	Exit with switch 7
007	377			
010	344		JPD UNC COUNT	
011	004			
012	213		SUB X C=1	X control number of times
013	001			
014	244		JPD X=0 HAVE NUM	Have random number now
015	027			
016	371		ROTL B3	Rotate
017	072	CLEAR	SET 0 b7 (377)	Test for switch 7 released
020	377			
021	272		SKP 0 b7 (377)	
022	377			
023	344		JPD UNC CLEAR	Not released
024	017			
025	344		JPD UNC COUNT	Released
026	004			
027	134	HAVE NUM	STORE B (200)	Display
030	200			
031	344		JPD UNC START	Dummy instruction
032	033			
033	223	START	LOAD X G=3	Initial value
034	003			
035	344		JPD UNC CLEAR	Go test for release of switch 7
036	017			

Here are a variety of problems for you.

Problem 1

Imagine that you are drawing balls numbered 0 to 255 (0 to 377 octal) which you return to the container after drawing each one. On the average, how many times would you expect to draw balls before you repeat a number previously drawn? You can use just the program we have given and keep track of the numbers you have drawn on paper. Preferably, you can write a program for this purpose.

Problem 2

Count the number of times that 0, 1, 2, -----, 7 appears in the units position. On very large samples each digit should appear "about" one-eighth of the time. Similar counts can be made for the "tens" and "hundreds" digit.

Problem 3

Following a 0 digit in the units position, count the number of times it is followed by a 0, a 1, a 2, etc. in the units position of the next number. This is partial check to determine whether two successive numbers are independent. If they are independent, knowing the first number does not help you predict the next number. The opposite of independent is dependent. If the second number is dependent to any degree on the first number, knowledge of the first number can help you predict the second number.

We suggested one type of test above but this can be extended in many obvious ways. For example, is the hundreds digit of a second number dependent, at least partially, upon the units digit of the first number?

Problem 4

Write a program to simulate the throw of dice or the deal of a deck of cards. Hint: Use the units octal digit but throw the number away if it is a 0 or 7. Use the "tens" digit for the other die.

Problem 5

Try writing your own random number generator through the use of an external variable such as we used.

Problem 6

Try writing a random number generator based only on internal operations of the computer.

*   *   *

A pseudo-random sequence of 255 bits can be generated by the following process:

1. Start with any number in a register other than 000.
2. Count the number of ones in bit positions 7, 5, 3, 2, 1, and 0.
3. Do a left shift of one bit on the number. This throws away bit 7 and creates a zero in bit 0.
4. If the number of 1's in step 2 is odd, put a 1 in bit 0, but if the number of 0's in step 2 is even, put a 0 in bit 0.

Each cycle of these four steps generates one new bit, say it is the bit which is entered into position 0. After 255 (decimal) cycles the pattern will repeat.

APPENDIX I

Oct	Dec	Oct	Dec	Oct	Dec	Oct	Dec	Oct	Dec	Oct	Dec	Oct	Dec	Oct	Dec
000	000	040	032	100	064	140	096	200	128	240	160	300	192	340	224
001	001	041	033	101	065	141	097	201	129	241	161	301	193	341	225
002	002	042	034	102	066	142	098	202	130	242	162	302	194	342	226
003	003	043	035	103	067	143	099	203	131	243	163	303	195	343	227
004	004	044	036	104	068	144	100	204	132	244	164	304	196	344	228
005	005	045	037	105	069	145	101	205	133	245	165	305	197	345	229
006	006	046	038	106	070	146	102	206	134	246	166	306	198	346	230
007	007	047	039	107	071	147	103	207	135	247	167	307	199	347	231
010	008	050	040	110	072	150	104	210	136	250	168	310	200	350	232
011	009	051	041	111	073	151	105	211	137	251	169	311	201	351	233
012	010	052	042	112	074	152	106	212	138	252	170	312	202	352	234
013	011	053	043	113	075	153	107	213	139	253	171	313	203	353	235
014	012	054	044	114	076	154	108	214	140	254	172	314	204	354	236
015	013	055	045	115	077	155	109	215	141	255	173	315	205	355	237
016	014	056	046	116	078	156	110	216	142	256	174	316	206	356	238
017	015	057	047	117	079	157	111	217	143	257	175	317	207	357	239
020	016	060	048	120	080	160	112	220	144	260	176	320	208	360	240
021	017	061	049	121	081	161	113	221	145	261	177	321	209	361	241
022	018	062	050	122	082	162	114	222	146	262	178	322	210	362	242
023	019	063	051	123	083	163	115	223	147	263	179	323	211	363	243
024	020	064	052	124	084	164	116	224	148	264	180	324	212	364	244
025	021	065	053	125	085	165	117	225	149	265	181	325	213	365	245
026	022	066	054	126	086	166	118	226	150	266	182	326	214	366	246
027	023	067	055	127	087	167	119	227	151	267	183	327	215	367	247
030	024	070	056	130	088	170	120	230	152	270	184	330	216	370	248
031	025	071	057	131	089	171	121	231	153	271	185	331	217	371	249
032	026	072	058	132	090	172	122	232	154	272	186	332	218	372	250
033	027	073	059	133	091	173	123	233	155	273	187	333	219	373	251
034	028	074	060	134	092	174	124	234	156	274	188	334	220	374	252
035	029	075	061	135	093	175	125	235	157	275	189	335	221	375	253
036	030	076	062	136	094	176	126	236	158	276	190	336	222	376	254
037	031	077	063	137	095	177	127	237	159	277	191	337	223	377	255

SUMMARY OF INSTRUCTION CODING

Instruction	First Byte Octal Digits		
	D--	-D-	--D

Add, Sub,
Load, Store
	A Reg = 0
	B Reg = 1
	X Reg = 2
		Add = 0
		Sub = 1
		Load = 2
		Store = 3
			Constant = 3
			Memory = 4
			Indirect = 5
			Indexed = 6
			Ind/Ind = 7

Or, And,
Lneg
	3	Or = 0
		(Noop = 1)
		And = 2
		Lneg = 3
			Constant = 3
			Memory = 4
			Indirect = 5
			Indexed = 6
			Ind/Ind = 7

Jumps	A Reg = 0
	B Reg = 1
	X Reg = 2
	Unc. = 3
		JPD = 4
		JPI = 5
		JMD = 6
		JMI = 7
			(≠ 0) = 3
			(= 0) = 4
			(< 0) = 5
			(≥ 0) = 6
			(> 0) = 7

Bit Test and Manipulation	Set to 0 = 0 Set to 1 = 1 Skip on 0 = 2 Skip on 1 = 3	Digit Value = Position	2
Shifts, Rotates (one byte only)	Right Shift = 0 Right Rotates = 1 Left Shift = 2 Left Rotate = 3	A = 0 B = 4 ----plus---- 1 place = 1 2 places = 2 3 places = 3 4 places = 0	1

Miscellaneous
(one byte only)
	Halt = 0 or 1
	Noop = 2 or 3
		Any value	0

THE
END