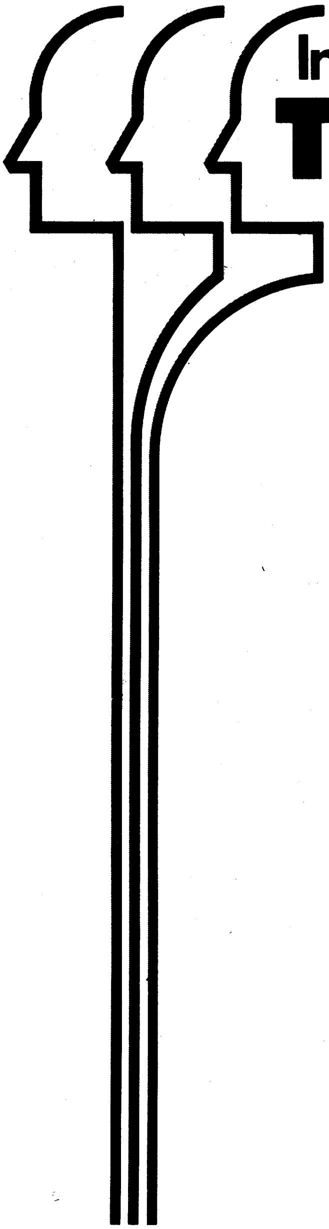


Introduction to The Instructor 50™ Desktop Computer

signetics

a subsidiary of **U.S. Philips Corporation**

Introduction to

The Instructor 50™

Desktop Computer

by

J. E. Doll

signetics

a subsidiary of U.S. Philips Corporation

Signetics reserves the right to make changes in the products contained in this book in order to improve design or performance and to supply the best possible products. Signetics also assumes no responsibility for the use of any circuits described herein, conveys no license under any patent or other right, and makes no representations that the circuits are free from patent infringement. Applications for any integrated circuits contained in this publication are for illustration purposes only and Signetics makes no representation or warranty that such applications will be suitable for the use specified without further testing or modification. Reproduction of any portion hereof without the prior written consent of Signetics is prohibited.

signetics

PREFACE

Computers today are a pervasive part of our society. No longer the exclusive domain of large corporations, computers are now available to nearly everyone at a price comparable to good stereo equipment. Specialized computers are already being used in home appliances, video games, and automobiles.

This manual is written especially for people just starting to explore the world of computers. It is designed to give you background information, an understanding of how computers work, and insights into their functions.

As you read the manual, you will be introduced to a lot of new terminology. You are urged to pay particluar attention to these new terms, called "buzz words," as they are defined. Each buzz word embodies a basic concept of computer or electronic technology. They will be underlined in the text as they appear. Through an understanding of these concepts, you will become familiar with your Instructor 50 or any other microcomputer. If you are not sure you understand a term or concept, it is wise to review its definition or explanation before you proceed.

Armed with your new knowledge you will be able to apply your Instructor 50 to a variety of functions. This variety is limited only by your own imagination. As a stand-alone unit, the Instructor 50 can be used as a home message center, a digital clock, a stop watch, or a game center. By connecting it to other external devices it can be used to play music on your stereo, provide video games on your T.V., keep your household records, and compute your financial records.

So, welcome to the world of computers! You have nothing to fear. Computers are no longer mysterious giants; they are as easy to operate as your automobile or microwave oven.

TABLE OF CONTENTS

PREFACE

CHAPTER I INTRODUCTION AND BACKGROUND

Historical Perspectives 1-1

Concept of Computer 1-2

CHAPTER II THE BUILDING BLOCKS OF COMPUTERS

Logic Signals 2-1

Combining Logic Signals 2-2

Flip/Flops: The Next Level of Complexity 2-9

Combining Flip/Flops into Counting Circuits 2-13

Combining Flip/Flops into Registers 2-16

CHAPTER III HOW COMPUTERS COUNT

I. The Binary Numbering System 3-1

II. Conversion from Binary to Decimal 3-5

III. Conversion from Decimal to Binary 3-6

IV. Binary Arithmetic 3-8

Addition 3-8

Subtraction 3-10

Multiplication and Division 3-13

V. The Hexadecimal Numbering System

A Convenient Compromise 3-14

CHAPTER IV INSIDE YOUR INSTRUCTOR 50

I. Basic Computer Organization 4-1

Memory System 4-2

The Arithmetic and Logic System 4-4

The Input/Output System 4-6

The Timing and Control System 4-8

Central Processor Unit 4-9

II. Computer Wares 4-9

What is a Program? 4-10

Moving Information Between Blocks 4-10
Software vs. Firmware 4-12
Organization of the 2650 CPU 4-13
The Program Status Word 4-16
III. How It All Works Together 4-17
Instruction is Fetched 4-17
Instruction is Executed 4-18
Example: A One-Instruction Program 4-19
Example: A Two-Instruction Program 4-21
IV. Types of Instructions (What Instructions Can Do) 4-22
Arithmetic Instructions 4-23
Input/Output (I/O) Instructions 4-23
Load and Store Instructions 4-23
Logical Instructions 4-23
Branch Instructions 4-25
Multi-Byte Instructions 4-27
Addressing Modes 4-29
V. Interpreting Instruction Descriptions 4-35
Making it Work 4-38

CHAPTER V PROGRAMMING TECHNIQUES
I. Organizing Your Program 5-1
II. Looping and Branching 5-9
III. Subroutines 5-13
Nested Subroutines 5-20
IV. Interrupts 5-22

CHAPTER VI 5-27

GLOSSARY

LIST OF TABLES AND FIGURES

FIGURES

1.1 THE 2650 MICROPROCESSOR 1-2

1.2 ABACUS 1-3

1.3 FUNDAMENTAL COMPUTING TASK 1-4

2.1 DOORBELL CIRCUIT 2-1

2.2 ALARM CIRCUIT 2-3

2.3 LOGIC SYMBOL FOR INVERTER 2-3

2.4 TRUTH TABLE FOR INVERTER 2-3

2.5 AND CIRCUIT 2-4

2.6 LOGIC SYMBOL FOR AND GATE 2-5

2.7 TRUTH TABLE FOR AND GATE 2-5

2.8 OR CIRCUIT 2-6

2.9 LOGIC SYMBOL FOR OR GATE 2-6

2.10 TRUTH TABLE FOR OR GATE 2-6

2.11 EXCLUSIVE-OR CIRCUIT 2-7

2.12 LOGIC SYMBOL FOR EXCLUSIVE-OR GATE 2-7

2.13 TRUTH TABLE FOR EXCLUSIVE-OR GATE 2-7

2.14 LOGIC SYMBOL AND TRUTH TABLE FOR NAND GATE 2-8

2.15 SYMBOL AND TRUTH TABLE FOR NOR GATE 2-9

2.16 LOGIC SYMBOL AND SCHEMATIC FOR D-TYPE FLIP/FLOP 2-9

2.17 FLIP/FLOP TIMING DIAGRAM 2-11

2.18 FLIP/FLOP TRUTH TABLE 2-12

2.19 DOORBELL MEMORY CIRCUIT 2-13

2.20 DOORBELL COUNTING CIRCUIT 2-14

2.21 OUTPUT STATES OF DOORBELL COUNTING CIRCUIT 2-15

2.22 AN 8-BIT REGISTER 2-17

4.1 BASIC COMPUTER ORGANIZATION 4-1

4.2 DIAGRAM OF 2650 MEMORY 4-2

4.3 ARITHMETIC AND LOGIC SYSTEM BLOCK DIAGRAM 4-4

4.4 ALU LOGICAL OPERATIONS 4-5

4.5 INPUT/OUTPUT SYSTEM 4-7

4.6 COMPUTER COMMUNICATIONS PATHS 4-10

4.7 2650 CPU REGISTERS 4-14

4.8 EXAMPLES OF LOGICAL INSTRUCTIONS 4-24

4.9 OPERATION OF BRANCH INSTRUCTION 4-26

4.10 OPERATION OF CONDITIONAL BRANCH INSTRUCTION 4-27

4.11 FORMAT OF BCTA INSTRUCTION 4-28

4.12 FORMAT OF ADDZ INSTRUCTION--REGISTER ADDRESSING 4-30

4.13 FORMAT OF ADDI INSTRUCTION--IMMEDIATE ADDRESSING . . . 4-30

4.14 FORMAT OF ADDR INSTRUCTION--RELATIVE ADDRESSING 4-32

4.15 FORMAT OF ADDA INSTRUCTION--ABSOLUTE
ADDRESSING WITH INDEXING 4-33

5.1 A SIMPLE FLOW CHART 5-2

5.2 MNEMONIC LISTING ON CODING FORM 5-3

5.3 DEFINING LABELS 5-5

5.4 CODING THE INSTRUCTIONS 5-7

5.5 A CONDITIONAL LOOP 5-10

5.6 FLOWCHART TO FLASH LIGHT 5-10

5.7 A ONE SECOND TIME DELAY 5-12

5.8 PROGRAM WITH SUBROUTINE 5-15

5.9 TIME DELAY SUBROUTINE 5-17

5.10 FLOW CHART TO FLASH LED'S 5-18

5.11 PROGRAM TO FLASH LED'S 5-19

5.12 NESTED SUBROUTINES 5-20

5.13 PROGRAM USING INTERRUPT 5-23

5.14 CODE FOR PROGRAM USING INTERRUPT 5-24

5.15 CODE FOR PROGRAM USING INDIRECT INTERRUPT 5-26

TABLES

3.1 FLIP/FLOP COUNTING PATTERN 3-2

3.2 DECIMAL AND BINARY EQUIVALENTS 3-3

3.3 BINARY EQUIVALENTS OF POWERS OF 2 3-6

1. INTRODUCTION AND BACKGROUND

Historical Perspectives

In the early 1950s the very first commercial computers began to appear on the scene. At that time a computer was built of vacuum tubes, filled a one-story building, and required a specialized staff to maintain and operate it. It cost between half a million and a million dollars, and those were 1950 dollars. Market researchers for large corporations analyzed the new device and most of them concluded that due to its high cost of production and maintenance that perhaps 50 to 200 computers could be built and sold. After that they predicted the market would be flooded.

In spite of advice to the contrary, several corporations, notably IBM Corporation, added the computer to their product line. By the late 1950s and early 1960s many large corporations were using computers for their data processing operations. What is astounding is not the shortsightedness of the market researchers, but the technological advances which have occurred within the computer industry in the past few decades. With the invention of transistors, computers could be built for a small fraction of the size and cost of their vacuum tube counterparts. Soon it was discovered that thousands of transistors could be integrated onto a single tiny chip of silicon semiconducting material. Again, with the advent of these integrated circuits, the size, power, and cost requirements for a computer decreased dramatically. Funding provided for space program research during the 1960s provided a strong impetus for many of these technological breakthroughs. And as the size and cost of computer power was decreasing it became possible to produce more and more powerful computers, with more sophisticated functions. The 2650 microprocessor, the *integrated circuit* at the heart of your Instructor 50, provides more sophisticated computational power than early commercial computers of the 1950s.

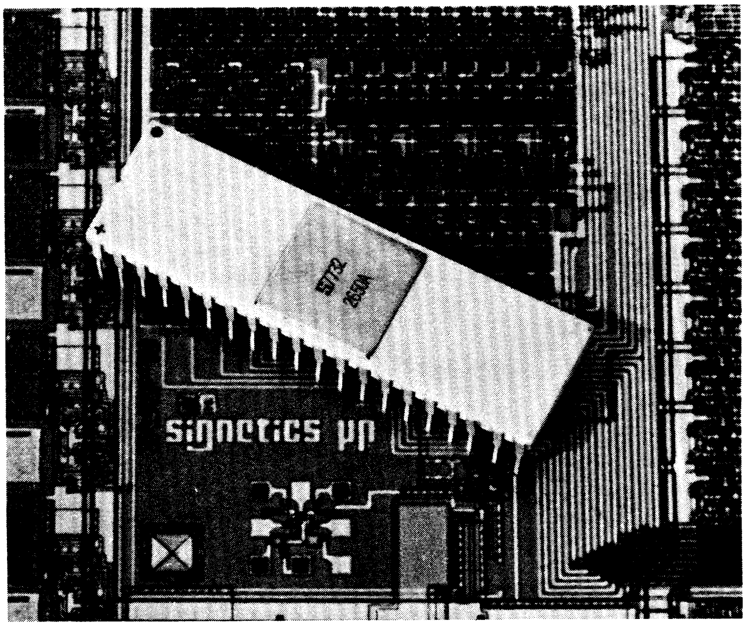


Figure 1.1 THE 2650 MICROPROCESSOR

While the size and cost of computing power was diminishing, sophistication available in commercial machines was increasing. But the basic principles of construction have remained essentially the same. These principles are used in virtually all computers being produced today, and are the basis for your Instructor 50. It is these principles which this manual will introduce and explain.

Concept of Computer

One of the earliest computing devices was the abacus. An abacus is an arrangement of beads which are slid along rods to form different patterns. Various positions of the beads represent different numbers. By manipulating the beads the operator of the abacus can perform mathematical problems quickly and accurately. Modern day digital computers are descendents of the abacus. The word digital is often associated with modern computers, implying that they operate using digits or numerical values. In reality, of course, what is manipulated is not numerical values but electronic signals. We simply assign numerical value to the signals that are being manipulated just as the operator of the abacus assigns numerical value to the position of his beads.

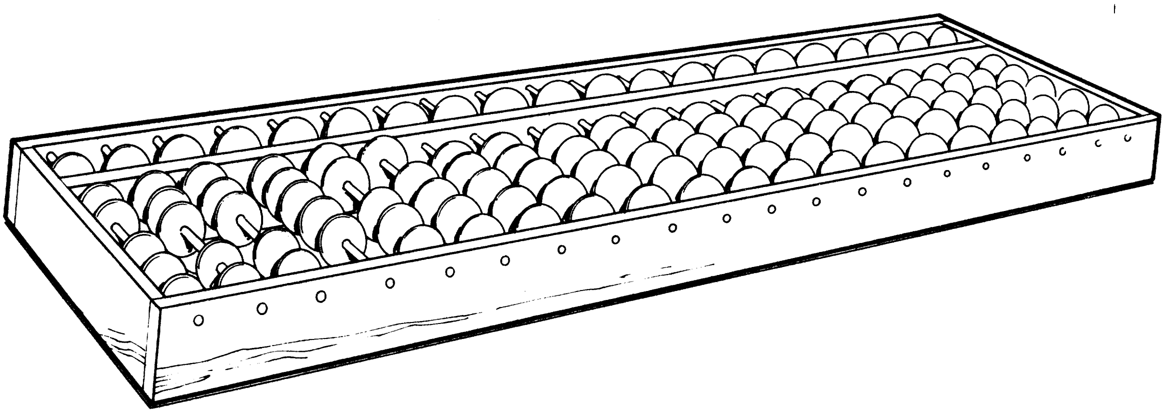


Figure 1.2 ABACUS

With the computer, however, the process of sequentially manipulating the beads (substitute electronic signals) is automated. A human operator is not required. A computer, then, is a *device for manipulating numbers*. Unlike the abacus operator who can decide himself what problems to solve and how to solve them, the computer cannot make these decisions. These decisions are the job of the computer programmer.

First the *programmer* reduces the task to a sequence of manipulations which can be performed by the computer. The computer then carries out these manipulations, or operations on numbers, in the sequence specified by the *programmer*. This sequence of operations or manipulations is called a computer program. The computer is designed to carry out these operations automatically, one at a time, similarly to the way in which the operator of the abacus solves his problem by manipulating beads. As you can see, the computer itself is not possessed of judgment any more than is the abacus. Any intelligence that a computer may appear to possess comes from its human *programmer*. Once the *program* has been specified the computer can carry out the tasks of the *program* at a rapid pace.

The fundamental computing task is diagrammed in Figure 1.3.

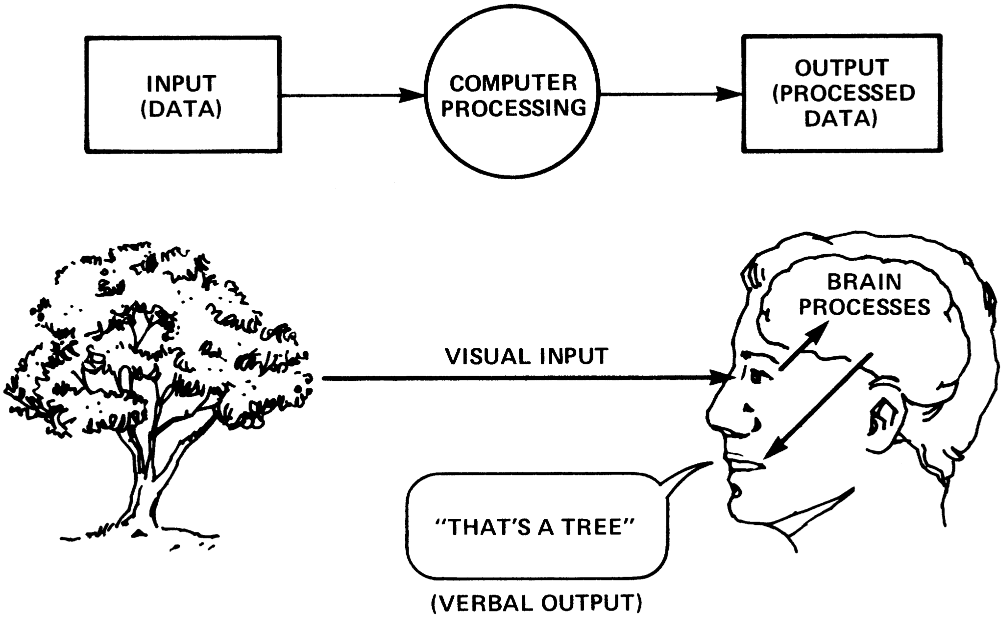


Figure 1.3 FUNDAMENTAL COMPUTING TASK

First, information is input to the computer representing the data to be manipulated. This data is presented to the computer in the form of electronic signals in a very specific format. For instance, if the task of the computer is to add together two numbers, electrical signals representing the two numbers will be *input* to the computer.

The computer program, then, directs the computer to perform the manipulations necessary to add the two numbers together and to come up with an answer. This is called the processing of the data. Finally, the computer presents another set of electrical signals to the outside world. These signals represent the numerical or logical result of the computation and are called output. These three activities, *input, processing,* and *output* are fundamental to any computing activity. We humans also engage in *input, processing,* and *output* activities constantly and on a much more complex scale.

For example, imagine that you are taking a stroll through the woods. You are constantly receiving *input*. You see the green foliage, smell the damp ferns, hear the wind rustling through the tree branches. In an instantaneous process you interpret that you are walking in the woods. You match this *input* to the memory of another stroll through the woods or perhaps to something you have read or heard about. The process is so rapid and subtle that you may not even be aware it is occurring. Suddenly a companion asks, "Isn't it a beautiful day?" Your brain rapidly recognizes the verbal *input* as a question requiring an affirmative or negative response. The brain then evaluates (*processes*) all sensory data important to defining what a beautiful day is. "It certainly is," you respond with verbal *output*. You may also nod your head in an affirmative body gesture. Obviously the range of electronic signals that a computer can accept as *input* and the signals that it can generate as *output* are feeble indeed in comparison to the variety of *inputs* and *outputs* that are available to human beings.

There is no point in writing a computer program to solve a problem that occurs only occasionally and is easy to solve. Problems uniquely suited to solution by a computer program are those which are either tedious and repetitive, or those which appear often enough to justify the effort of programming. Some examples are keeping track of a checking account or a charge account balance.

For these problems to be solved by a computer, they must first be reduced to *digital* or numerical form. Fortunately, a number of ingenious methods have been devised to accomplish this task. These methods have evolved over the years from an analysis of fundamental logical and mathematical processes. An understanding of these basic concepts provides the foundation for designing the building blocks of computers.

2. THE BUILDING BLOCKS OF COMPUTERS

Logic Signals

The logic signal is the most fundamental concept used in building modern computers. It is a signal which can exist in only two conditions, and is used to represent some meaningful concept. For instance, the two states of a *logic signal* can represent the concepts "yes" and "no"; "on" and "off"; "true" and "false"; "present" and "absent"; "now" and "not now"; "active" and "not active."

A number of physical devices can be used to represent these abstract concepts. For Paul Revere, it was one lantern if by land, two if by sea. Mr. Revere may not have recognized the fact, but he was transmitting a simple *logic signal* which had only two possible conditions. The meanings of those two conditions were agreed upon in advance by Mr. Revere and his fellow revolutionaries.

A more familiar everyday example of a *logic signal* is provided by your doorbell. Consider Figure 2.1, a simple doorbell circuit. When a finger is applied to the doorbell pushbutton an electrical contact is closed. This allows electrical current to flow from the battery through the switch, through the bell and back into the battery. The sound of the bell indicates that there is someone at the door.

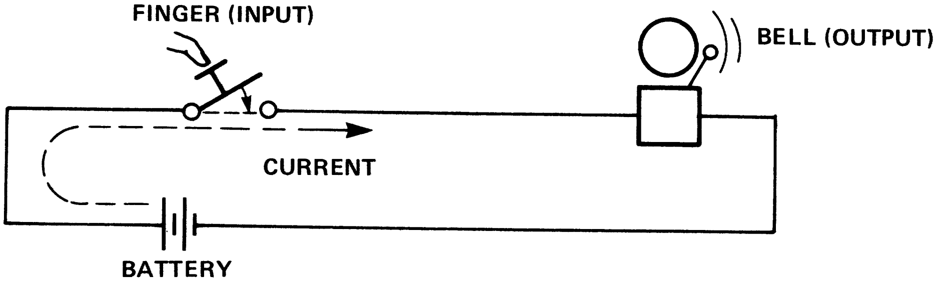


Figure 2.1 DOORBELL CIRCUIT

In this case, electrical current has been used to transmit a *logic signal*. If the current is on or flowing someone is pushing the doorbell. The *input* is provided by the finger pushing the doorbell. After pushing the button the form of the *logic signal* is changed into an electrical current. Finally, it is changed from current into sound by the bell and transmitted through the air to your ear. While the physical representation of the signal changes forms several times during its transmission, the meaning of the signal does not change. It is always present or absent, true or false, and always means that someone is pushing the button or someone isn't.

Computers generally use two electrical voltage levels on wires to transmit *logic signals* from one place to another. Inside your Instructor 50, a *logic signal* is represented by presence or absence of an electrical voltage on a wire. If a computer could only transmit *logic signals* from one place to another, it would be hardly more remarkable than your doorbell. A computer, however, has the ability to manipulate *logic signals* as well.

Combining Logic Signals

The simplest manipulation that can be performed on a logic signal is to invert it. That is, if the signal is true, make it false; if it is false, make it true. This can be very useful at times.

Let us suppose, for example, that a secret society has been formed to meet and discuss certain sensitive subjects. The first meeting takes place at a member's home and, since the meeting must be secret, they elect to post a guard at the door. To make sure the guard hasn't fallen asleep he is instructed to keep his finger on the doorbell. It soon becomes evident that the business of the meeting cannot be conducted over the constant din of the doorbell. The meeting cannot continue.

This unfortunate circumstance could have been avoided had the society been familiar with the *inversion* of logic signals. Consider, for example, the alarm circuit of Figure 2.2. This is similar to the doorbell circuit of Figure 2.1 except that the pushbutton is constructed to turn the current off when a finger is applied to the button. Thus, so long as the guard pushes the button (input), the alarm (output) does not sound. The pushbutton has been wired to *invert* the logic signal.

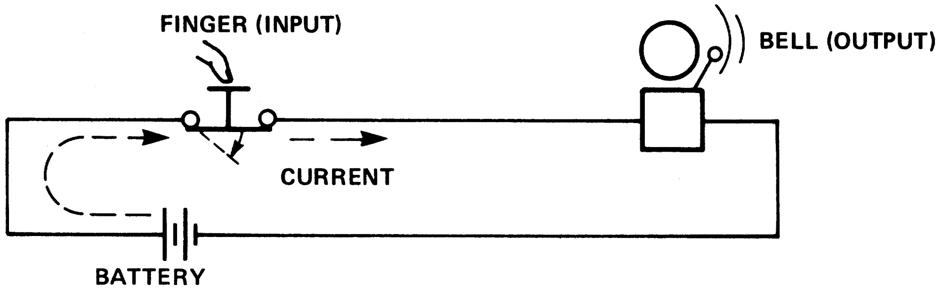


Figure 2.2 ALARM CIRCUIT

Inversion is very common in computer circuits. And the circuit that performs this function, the inverter, has a special symbol of its own, shown in Figure 2.3.

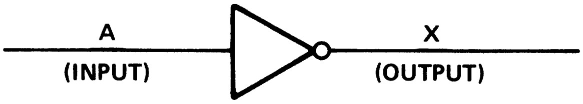


Figure 2.3 LOGIC SYMBOL FOR INVERTER

A logic signal which we have labeled A is applied to the *input* side of this circuit. The *output* side of the circuit, which we have labeled X is always the opposite logic condition from the input. This is shown in the truth table for the *inverter* circuit of Figure 2.4.

INPUT	OUTPUT
A	X
0	1
1	0

Figure 2.4 TRUTH TABLE FOR INVERTER

Recall that a logic signal can have only two possible states. It is common practice to label these two states Ø and 1. Thus, in our *truth table* under the input A we show that when A is a Ø the output X is a 1. And when the input A is a 1, the output X is a Ø.

It would be well at this point to explain that it is common practice among programmers to represent zeroes with a slash through them to distinguish them from upper case O's. We will do this throughout this manual to avoid confusion.

The information in this truth table can also be expressed in the form of an equation:

$$X = \bar{A}$$

The little bar above the A indicates *inversion*. This equation may be read "X equals A not," or "X equals not A" or "X equals A bar."

It is also possible to combine two or more *logic signals* to produce a logical result. Our simple doorbell example, above, showed only one pushbutton. Consider the two pushbutton circuit of Figure 2.5. This is called an AND circuit because both button A *AND* B must be pushed simultaneously to produce an output. If button A or button B is not pushed, the bell is silent.

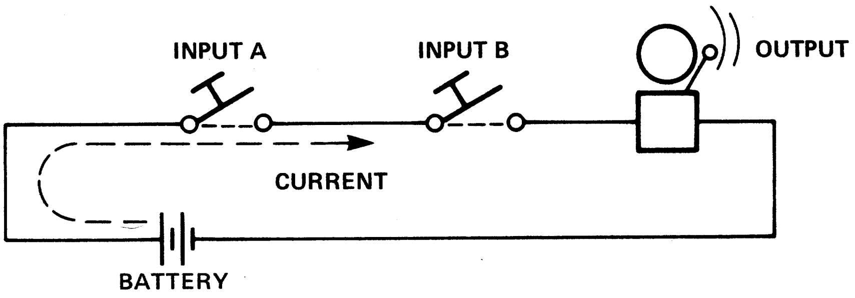


Figure 2.5 AND CIRCUIT

The logic symbol for the AND circuit, or AND gate as it is sometimes called, is shown in Figure 2.6.

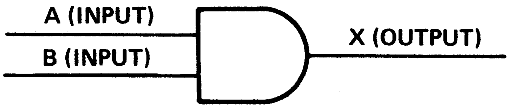


Figure 2.6 LOGIC SYMBOL FOR AND GATE

Note that the logic symbol has two *inputs* and one *output*. The function of the *AND gate* can be described by the truth table of Figure 2.7. The truth table shows that when input A is a Ø *AND* input B is a Ø the output, X, is a Ø. When input A is a 1 but B is a Ø the output X is still a Ø. When input B is a 1 but input A is a Ø, the output X is a Ø. Only in the case in which both inputs A *AND* B are 1s is the output, X, a 1.

INPUTS		OUTPUT
B	A	X
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

Figure 2.7 TRUTH TABLE FOR AND GATE

This can also be expressed in equation form:

$$X = A \cdot B$$

read "X equals A *AND* B." Thus we see that *AND* is a simple logic operation which combines two *inputs* to form one *output*.

Similarly, it is possible to combine two logic signals in an OR function. Consider the pushbutton circuit of Figure 2.8. When pushbutton A is pushed, the bell is connected to the battery through button A. When pushbutton B is depressed the bell is connected to the battery through button B. Pushing either button A *OR* button B will cause the bell to ring.

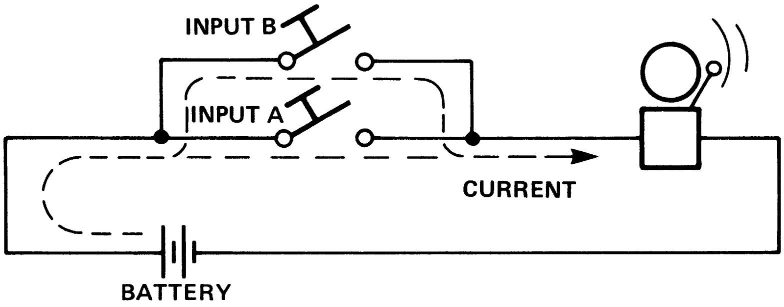


Figure 2.8 OR CIRCUIT

The logic symbol for an *OR gate* is shown in Figure 2.9. Its operation is described by the truth table of Figure 2.10. This truth table shows that when both inputs A and B are Øs the output, X, is a Ø. When input A is a 1 but input B is a Ø the output, X, is a 1, or if input B is a 1 but input A is a Ø the output, X, is a 1. If either A *OR* B (or both) is a 1 the output is a 1. This can also be expressed in equation form by writing:

$$X = A + B$$

read "X equals A *OR* B."

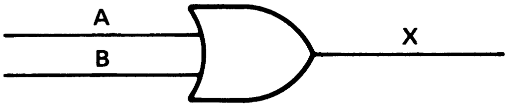


Figure 2.9 LOGIC SYMBOL FOR OR GATE

INPUTS		OUTPUT
B	A	X
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

Figure 2.10 TRUTH TABLE FOR OR GATE

The final logic function with which we will concern ourselves is the Exclusive-OR function. Using our pushbutton analogy, it is shown by the circuit of Figure 2.11.

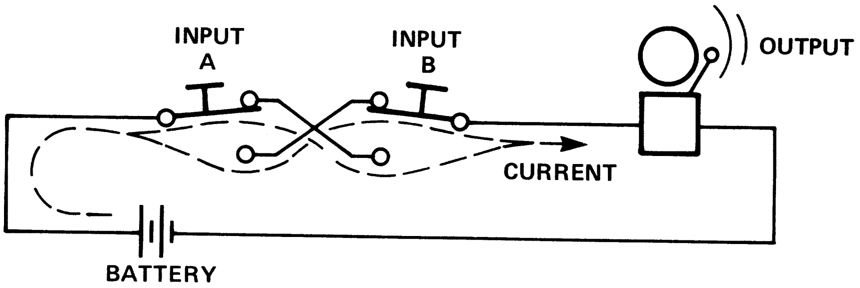


Figure 2.11 EXCLUSIVE-OR CIRCUIT

By following the current paths in Figure 2.11 you can see that if button A is pushed the bell will ring. Or if button B is pushed the bell will ring. But if both buttons A and B are pushed simultaneously the current path will be broken and the bell will not ring. The logic symbol for the *Exclusive-OR gate* is shown in Figure 2.12.

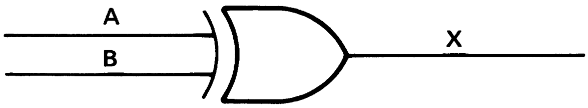


Figure 2.12 LOGIC SYMBOL FOR EXCLUSIVE-OR GATE

Its function is described by the truth table of Figure 2.13.

INPUTS		OUTPUT
B	A	X
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	Ø

Figure 2.13 TRUTH TABLE FOR EXCLUSIVE-OR GATE

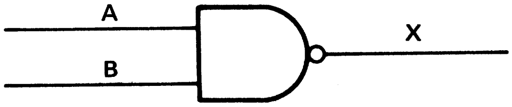
Note that if inputs A and B are both Ø the output, X, is Ø. And if inputs A and B are both 1s, the output, X, is Ø. Only if input A is a 1 and input B is a Ø, or if input B is a 1 and input A is a Ø, is the output X, a 1. The inputs A and B *must be opposite* in their logic values for the output to be a logic 1. This can be expressed by the equation:

$$X = A \oplus B$$

read "X equals A *Exclusive-OR* B."

These four logic gates, the *inverter*, the *AND gate*, the *OR gate*, and the *Exclusive-OR gate* are the fundamental building blocks from which all digital computers are built. You will see similar logic symbols used in great abundance in the Instructor 50 schematic diagram shown in the back of your Instructor 50 User's Guide.

A *schematic diagram* is simply a diagram which shows the scheme for connecting various simple circuits to form a larger, more complicated circuit. In drawing these diagrams it is common to combine the functions of an inverter circuit with one of the other logic gates. For instance, the output of an AND gate may be inverted to produce the opposite of the AND function. This might be called the not-AND function, but is more commonly called the NAND function. The logic diagram and truth table for the *NAND* function are shown in Figure 2.14.

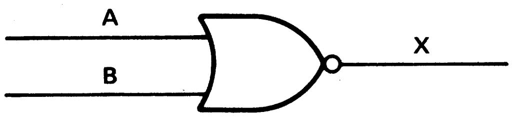


INPUTS		OUTPUT
B	A	X
Ø	Ø	1
Ø	1	1
1	Ø	1
1	1	Ø

$$X = \overline{A \cdot B}$$

Figure 2.14 LOGIC SYMBOL AND TRUTH TABLE FOR NAND GATE

The little circle at the output of the symbol for the *NAND gate* denotes the inversion function. Similarly, the symbol for a NOR gate and its corresponding truth table are shown in Table 2.15. Again, the little circle denotes inversion. Note that the *output* column of these truth tables is exactly opposite the truth tables for their *AND* and *OR* counterparts.



INPUT		OUTPUT
B	A	X
Ø	Ø	1
Ø	1	Ø
1	Ø	Ø
1	1	Ø

$X = \overline{A+B}$

Figure 2.15 SYMBOL AND TRUTH TABLE FOR NOR GATE

Flip/Flops: The Next Level of Complexity

By connecting these simple *logic gates* together in certain ways, it is possible to build circuits that do a variety of useful things. Several common connections have already been designed, and rather than showing the individual logic gates on the diagrams for these common circuits, it is customary to show them simply as blocks on a schematic diagram.

One common connection of gates often used in computers is called a flip/flop. A logic symbol for a D-type flip/flop is shown in Figure 2.16, along with a schematic diagram for the internal connection of its gates. The schematic diagram has been included only to convince you that this circuit can indeed be built by connecting simple logic gates and we will not analyze it in detail. Instead, we will describe the behavior of this circuit in terms of its *inputs* and *outputs*.

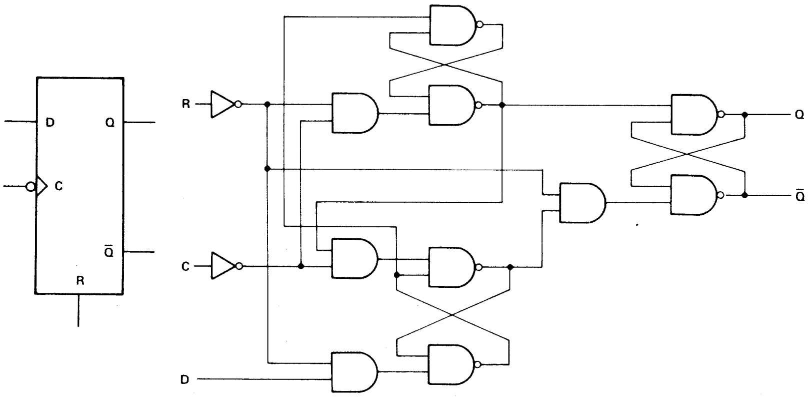


Figure 2.16 LOGIC SYMBOL AND SCHEMATIC FOR D-TYPE FLIP/FLOP

This circuit has two *outputs* labeled Q and $\bar{Q}$, and three *inputs* labeled D, C, and R. Of course, the logic values that appear at the *outputs* of this circuit depend on the logic values that are presented to the *inputs* of the circuit. The *outputs* are labeled Q and $\bar{Q}$ because they always assume opposite logic values. If Q is a 1, then $\bar{Q}$ is a Ø. If Q is a Ø, then $\bar{Q}$ is a 1.

The input labeled C is a timing or clock signal. A transition on this signal from a logic 1 to a logic Ø determines when the output is allowed to change state. When a transition occurs on the *clock (C) input* from a 1 to a Ø, the Q output will assume the logic value which is presented to the D, or DATA input. For instance, if the *D input* sees a logic 1, and the *clock input* makes the transition from a logic 1 to a logic Ø, then logic 1 will appear on the Q output and logic Ø on the $\bar{Q}$ output. This condition will remain at the outputs until two things occur: First, a logic Ø must appear on the *D input*. Concurrently, another transition from a logic 1 to a logic Ø must occur on *input C*. Another way of looking at this is that the input appearing on *D* is *delayed* from reaching the output until the transition (1-to-Ø) occurs on *C*.

Input R is the Reset function for the flip/flop. All of the foregoing is true so long as *input R* is at a logic Ø. Whenever *input R* is a logic 1, the Q output becomes a Ø and the $\bar{Q}$ output becomes a 1, independent of what happens at the *C and D inputs*.

The behavior of this circuit may be further illustrated by the timing diagram of Figure 2.17. Let us assume that the *clock input* is fluctuating between a logic 1 and logic Ø, as shown in the timing diagram. (Incidentally, in your Instructor 50 a logic 1 is represented by a voltage level of +5 volts, and a logic Ø by a voltage level of about Ø volts. This is a common practice with logic circuits, so that engineers sometimes refer to a logic 1 as a +5 volts, and a logic Ø as Ø volts, or to a logic 1 as a high level and a logic Ø as a low level.) The critical instant is that in which the *clock signal* makes the transition from a logic 1 to a logic Ø.

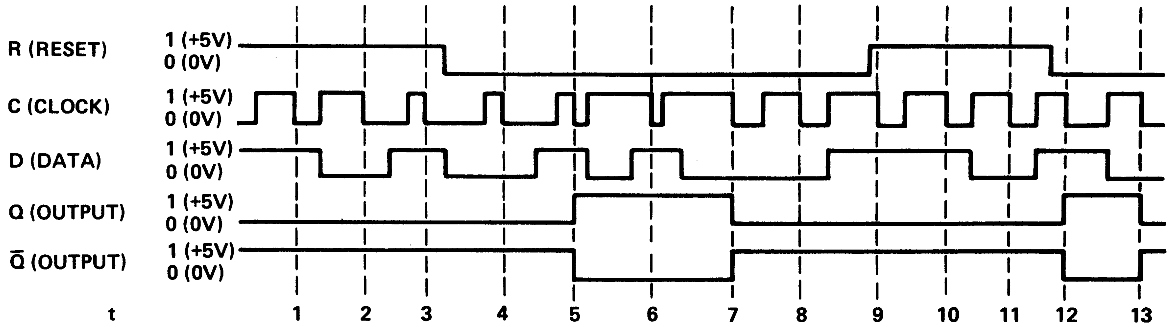


Figure 2.17 FLIP/FLOP TIMING DIAGRAM

We've drawn vertical dotted lines on the diagram to indicate these instants, and have labeled them as clock times 1 through 13. During clock times 1, 2, and 3 the *Reset* signal is a high, or logic 1. Therefore, the Q output is a low. This is independent of any activity on the D input. Since the $\bar{Q}$ output must always be opposite the Q output, the $\bar{Q}$ output is a high at this time.

At the occurrence of clock time 4 the *Reset* signal is a Ø but the D input is also a Ø, so the Q output remains a Ø, and the $\bar{Q}$ output remains a 1. At clock time 5 the D input is a logic 1, so the Q output becomes a logic 1 to match the D input, and the $\bar{Q}$ output becomes a logic Ø. At clock time 6 the D input is again a logic 1 so the Q output remains a logic 1 and the $\bar{Q}$ output remains a logic Ø. At clock time 7, the D input is a logic Ø so the Q output goes to a logic Ø, and $\bar{Q}$ becomes a logic 1.

At clock time 8, the D input is still a logic Ø, so the Q output remains a logic Ø and the $\bar{Q}$ output remains a logic 1. At clock time 9 the D input is a logic 1 but the *Reset* signal has also become a logic 1. Therefore, the Q output remains a logic Ø and the $\bar{Q}$ output remains a logic 1.

This is also true for clock time 10, and clock time 11. At clock time 12 the *Reset* signal has fallen back to Ø and the D input is now a 1, so the Q output becomes a 1 and the $\bar{Q}$ output a Ø. At clock time 13 the D input has fallen to a Ø, so the Q output becomes a Ø and the $\bar{Q}$ output becomes a 1.

This behavior can be summarized the the truth table of Figure 2.18. On the input side of the diagram we show the conditions of the input at clock time t_n, that is, at a particular time at which the clock signal makes a transition from a 1 to a Ø. The output side shows how the outputs will look at clock time $t_n + 1$ (when the next clock transition occurs). The first line of the truth table shows that when input R or Reset is a 1, output Q is a Ø and output $\bar{Q}$ is a 1. The "X" indicates that it doesn't matter what the condition of input D is. If input R is a Ø and input D is a Ø, then after the clock transition occurs, output Q will become a Ø and output $\bar{Q}$ will become a 1. Also if input R is Ø, and input D is a 1, after the clock transition output Q will become a 1 and output $\bar{Q}$ will become a Ø.

t_n		$t_n + 1$	
INPUT R	INPUT D	OUTPUT Q	OUTPUT $\bar{Q}$
1	X	Ø	1
Ø	Ø	Ø	1
Ø	1	1	Ø

Figure 2.18 FLIP/FLOP TRUTH TABLE

Flip/flops are very important in computers because of their ability to "remember" the logic value on their D input until the occurrence of a subsequent clock transition. It is said that a *flip/flop* can store one bit of information. The word "bit" is a contraction of the words "BInary digiT." Through the use of logic signals we have the ability to transmit *bits* from one place to another within the computer. Through the use of *logic gates* we have the ability to combine two or more logic *bits* to form a resultant *bit*. And with the *flip/flop* we have the ability to store the logic *bit* for subsequent examination.

For example, let's say that you would like to know if anyone rings your doorbell while you're away from home. This could be accomplished by hooking your doorbell into the circuit shown in Figure 2.19. Before you leave the house you momentarily push the Reset button connecting the Reset input to +5 volts or logic 1, and forcing the Q output to a logic Ø or Ø volts. No indication will be present on the meter.

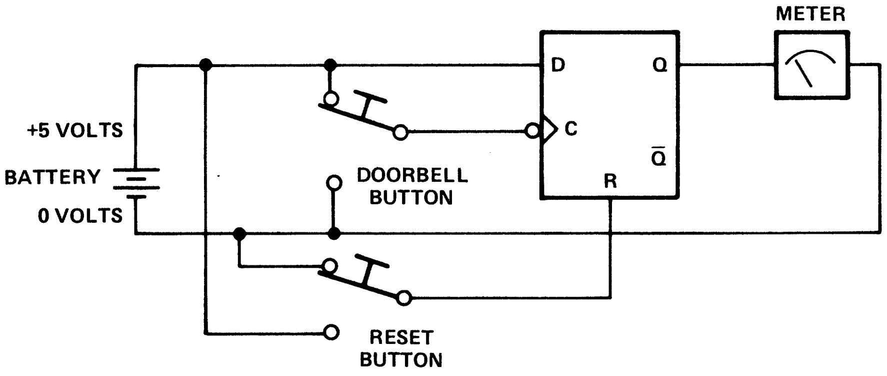


Figure 2.19 DOORBELL MEMORY CIRCUIT

This condition will remain until someone presses the doorbell button. At that time, the clock input will be momentarily disconnected from the +5 volts or logic 1, and will be connected to Ø volts or logic Ø. This will cause the logic 1 wired to input D to appear at the Q output. When the button is released the Q output will retain the logic 1. When you return home you may examine the meter to determine if anyone has pushed your doorbell button. If it reads about 5 volts, someone has. The fact that the button has been pushed has been "remembered" by the *flip/flop* as one *bit* of information.

Combining Flip/Flops into Counting Circuits

Let's say that you're not content with knowing simply that your doorbell has been pushed. You'd also like to know how many times it was pushed while you were gone. To accomplish this, you might connect the circuit of Figure 2.20. Again, before you leave the house, you push the Reset button momentarily connecting the Reset inputs to +5 volts or a logic 1 and setting all of the Q outputs ($Q_\emptyset$, Q_1, and Q_2) to a logic Ø. Because the $\bar{Q}$ outputs are opposite the Q outputs in logic value, all of the D inputs are now connected to a logic 1.

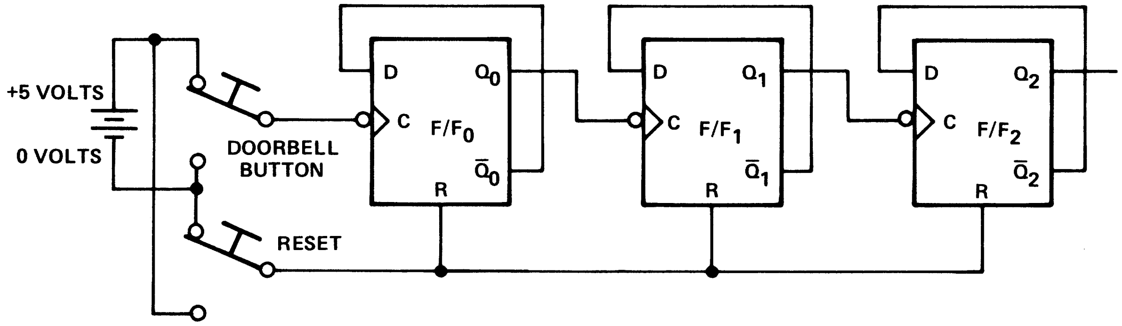


Figure 2.20 DOORBELL COUNTING CIRCUIT

When the doorbell button is pushed the clock input of flip/flop Ø makes a momentary transition from a logic 1 to a logic Ø. This causes the 1 which appears on the D input to be transferred to the output $Q_\emptyset$. $\bar{Q}_\emptyset$ becomes a Ø at this time. Since the clock input of flip/flop 1 made a transition from a logic Ø to a logic 1 the output of flip/flop 1 does not change state. Since no change occurred on the inputs of flip/flop 2 the output of flip/flop 2 does not change state.

Now let's see what happens when we push the button again. This time when we push the button the D input of flip/flop Ø sees a logic Ø. Therefore, when the button is pushed this logic Ø will appear on its output $Q_\emptyset$. When this happens flip/flop 1 will see a transition on its clock line from a 1 to a Ø. Since its D input is a 1, its output, Q_1 will become a logic 1. On the third button push the D input to flip/flop Ø is again a logic 1. Therefore, the output $Q_\emptyset$ becomes a logic 1 when the button is pushed a third time. Since the clock line of flip/flop 1 is making a transition from a Ø to a 1, no change in state occurs at the output of flip/flop 1. Since flip/flop 2 sees no change on its inputs no change in flip/flop 2's outputs occur.

When the button is pushed the fourth time the output of flip/flop Ø, $Q_{\emptyset}$, becomes a logic Ø. This causes a 1-to-Ø transition on the clock line of flip/flop 1 and output Q_1 also becomes a logic Ø. This causes a 1-to-Ø transition on the clock line of flip/flop 2. The output of flip/flop 2, Q_2, now becomes a logic 1.

This action is summarized in Figure 2.21.

BUTTON PUSH	$Q_{\emptyset}$	Q_1	Q_2
Ø	Ø	Ø	Ø
1	1	Ø	Ø
2	Ø	1	Ø
3	1	1	Ø
4	Ø	Ø	1
5	1	Ø	1
6	Ø	1	1
7	1	1	1
8	Ø	Ø	Ø

Figure 2.21 OUTPUT STATES OF DOORBELL COUNTING CIRCUIT

Notice that each time the button is pushed the output $Q_{\emptyset}$ changes state. If it has been a Ø it becomes a 1. If it has been a 1 it becomes a Ø. Each time output $Q_{\emptyset}$ makes the transition from a 1 to a Ø the output Q_1 changes state. Likewise, each time the output Q_1 makes a transition from a 1 to a Ø the output Q_2 changes state. In this manner, output $Q_{\emptyset}$ cycles completely for each two times the button is pushed. Output Q_1 cycles completely each time $Q_{\emptyset}$ cycles twice or each four times the button is pushed. And output Q_2 cycles completely for each two cycles of Q_1, each four cycles $Q_{\emptyset}$, or each eight times the button is pushed.

By looking at this table you could apply your volt meter to the outputs $Q_{\emptyset}$, Q_1, and Q_2, determine what combination of states appeared, and figure out how many times your doorbell had been pushed while you were gone. However, this only works for up to seven pushes of the button. For the eighth push of the button, all three flip/flops see a transition on their clock lines from a logic 1 to a logic Ø, and all three outputs become logic Ø. So it is impossible to tell the eighth button push from no button pushes at all. However, if you had hooked up another flip/flop to the output of flip/flop 2 in similar fashion, its output would have become a logic 1, while the other three were reverting to logic Øs when the eighth button push occurred.

Extending the circuit by adding flip/flops in this manner would allow one to count to an arbitrarily high number. Notice that for each flip/flop the time to cycle from a Ø to a 1 and back to Ø is twice as long as that for the preceding flip/flop. For flip/flop Ø it is two button pushes. For flip/flop 1 it is four button pushes. For flip/flop 2 it is eight button pushes. For four flip/flops the number would be 2 x 2 x 2 x 2 = 16 button pushes. And for 5 flip/flops it would be 2 x 2 x 2 x 2 x 2 = 32 button pushes.

Combining Flip/Flops into Registers

In the preceding sections we saw how a single flip/flop could store one bit of information. While this is often useful in a computer it is more often useful to store a group of bits at once. This is usually done by means of a register. A register is nothing more than a group of flip/flops arranged as a single circuit as shown in Figure 2.22. When the clock line makes a transition from a 1 to a Ø, whatever combination of bits appears on the input lines will be stored in the flip/flops, and will appear on the output lines, until another clock transition takes place. This is true unless the Reset line is momentarily raised to a logic 1, or as it may be said, pulsed to a logic 1, forcing all of the flip/flops to revert to the Reset condition with their outputs at logic Ø. This operation is known as clearing the *register*. In microcomputers it is especially common to arrange *registers* as groups of exactly 8 flip/flops. According to common convention 8 bits of information are called 1 byte of information.

Given that each *byte* of information contains 8 bits and that each bit can exist in two possible conditions, how many unique combinations of bits can occur within one *byte*? This is easy to calculate. Each bit has two possibilities. In one byte there are 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 = 2^8 = 256 possible combinations.

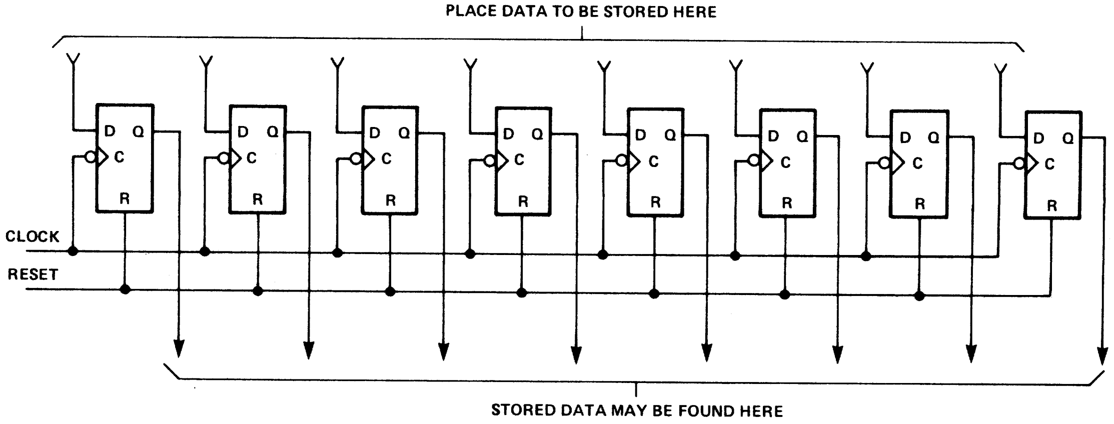


Figure 2.22 AN 8-BIT REGISTER

In computers, *registers* are used to store data. The data generally represent numbers, logic values, or computer programs. Exactly what is represented by the combination of 1s and Øs stored in a *register* must be kept track of by the computer programmer.

By looking at this table you could apply your volt meter to the outputs $Q_Ø$, Q_1, and Q_2, determine what combination of states appeared, and figure out how many times your doorbell had been pushed while you were gone. However, this only works for up to seven pushes of the button. For the eighth push of the button, all three flip/flops see a transition on their clock lines from a logic 1 to a logic Ø, and all three outputs become logic Ø. So it is impossible to tell the eighth button push from no button pushes at all. However, if you had hooked up another flip/flop to the output of flip/flop 2 in similar fashion, its output would have become a logic 1, while the other three were reverting to logic Øs when the eighth button push occurred.

Extending the circuit by adding flip/flops in this manner would allow one to count to an arbitrarily high number. Notice that for each flip/flop the time to cycle from a Ø to a 1 and back to Ø is twice as long as that for the preceding flip/flop. For flip/flop Ø it is two button pushes. For flip/flop 1 it is four button pushes. For flip/flop 2 it is eight button pushes. For four flip/flops the number would be 2 x 2 x 2 x 2 = 16 button pushes. And for 5 flip/flops it would be 2 x 2 x 2 x 2 x 2 = 32 button pushes.

Combining Flip/Flops into Registers

In the preceding sections we saw how a single flip/flop could store one bit of information. While this is often useful in a computer it is more often useful to store a group of bits at once. This is usually done by means of a register. A register is nothing more than a group of flip/flops arranged as a single circuit as shown in Figure 2.22. When the clock line makes a transition from a 1 to a Ø, whatever combination of bits appears on the input lines will be stored in the flip/flops, and will appear on the output lines, until another clock transition takes place. This is true unless the Reset line is momentarily raised to a logic 1, or as it may be said, pulsed to a logic 1, forcing all of the flip/flops to revert to the Reset condition with their outputs at logic Ø. This operation is known as clearing the *register*. In microcomputers

3. HOW COMPUTERS COUNT

THE BINARY NUMBERING SYSTEM

The processes of counting and assigning numerical value are fundamental to any computing activity. You engage in these processes many times daily. At an early age it became second nature to you so that the process itself required little attention. In today's computers the counting process is reduced to its simplest possible level. The notation we use to describe computer counting may not be familiar to you yet. The principles involved are familiar, however, as we shall presently see. With a little practice you can quickly learn to express the counting process in computer notation. Perhaps you recognized in the section on counting circuits in the last chapter how a counting process was actually taking place. The state of each flip/flop in our "counting circuit" can be represented by one of two symbols, Ø or 1. If we assume that the three flip/flops are all in the Ø state to begin with, we can see that each clock pulse will produce a new and unique pattern of Øs and 1s, as shown in Table 3.1. Notice how a unique combination of three "bits" was produced for each of the first 7 clock pulses. Notice also that on the 8th clock pulse that these three bits revert to Ø-Ø-Ø and begin another cycle of the same counting pattern.

Had another flip/flop been added to our circuit it would have been switched to the 1 state during the second cycle. By connecting still more flip/flops to our circuit we could count to an arbitrarily high number. Each clock pulse would have a unique bit pattern associated with it. What we have really been doing is counting in the binary numbering system. As you can see each unique bit pattern is associated with a given clock pulse. If you examine this sequence closely, you may also see that a simple rule applies in progressing from one *binary number (bit pattern)* to the next: On each clock pulse, the state of the first flip/flop, flip/flop Ø, changes. For every 2 clock pulses the state of the second flip/flop, flip/flop 1, changes; for every 4 clock pulses the state of the 3rd flip/flop, flip/flop 2, changes and so on. For every 2^n clock pulses the state of the F/F_n changes.

Table 3.1 FLIP/FLOP COUNTING PATTERN

CLOCK PULSE NUMBER	(F/F_3)	F/F_2	F/F_1	$F/F_Ø$
Ø		Ø	Ø	Ø
1		Ø	Ø	1
2		Ø	1	Ø
3		Ø	1	1
4		1	Ø	Ø
5		1	Ø	1
6		1	1	Ø
7		1	1	1
8	(1)	Ø	Ø	Ø
9	(1)	Ø	Ø	1

There is a strong similarity between this binary system of counting and the decimal system we use in everyday life. To point out their similarities, let's examine a longer table of decimal numbers and their binary equivalents. See Table 3.2.

As we move down the decimal column of the table, notice what happens. First, we sequence through our entire vocabulary of unique symbols, (Ø-9) in the least significant (right hand) digit. We then increase the value of the tens column by 1 and repeat the sequence of the *least significant digits*. This process is repeated until, when we reach 99, our symbol vocabulary is exhausted for the tens column. We then increment the hundreds column in a similar fashion.

Exactly the same process is taking place on the binary side of the table, except that we have reduced the number of unique symbols used from 10 to 2. Thus, when we reach a count of one in the *least significant digit*, our symbol vocabulary is exhausted, and we must increment the

Table 3.2 DECIMAL AND BINARY EQUIVALENTS

Decimal	Binary
Ø	Ø
1	1
2	1Ø
3	11
4	1ØØ
5	1Ø1
6	11Ø
7	111
8	1ØØØ
9	1ØØ1
1Ø	1Ø1Ø
11	1Ø11
12	11ØØ
13	11Ø1
14	111Ø
15	1111
16	1ØØØØ
17	1ØØØ1
18	1ØØ1Ø
19	1ØØ11
2Ø	1Ø1ØØ
21	1Ø1Ø1
-	-
-	-
-	-
29	111Ø1
3Ø	1111Ø
31	11111
32	1ØØØØØ
33	1ØØØØ1
-	-
-	-
-	-
99	11ØØØ11
1ØØ	11ØØ1ØØ
1Ø1	11ØØ1Ø1
-	-
-	-
-	-
199	11ØØØ111
2ØØ	11ØØ1ØØØ
2Ø1	11ØØ1ØØ1
-	-
-	-
-	-
254	1111111Ø
255	11111111
256	1ØØØØØØØØ

next least significant digit. While in the decimal system the *next least significant digit* had a "weight" of ten, the binary system has a "weight" of only two. The weight of each successive digit in the binary system increases by a factor of two. Thus, a 1 in the *least significant* column is worth 1. A "1" in the *next least significant* column is worth 2 times 1 or 2. The *second least significant* column is worth 2 times 2 or 4. The *third least significant* column is worth 2 times 4 or 8, and so on. Instead of the ones, tens, and hundreds columns of the decimal system, we have the ones, twos, fours, eights, sixteens (etc.) columns of the binary system.

To further illustrate the binary counting process, consider the odometer in your car. In theory, it would be a simple matter to build a binary odometer. If you were to take your present odometer, erase the numbers from all the wheels and paint a Ø on one side of each and a 1 on the other side of each you would then have a binary odometer. Of course, for it to read correctly in miles you would have to change the gearing ratio to the odometer. As the right hand wheel turns around, it would progress from the binary digit Ø to 1. As the Ø came around again it would increase the next odometer wheel to 1. While the notation has changed, the counting *process* is similar to that used in the decimal system.

Let us quickly review some things about the decimal and binary number systems. When we write a number in the decimal number system, such as 254, we know that the four means four units or four ones because it appears in the right hand, least significant, or lowest order, column. We know the five means 50 because it appears in the next least significant, or tens column. We know the two means 200 because it appears in the third least significant column, or hundreds column. The meaning of the decimal number 254, is really

2 x (1Ø x 1Ø) + 5 x 1Ø, + 4 x 1 or 2 x $1Ø^{2}$ + 5 x $1Ø^{1}$ + 4 x $1Ø^{Ø}$.

It is not legitimate to write down the character sequence 254 in the binary number system, because only the characters Ø and 1 are allowable as binary digits. In the binary system the least significant or lowest order column is the ones column. However, the next least significant digit is not the tens column but the twos column. The third least significant is the fours column. Each column increases in weight by the base of the number system, in this case, two. The base of the number system is determined by how many unique symbols are used in the number system.

CONVERSION FROM BINARY TO DECIMAL

If we understand the weighting of the columns in binary notation, it is a simple matter to convert a number from its binary form into its decimal equivalent. For instance, let's look at the binary number 111Ø1. The first step is to be aware of the weight of each column. Starting with the right hand column the weights are successively 1, 2, 4, 8, and 16. The weight of each column is twice as great as that to its immediate right. A 1 appearing in the sixteens column means 1 x 16. A 1 appearing in the eights column means 1 x 8. A 1 appearing in the fours column means 1 x 4. A Ø appearing in the twos column means Ø x 2. And a 1 appearing in the ones column means 1 x 1. Thus we have

1 x 16 + 1 x 8 + 1 x 4 + Ø x 2 + 1 x 1

= 16 + 8 + 4 + Ø + 1 = 29.

Notice that 111Ø1 is the binary number shown as equivalent to the decimal number 29 in Table 3.2. To distinguish this number in binary notation from the decimal number 11,1Ø1 we may write $29_{1Ø} = 111Ø1_{2}$. The subscript denotes the base of the number system.

To further illustrate the behavior of binary columns let us extract some special numbers from Table 3.2. Let's take all of the decimal numbers which are even powers of 2. In this manner we may construct Table 3.3.

Table 3.3 BINARY EQUIVALENTS OF POWERS OF 2

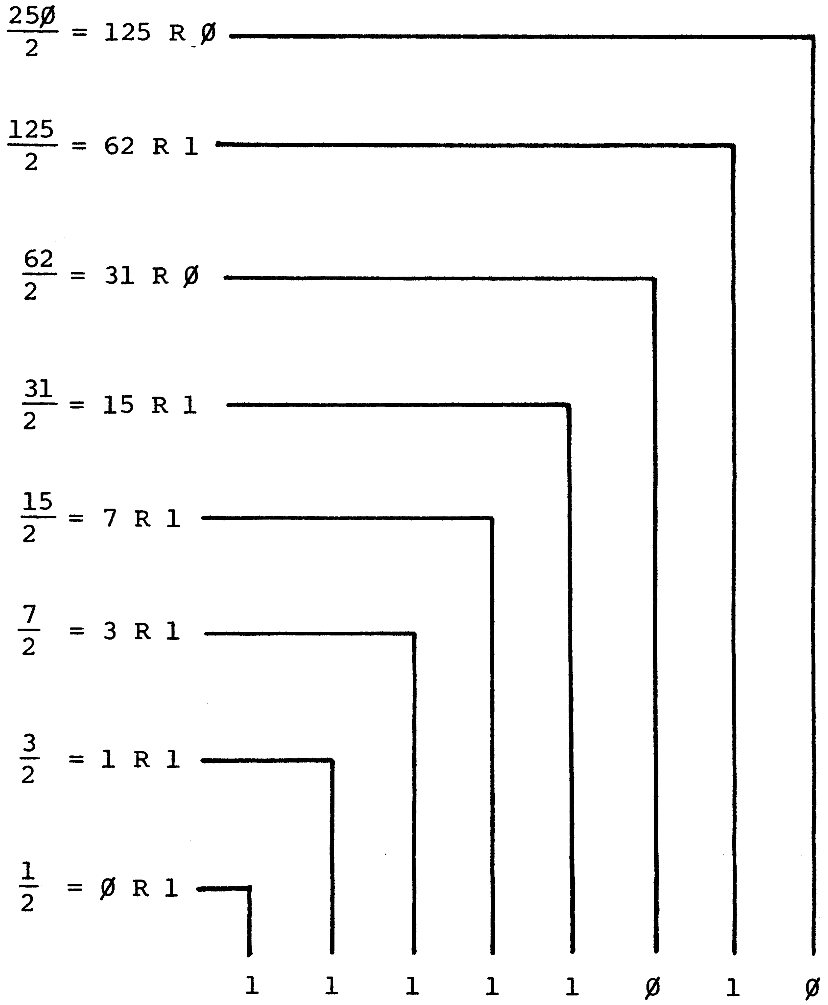
DECIMAL		BINARY		POWER OF 2
$1_{1Ø}$	=	1_{2}	=	$2^{Ø}$
$2_{1Ø}$	=	$1Ø_{2}$	=	2^{1}
$4_{1Ø}$	=	$1ØØ_{2}$	=	2^{2}
$8_{1Ø}$	=	$1ØØØ_{2}$	=	2^{3}
$16_{1Ø}$	=	$1ØØØØ_{2}$	=	2^{4}
$32_{1Ø}$	=	$1ØØØØØ_{2}$	=	2^{5}
$64_{1Ø}$	=	$1ØØØØØØ_{2}$	=	2^{6}
$128_{1Ø}$	=	$1ØØØØØØØ_{2}$	=	2^{7}
$256_{1Ø}$	=	$1ØØØØØØØØ_{2}$	=	2^{8}

As an exercise, make up a few binary numbers. Convert them to their decimal equivalents and see if they come out the same as the decimal equivalents shown in Table 3.2.

CONVERSION FROM DECIMAL TO BINARY

Converting numbers from decimal form into their binary equivalents is nearly as straightforward as the other way around. Of course, the simplest way to do this is to look up the binary equivalent in a table such as 3.2. However, if you don't have a table handy the following method will work.

First, divide the decimal number by two and write down the remainder. Repeat this process until the number has been reduced to Ø. The first remainder is the least significant digit of your binary number. The last remainder is the most significant digit of your binary number. For example, let's convert the decimal number 25Ø into binary form. We have



$\frac{\text{25Ø}}{2}$ = 125 R Ø

$\frac{125}{2}$ = 62 R 1

$\frac{62}{2}$ = 31 R Ø

$\frac{31}{2}$ = 15 R 1

$\frac{15}{2}$ = 7 R 1

$\frac{7}{2}$ = 3 R 1

$\frac{3}{2}$ = 1 R 1

$\frac{1}{2}$ = Ø R 1

1 1 1 1 1 Ø 1 Ø

The binary number is thus 11111Ø1Ø. Or $\text{11111Ø1Ø}_2 = \text{25Ø}_{\text{1Ø}}$.

Let's go back to our original example and see why this works. When we divided 25Ø by 2 we found that there were 125 twos and no ones. If $125_{\text{1Ø}}$ were a legitimate binary digit we could put it in the twos column and a Ø in the ones column and we would be done. Since it isn't, the question is now of the 125 twos, how many fours are there? If we divide 125 by 2 we find that there are 62 fours, and one two. Again if 62 were a legitimate binary digit we could put it in the fours column, the 1 in the twos column and the Ø in the ones column and we would be done. But since 62 is not a legitimate binary number we must again divide by 2 to find out how many eights there are. This results in an answer of 31 eights and no fours. Again 31 is not a legitimate binary digit so we divide it by 2 to determine that there are 15 sixteens and one eight. 15 ÷ 2 means 7 32s and 1 sixteen. 7 divided by 2 tells us that there are three 64s and one 32. 3 divided by 2 tells us that there is one 128 and one 64. And 1 divided by 2 tells us that there are no 256s and one 128. In this case the last step would not have been necessary, since we already knew there was one 128, and one is a legitimate binary digit.

As an exercise try converting the following decimal numbers into their binary equivalents.

a. $258_{1\emptyset}$ d. $62_{1\emptyset}$

b. $13\emptyset_{1\emptyset}$ e. $128_{1\emptyset}$

c. $67_{1\emptyset}$ f. $15_{1\emptyset}$

BINARY ARITHMETIC

Binary numbers can be added, subtracted, multiplied, and divided in a manner similar to the equivalent processes for decimal numbers. As a matter of fact, these possibilities are considerably simpler for the binary number system than they are in decimal. Consider the process of addition. In adding two binary digits, there are only four possibilities:

Addition

a) Ø + Ø = Ø

b) Ø + 1 = 1

c) 1 + Ø = 1

d) 1 + 1 = Ø and a carry.

Let's do some examples of simple decimal addition problems shown with their binary equivalents:

Decimal	Binary
5	Ø1Ø1
+ 2	ØØ1Ø
7	Ø111

In this example we have 1 + Ø = 1 for the ones column, Ø + 1 = 1 for the twos column, and 1 + Ø = 1 for the fours column. Note that leading zeroes are understood (as shown by) when they are not explicitly stated.

Next, let's try a slightly more complicated example, in which a carry occurs:

```
        Decimal                  Binary

                          carry  1  1

           6                     0  1  1  Ø
        +  2                     0  Ø  1  Ø
        --------                ------------
           8                     1  Ø  Ø  Ø
```

Here we have Ø + Ø = Ø for the ones column. In the twos column, we have 1 + 1 = Ø and a carry to the fours column. In the fours column we have 1 + Ø = 1 which, when added to the carry, gives 1 + 1 = Ø with another carry. In the eights column we have Ø + Ø = Ø which, when added to the carry, produces 1 + Ø = 1.

One further example will illustrate the use of carries:

```
        Decimal                  Binary

                          carry     1  1

          11                     1  Ø  1  1
        +  3                     Ø  Ø  1  1
        --------                ------------
          14                     1  1  1  Ø
```

In the ones column, 1 + 1 = Ø and a carry to the twos column. In the twos column, 1 + 1 = Ø and a carry to the fours column. Adding the carry into the twos column we have 1 + Ø = 1. In the fours column we have Ø + Ø = Ø which, added to the carry, gives 1 + Ø = 1. In the eights column, we have 1 + Ø = 1.

Make up a few examples for yourself and try them to see that the answer to an addition problem is the same, whether it is performed in binary notation or decimal notation.

Subtraction

Binary subtraction can be carried out in a method analogous to decimal subtraction. However, most computers do not have the necessary hardware to carry out this function directly. Suppose, for example, that a large number is to be subtracted from a small number. The result would be a negative answer. Thus, it is necessary to be able to represent negative numbers in binary notation. Since a representation for negative numbers is required anyway, it turns out to be simpler to convert a positive number to its negative equivalent and add, rather than to carry out the subtraction process directly.

Several schemes have been proposed for representing negative numbers in binary notation. One possible method is to reserve one bit to indicate the sign of the number using the rest of the bits to indicate the numerical value. This works nicely. If the most significant bit of a binary number is called the sign bit, a Ø in this bit means that the number is positive. A 1 in this bit means that this number is negative. In attempting to represent negative numbers in binary form, we must use a notation which meets these additional requirements:

1. The sum of two negative numbers must be a negative number of proper magnitude.
2. The sum of a positive and negative number must be of proper sign and magnitude.

It turns out that this can be accomplished by what is called two's complement notation. The two's complement of a binary number is its negative equivalent. To convert a binary number into its *two's complement*, follow these two steps:

1. Invert all bits in the number. In other words, wherever there is a 1 write down a Ø. Wherever there is a Ø write down a 1.

2. Then add 1 to the resulting number. To illustrate, consider some examples using 8-bit binary numbers. Let's take the binary equivalent

of the decimal number 5, which is ØØØØØ1Ø1 and convert it to negative 5 (in *two's complement* notation). (When the bits are inverted we have 11111Ø1Ø. Adding 1 to this we get 11111Ø11.)

Original number	ØØØØØ1Ø1$_2$ = 5$_{1Ø}$
(1) Invert all bits	11111Ø1Ø
(2) Add 1	11111Ø11$_2$ = -5$_{1Ø}$

This is the binary *two's complement* notation for the decimal number -5. Notice that the *most significant bit* is a 1, the *most significant bit* being the *sign bit*, and indicates that the *sign* is negative. If this is a valid representation for -5, we should be able to take its *two's complement* again and come up with +5. Let's try.

Original number	11111Ø11$_2$ = -5$_{1Ø}$
Invert all bits	ØØØØØ1ØØ
Add 1	ØØØØØ1Ø1$_2$ = 5$_{1Ø}$

First we invert all the bits, coming up with ØØØØØ1ØØ. Then add 1 and we get ØØØØØ1Ø1, which is indeed the binary representation of +5. To prove that this notation works let's do a couple of sample problems.

For instance, let's subtract 5 from 7, and see if we get 2. The way we'll actually do this problem is 7 + (-5) = 2.

Two's complement of 5	11111Ø11 = -5$_{1Ø}$
Add to 7	11111Ø11 + ØØØØØ111 = ØØØØØØ1Ø$_2$

First, we take the *two's complement* of the number 5 which is 11111Ø11. This is then added to 7. 11111Ø11 + ØØØØØ111 yields ØØØØØØ1Ø, the binary equivalent of 2. There is a carry out of the *most significant digit*. This will always be true when a smaller number is subtracted from a larger one and must be ignored.

To prove that this works if we subtract a larger number from a smaller one, consider the example 5 - 7.

Binary 7	ØØØØØ111
Complement all bits	11111ØØØ
Add 1 to get two's complement	11111ØØ1
Add binary equivalent of 5	+ ØØØØØ1Ø1
	= 1111111Ø

First we convert 7 to its *two's complement* equivalent. Inverting all the bits we get 11111ØØØ, adding 1 we get 11111ØØ1. Adding to this the binary equivalent for 5 we add ØØØØØ1Ø1, the result being 1111111Ø. We can immediately see that the result is a negative number because the sign bit is a 1.

Let's take the *two's complement* of this number to see if we get the equivalent of binary 2.

Invert all bits	ØØØØØØØ1
Add 1	ØØØØØØ1Ø = $2_{1\emptyset}$

Inverting all the bits we get ØØØØØØØ1. Adding 1 we get ØØØØØØ1Ø, which is indeed the binary equivalent of 2.

As a final example, let's add two negative numbers together to see if we get a negative result. For this example, we'll add -3 and -4 to perform $(-3) + (-4) \overset{?}{=} -7$. First, we write down the binary equivalent for each number.

Binary equivalent of 3	ØØØØØØ11
Binary equivalent of 4	ØØØØØ1ØØ
Two's complement of 3	111111Ø1
Two's complement of 4	+ 111111ØØ
Add together	= 11111ØØ1 with a carry
Ignore carry	
Two's complement of result	ØØØØØ111 = $7_{1Ø}$

We have ØØØØØØ11 for 3, and ØØØØØ1ØØ for 4. Taking the *two's complements* for these numbers we get 111111Ø1 for 3 and 111111ØØ for 4. Adding the two *two's complement* numbers we get 11111ØØ1 with a carry. Ignoring the carry and taking the *two's complement* of our resultant number we get ØØØØØ111, which is indeed the binary equivalent for 7.

A carry is generated in the case of (1) adding two negative numbers and (2) adding a large negative number to a small positive number. This is expected and must be ignored. By using this expedient it is possible to easily perform addition and subtraction operations and get accurate results working with both positive and negative numbers.

Notice that a number in *two's complement* notation, if it is a negative number, is much different in value from the same combination of binary digits in direct binary notation. Either signed or unsigned notation may be used in solving a particular problem. It is up to the computer programmer to keep track of what notation he is using.

Multiplication and Division

Multiplication and division problems can be solved using binary notation by successive addition or subtraction of the multiplier or divisor. The familiar rules used in working with the decimal number system also work in binary notation. For instance, using binary notation we should be able to multiply 6 x 5 to get 30 and divide 30 by 5 to get 6. This would be done as follows:

```
   6                 11Ø
 x 5               x 1Ø1
 ---             --------
  3Ø                 11Ø
                    ØØØØ
                ØØØ11ØØØ
                ØØØ1111Ø
```

```
     6                  11Ø
 5 | 3Ø         1Ø1 | 1111Ø
                      1Ø1
                       1Ø1
                       1Ø1
                       ØØØ
                       ØØØ
                         Ø
```

Try this also: Multiply 9 x 3 to get 27, then divide 27 by 3 to get 9.

```
   1ØØ1                    1ØØ1_2 = 9_1Ø
   x 11              11 | 11Ø11
   1ØØ1                   11
  1ØØ1Ø                    ØØ
  11Ø11_2 = 27_1Ø          ØØ
                            Ø1
                            ØØ
                             11
                             11
                              Ø
```

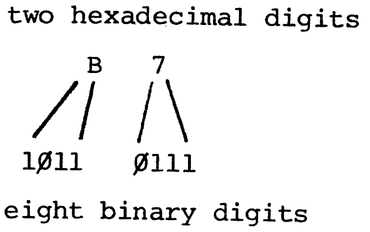
THE HEXADECIMAL NUMBERING SYSTEM
A CONVENIENT COMPROMISE

Most microcomputers, including the Instructor 50, store binary numbers in 8-bit bytes. A string of 8 or more 1s and Øs is easy for the computer to understand. It gets very confusing for us humans, however, if we have to remember and work with a long string of 1s and Øs. But, if we work with our more familiar decimal number system, we find that we constantly have to convert between decimal and binary numbers. This becomes a tedious and time consuming process. Fortunately, there is a convenient compromise.

This compromise is a third numbering system called a hexadecimal system. The hexadecimal system is a base 16 number system, and as such it has 16 unique symbols in its vocabulary. The first 10 symbols are the familiar characters Ø through 9. For the remaining 5 symbols we use the alphabetic symbols A, B, C, D, E, and F. Numbers written in the *hexadecimal system* are more similar in appearance to our familiar decimal numbers. However, they are very easy to convert back and forth between *hexadecimal* and binary forms. In fact, with a little practice, you will be able to convert between binary and *hexadecimal* notations by inspection.

A hint of why this is true may be seen by examining Table 3.4, which is a comparison of binary, *hexadecimal*, and decimal numbering systems. Notice that for the 16 unique symbols in the *hexadecimal* numbering system, Ø through F, there are 16 possible combinations (and only 16 possible combinations) of 4 binary digits. Therefore, each *hexadecimal* digit may be thought of as representing 4 binary digits, or each group of 4 binary digits may be thought of as representing 1 *hexadecimal* digit.

For example, let's convert the *hexadecimal* number $B7_{16}$ into its binary equivalent. The most significant *hexadecimal* digit, B, translates into binary 1Ø11. The least significant *hexadecimal* digit, 7, transmits into Ø111. Therefore, the binary number equivalent to *hexadecimal* B7 is 1Ø11Ø111. The conversion from binary notation into *hex-notation* is equally straightforward.



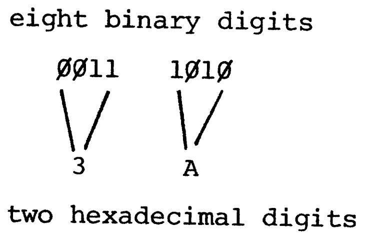


Table 3.4 COMPARISON OF DECIMAL, HEXADECIMAL, AND BINARY NOTATION

DECIMAL (BASE 10)		= HEXADECIMAL (BASE 16)		= BINARY (BASE 2)	
Ø	1Ø unique symbols	Ø	16 unique symbols	ØØØØ	2 unique symbols
1		1		ØØØ1	
2		2		ØØ1Ø	
3		3		ØØ11	
4		4		Ø1ØØ	
5		5		Ø1Ø1	
6		6		Ø11Ø	
7		7		Ø111	
8		8		1ØØØ	
9		9		1ØØ1	
1Ø		A		1Ø1Ø	
11		B		1Ø11	
12		C		11ØØ	
13		D		11Ø1	
14		E		111Ø	
15		F		1111	

For instance, let us take the binary number ØØ111Ø1Ø. First, starting with the least significant digit we group the binary digits in groups of 4. On the right hand side then we have 1Ø1Ø and on the left hand side we have ØØ11. Next, we simply substitute the corresponding hexadecimal digits for these combinations of binary digits. The resulting hexadecimal number is 3A.

It is also relatively easy to convert hexadecimal numbers to and from decimal form. To convert a hexadecimal number into decimal form we use the principal of column weighting as was done with the binary system. In the hexadecimal system each relatively more significant column increases in weight by the factor of 16. Thus, the least significant hexadecimal digit appears in the ones column, the second least significant hexadecimal digit in the 16s column, the third least significant hexadecimal digit is in the 256s column and so on. The method is as follows:

Consider the hexadecimal number A29C. This can be re-written

$A \times 16^3 + 2 \times 16^2 + 9 \times 16^1 + C \times 16^{\emptyset}$.

This is the same as

$1\emptyset_{1\emptyset} \times 4\emptyset96 + 2 \times 256 + 9 \times 16 + 12 \times 1$

or $4\emptyset96\emptyset_{1\emptyset} + 512_{1\emptyset} + 144_{1\emptyset} + 12_{1\emptyset} =$

$A29C_{16} = 41{,}628_{1\emptyset}$.

To convert this number back into hexadecimal form we use the successive division method that we used in converting decimal numbers into binary. In this case, however, instead of doing successive division by 2 we must successively divide by 16. The method is shown below.

$$\frac{41{,}628_{1\emptyset}}{16_{1\emptyset}} = 26\emptyset1_{1\emptyset} \text{ R } 12_{1\emptyset} \quad \text{or R } C_{16}$$

$$\frac{26\emptyset1_{1\emptyset}}{16_{1\emptyset}} = 162_{1\emptyset} \text{ R } 9 \quad \text{or R } 9_{16}$$

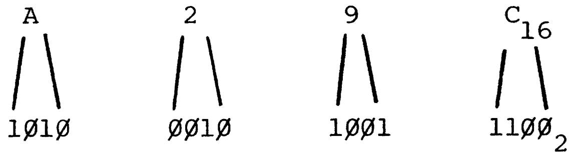
$$\frac{162_{1\emptyset}}{16_{1\emptyset}} = 1\emptyset_{1\emptyset} \text{ R } 2_{1\emptyset} \quad \text{or R } 2_{16}$$

$$\frac{1\emptyset_{1\emptyset}}{16_{1\emptyset}} = \emptyset_{1\emptyset} \text{ R } 1\emptyset_{1\emptyset} \quad \text{or R } A_{16}$$

$$A \quad 2 \quad 9 \quad C_{16} = 41{,}628_{1\emptyset}$$

As each remainder is brought down it is converted into its hexadecimal digit equivalent. Thus, we have been able to take the hexadecimal number A29C from hexadecimal form into decimal form and back into hexadecimal form again. As a further example, this 4 digit hexadecimal number may be converted into a 16 digit binary equivalent by substituting the appropriate 4 bit grouping for each hexadecimal digit. We then have A29C$_{16}$ = 1Ø1ØØØ1Ø1ØØ111ØØ$_{2}$.

As an exercise, convert this binary number into decimal form and show that it is equal to 41,628$_{1Ø}$.



4. INSIDE YOUR INSTRUCTOR 50

BASIC COMPUTER ORGANIZATION

In Chapter Two we learned about the basic building blocks of computers: logic signals, gates, flip/flops, and registers. In Chapters Two and Three we saw how these circuits could be made to count, store, and represent numbers. We also got a glimpse of how two numbers might be compared by our computer to make a logical decision.

In this chapter, we get our first real look at how an actual computer is organized. Much of this information is general in nature and applicable to nearly any computer. The Instructor 50 is particularly appropriate for this purpose, since the details of its construction are similar to both mini-computers and micro-computers.

For all modern general-purpose computers, from hand-held calculators to large data processing systems, the basic organization is similar to that shown in Figure 4.1.

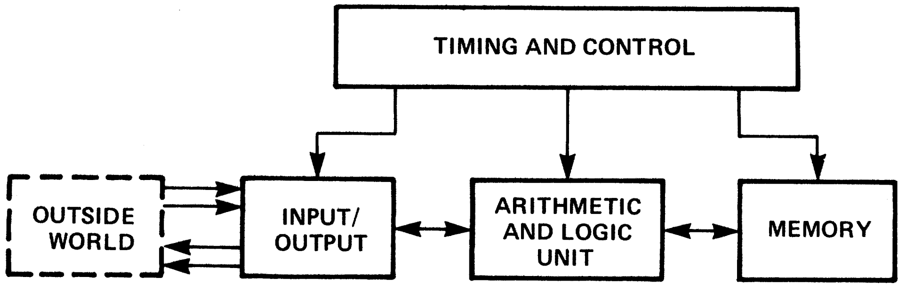


Figure 4.1 BASIC COMPUTER ORGANIZATION

Memory System

The memory section is simply a collection of *registers* in which binary numbers can be stored. Each register occupies a position in memory which is known as its location or address. In your Instructor 50, each *memory location* contains 8 bits (1 byte) of data (see Figure 4.2).

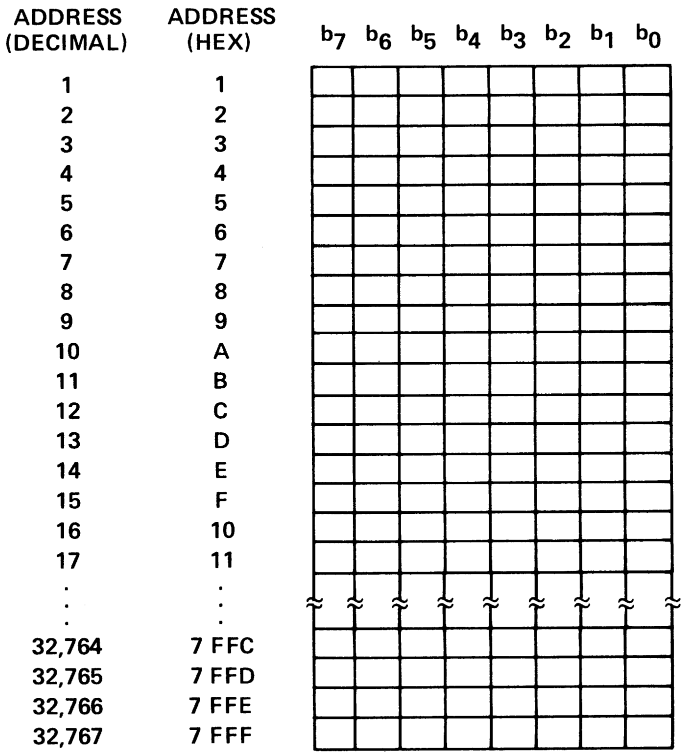


Figure 4.2 DIAGRAM OF 2650 MEMORY

It is important to keep in mind that there are two distinct values associated with each location in memory: the address of the location and the contents of the location. The address of a location is similar to a street address or a post office box number. It specifies the position or location in the memory at which a given byte of data is stored. The contents of a memory location is the actual data stored. Data can be retrieved from memory (or stored in memory) only by first specifying the *address* at which the desired data is to be found (or stored). Just how this is done will be explained later.

Of the 32,768$_{1Ø}$ locations available in the memory space of the 2650 microprocessor, only 2688$_{1Ø}$ of them have been physically implemented in your Instructor 50. Of these, the first 512$_{1Ø}$ locations (addresses ØØØØ$_{16}$ through Ø1FF$_{16}$) are available to you as a user to store and retrieve data. You may wish to verify this on your Instructor 50 by storing data in a few locations, then retrieving it to the display. The following exercise will illustrate how to do this.

<u>Exercise</u>:

To store and retrieve data from your Instructor 50, follow the instructions below:

1. Plug your Instructor 50 into the wall socket. As soon as you connect the plug, the screen should display "Hello." The left-hand keyboard is a series of function keys. Find the one labelled "MEM." Press it to display ".Ad.=" This is the symbol for *address*.

2. The right-hand keyboard contains the hexadecimal (numerical) keys. Enter the number 1ØØ. The display will show ".Ad.= 1ØØ." You have now specified *address* 1ØØ$_{16}$.

3. Press the ENT/NXT key on the function (left-hand) keyboard to display ".Ø1ØØ XX" where XX is a random number. For instance, if the display reads ".Ø1ØØ FA". This means that the data in *address* 1ØØ is currently FA.

4. Enter into *address* 1ØØ the hexadecimal code Ø1 by pressing the keys "Ø" and "1" on the keyboard. Press the ENT/NXT key. The display now reads ".Ø1Ø1 XX. At this point, you have entered Ø1 into location Ø1ØØ, and displayed the contents of location Ø1Ø1.

5. For now, however, press MEM to return the display to ".Ad= ". Enter the address 1ØØ from the hexadecimal keyboard. Press the ENT/NXT

key. Your display will show ".Ø1ØØ Ø1" --the data you entered into address 1ØØ.

Now using the steps above enter this following data into addresses $5Ø_{16}$ to 55_{16}.

Data	76	2Ø	1F	ØØ	95	ØC
Location	5Ø	51	52	53	54	55

After you have completed entering your data, press the MEM key and check each address to see that the entries are correct.

Access location $2ØØ_{16}$ and enter the data Ø1 as you did above. Press the ENT/NXT key. What is the display?

Note: In attempting to store data into location $2ØØ_{16}$ you tried to access a non-implemented memory location. The message "Error 3" on the display indicates that your attempt has not been successful.

The Arithmetic and Logic System

A simplified block diagram of a typical computer's arithmetic and logic system is shown in Figure 4.3.

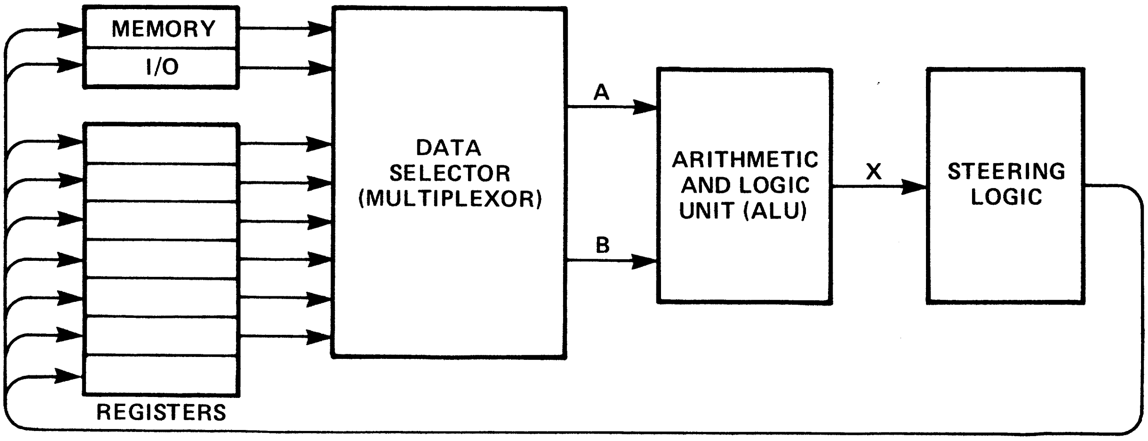


Figure 4.3 ARITHMETIC AND LOGIC SYSTEM BLOCK DIAGRAM

This system contains several *registers* for the temporary storage of data, usually in the form of eight bit bytes. A data selector, sometimes called a multiplexor connects the contents of one or two of these registers to the two inputs of the arithmetic and logic unit (ALU). The *arithmetic and logic unit* combines the two (eight-bit) inputs into one (eight-bit) output. It is called an *arithmetic and logic unit* because of the way its output is produced from its two inputs. For instance, the two inputs A and B may be added together as two binary numbers to produce a binary result, X, the sum of A and B. Or B may be subtracted from A in binary fashion to produce output X which is the difference between A and B. These are the arithmetic operations that it can perform. The logical operations consist of ANDing, ORing, or Exclusive-ORing the two inputs, or shifting or complementing one of the inputs. The results of these logical operations are illustrated in Figure 4.4.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
A	1	0	1	1	0	0	1	0
B	0	0	0	1	1	1	0	1

INPUTS

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	0	0	1	0	0	0	0	X = A "AND" B
1	0	1	1	1	1	1	1	X = A "OR" B
1	0	1	0	1	1	1	1	X = A "XOR" B
0	1	0	1	1	0	0	1	X = A "SHIFTED RIGHT"
0	1	0	0	1	1	0	1	X = $\bar{A}$ = A "COMPLEMENT"

OUTPUTS

Figure 4.4 ALU LOGICAL OPERATIONS

In each case the outputs are the result of doing some logical manipulation on the inputs. For the case, output X equals A AND B, each bit of the two input bytes is combined in an AND function to produce a bit of the output word. For instance, bit Ø of input A is ANDED to bit Ø of input B to produce bit Ø of the output word. Similar reasoning applies to the output in the case when X equals A OR B, or when X equals A Exclusive-ORed with B. The logical operation known as shifting simply means that each bit of the input byte is moved to the right one position or to the left one position, depending on whether a *left shift* or a *right shift* is being implemented. In the case of the *right shift*, as illustrated in Figure 4.4, a Ø was entered in the bit 7 position, or most significant bit. If an input is to be complemented each bit in the output byte is the opposite of the corresponding bit in the input byte. Each Ø becomes a 1 and each 1 becomes a Ø.

Once an output byte is produced by the *arithmetic and logic unit* it is directed by the steering logic into one of the registers, or into a memory location or to the computer's input/output system.

Blocks representing the memory and input/output systems of the computer are shown in Figure 4.3, because, while they are not part of the *arithmetic and logic system*, communication with them is sometimes necessary. The registers of Figure 4.3, are sometimes also called accumulators, because they accumulate the results of arithmetic and logical operations.

The Input/Output System

A computer would be of little use if it were impossible to get data into it from the outside world and to return processed data back to the outside world. This is the purpose of the input/output system of a computer.

The *input/output (I/O) system* of a computer consists primarily of an array of registers as illustrated in Figure 4.6. The primary difference between these *registers* and other *registers* of the computer is that signals to these *registers* are brought out to an electrical connector for connection to some external device.

Each of these *registers* is called a port. Depending on whether the function of the *port* is to bring data into the computer or transmit data out from the computer it is called an input port or an output port. For data to be input to the computer an external device is hooked up to the

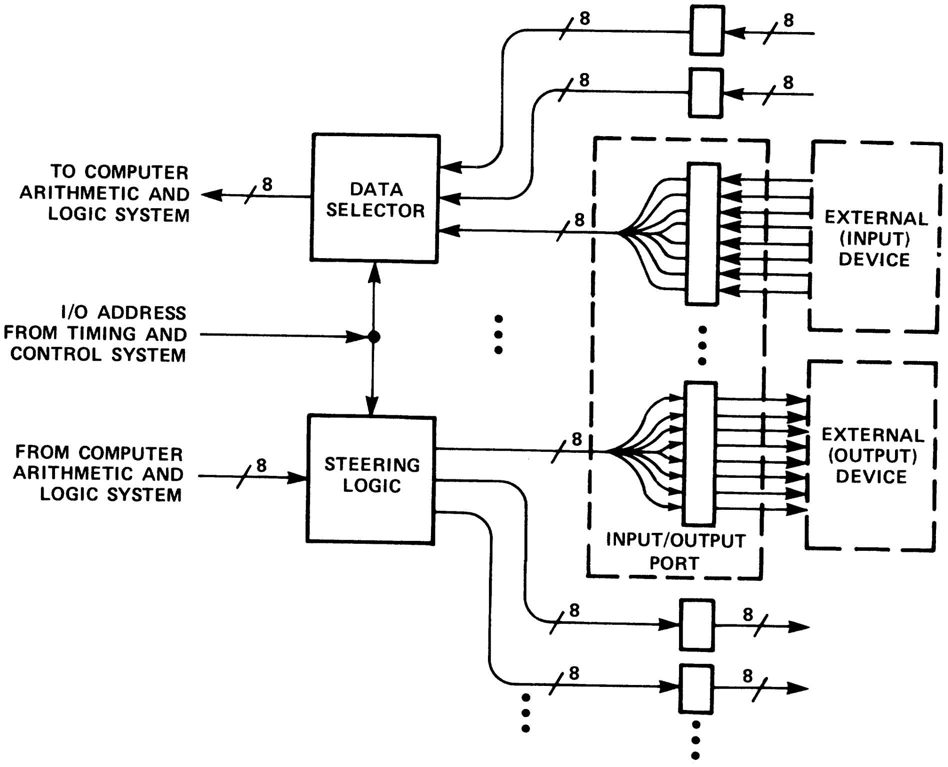

TO COMPUTER ARITHMETIC AND LOGIC SYSTEM
8
DATA SELECTOR
I/O ADDRESS FROM TIMING AND CONTROL SYSTEM
FROM COMPUTER ARITHMETIC AND LOGIC SYSTEM
STEERING LOGIC
EXTERNAL (INPUT) DEVICE
EXTERNAL (OUTPUT) DEVICE
INPUT/OUTPUT PORT

Figure 4.5 INPUT/OUTPUT SYSTEM

inputs of one of the *input ports*. Other *input ports* are likewise connected to their own separate external devices. For example, in your Instructor 50 the function keyboard, hexadecimal keyboard and eight toggle switches are connected to *input ports*.

The data selector logic then selects data from one of the several *input ports* and connects it to the arithmetic and logic system of the computer. In this manner, the computer may examine the signals being *input* from any external device, although only one external device may be examined at any one time. Since the sequence of examination may be very rapid, however, it will appear to the user of the computer that all *inputs* are being examined simultaneously.

An *output port* is very similar to an input port except that it is the outputs of the registers making up the *output port* which are connected to an external device. In operation a byte of data from the computer's arithmetic and logic system is presented to the steering logic of the

I/O system which connects it to one of several *ouput ports* available. This byte of data is then stored in the register corresponding to the selected *output port* and becomes available to the external device.

The *output port* registers are also sometimes called latches because they latch the data presented to them and hold it for the external device. On your Instructor 50 the eight digit LED display and the eight LEDs to the left of the operator's panel are connected to *output ports*. In computer terminology it is common to group a single *input port* with a single *output port* calling the combination an input/output port (I/O port).

The *input/output system*, then, may be thought of as a series of *input/output ports*, each of which is identified by an address or location called the input/output address. The purpose of the data selector for *input ports* and the steering logic for *output ports* is to select a particular port depending on the *address* that is presented to it by the computer. Notice that there is a strong similarity between the *input/output system* described here and the memory system described earlier. Data may be stored at a particular *address*, which is connected to an external output device. Data may be also retrieved from a particular *address*, which is connected to an external input device. This similarity is so strong that some computer designers elect to eliminate the *input/output system* entirely. Instead, *input and output devices* are connected as if they were locations in memory. This concept is illustrated on the operator panel of your Instructor 50. The eight LEDs and eight toggle switches form an eight-bit *input/output port*. The three position toggle switch just below this grouping allows you to connect the LEDs and switches either to memory location ØFFF, or to a port of the *input/output system*.

Exercise: Connect to ØFFF and read/write using MEM key.

The Timing and Control System

The timing and control block sends logic signals to the other three blocks (arithmetic and logic, memory, and input/output) to control their operations. It manipulates the input/output, arithmetic and logic, and memory circuits in a precisely timed sequence to make them perform the correct operations at the correct time. In this regard it is much like a puppeteer who operates a marionette by pulling on each of its strings in a precisely timed sequence to produce desired movements.

It is the *timing and control* section which tells the memory system when to store data into one of its locations, or to present data from one of its locations to the arithmetic and logic unit. Signals from the *timing and control* section also tell the data selector portion of the arithmetic and logic system which register to select, or to select data from the input/output system or memory system. It tells the arithmetic and logic unit what arithmetical or logical operation to perform. And it tells the steering logic where to direct the output of the arithmetic or logical calculation. The *timing and control* section also directs signals to the input/output system to tell it which external device (which input/output port) to select, when to transfer data from the external device to the computer, and when to transfer data from the computer to an output port.

Central Processor Unit

In your Instructor 50 the functions of the *timing and control system* and the *arithmetic and logic system* are combined on the 2650 microprocessor chip. This is a common practice with microcomputer systems and the combination is called the central processing unit or CPU.

COMPUTER "WARES"

Perhaps you've heard the terms hardware, software, and firmware before in connection with computers. Each of these "wares" is contained in your Instructor 50. In the discussion of the basic computer block diagram (Figure 4.1) you learned something about computer hardware, the physical stuff of which a computer is built. In the following sections we'll delve more deeply into the operation of these blocks as they are implemented in the Instructor 50. In the process we'll also learn about software, a term used in referring to computer programs.

What is a Program?

This question will take a while to answer and will become clearer as we progress. Briefly a program is a sequence of instructions to the computer which tell it what to do. These *instructions* are interpreted by the timing and control section of the computer. Each one specifies a sequence of logic signals which are output by the timing and control section to make the other sections perform some specific operation (such as adding two numbers together in the arithmetic and logic unit, or outputting data to a particular external device).

These *instructions* take the form of binary numbers, or codes, which are stored in memory locations. It is impossible to tell by examining the contents of any memory location whether these contents represent an *instruction* or some other data. For now suffice it to say that when the computer examines the contents of a certain memory location it expects those contents to be an *instruction code*, and interprets them as such.

Moving Information Between Blocks

By now it should be evident that much of a computer's activity involves moving bytes of information from one place to another within the computer. Figure 4.5 illustrates how this is accomplished. Basically, Figure 4.5 is similar to Figure 4.1 except that it is redrawn to emphasize communications paths.

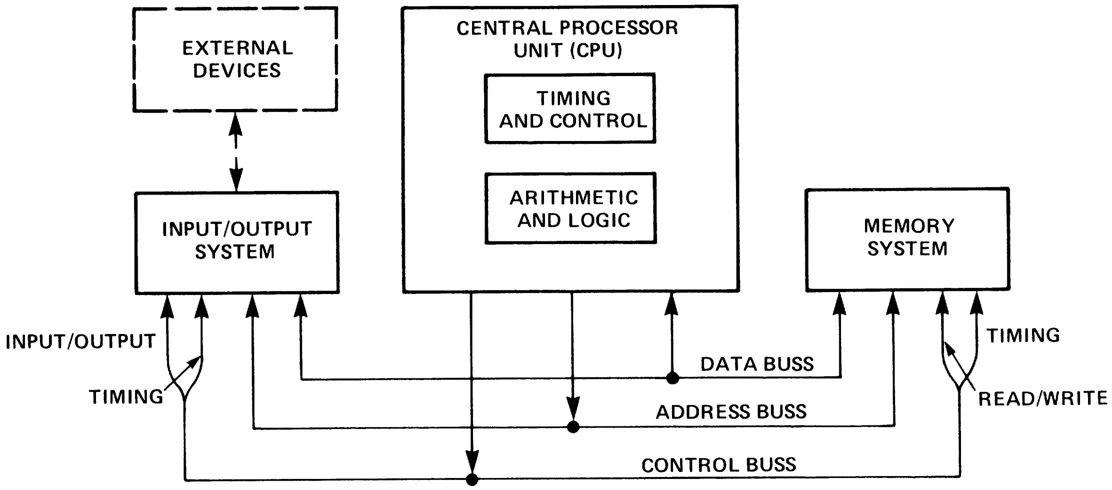


Figure 4.6 COMPUTER COMMUNICATIONS PATHS

A buss is a collection of several electrical conductors (wires) across which information may be transmitted. There are three primary busses in most computer systems: the data buss, the address buss, and the control buss.

The function of the data buss is to move bytes of data from one block to another. In your Instructor 50 the *data buss* consists of eight separate conductors, each carrying one bit of information. Thus, at any one time, a complete byte of information may be transmitted via the *data buss*. Information flow on the *data buss* may be either from the Central Processing Unit to the memory or to the input/output system or from the memory system or input/output system to the Central Processing Unit. This buss is thus said to be bi-directional in nature.

The address buss in your Instructor 50 consists of 15 conductors. Each conductor carries one bit of information. These 15 bits considered as a binary number can specify 2^{15} or 32,768 individual addresses. The binary value transmitted on the *address buss* determines the source or destination of data travelling across the data buss. For instance, the Central Processor Unit may store data in memory system location 15 by placing the number 15 on the *address buss* and the data to be stored on the data buss.

The control buss is a collection of miscellaneous signals which transmits various timing and control information from the Central Processor Unit to the other parts of the system. The *address buss* and the *control buss* are uni-directional in nature. That is, they transmit information only in the direction from the Central Processor Unit to the memory system or input/output system.

To see how all of this works in practice let's look at a couple of examples. Let us assume an instruction has been interpreted which calls for the Central Processor Unit to store the data byte $B3_{16}$ in memory location 5_{16}. The Central Processor Unit then places the data B3 on the *data buss* and the address $ØØØ5_{16}$ on the *address buss*. On the *control* *buss* a logic signal which tells the memory whether the operation is to be a read or write, that is, move data to the CPU from memory or to memory from the CPU, is placed in the write condition. When all of these signals are in readiness, a timing pulse, a logic signal which makes a short transition from a logic Ø to a logic 1 and back to logic Ø again, is issued on the timing control line. When the *timing pulse* occurs the memory system performs the operation indicated by its other inputs, i.e., it stores the data B3 (found on the *data buss*) in location 5 (as specified by the *address buss*.)

Later on another instruction may specify that the CPU is to retrieve the data previously stored in memory location 5. This is done by placing the address ØØØ5$_{16}$ on the *address buss* and placing the *read/*$\overline{\textit{write}}$ *control signal* in the read position. The CPU then issues the *timing pulse* on the *control buss* causing the memory system to place the contents of the location specified by the *address buss* (in this case location 5) onto the *data buss*, (in this case B3). The CPU then transfers the data on the *data buss* (B3) into one of its internal registers and removes the *timing signal*. When the *timing signal* is removed the memory system disconnects itself from the *data buss*.

Transfers of data between the CPU and input/output system work in an analogous fashion. The differences are that instead of specifying a location in memory on the address buss, the address now specifies a particular input/output port. And rather than a read/$\overline{\text{write}}$ signal on the control buss the control signal is an input/$\overline{\text{output}}$ signal.

Software vs. Firmware

In the discussion of the memory system above we described the memory system as a series of registers providing locations in which data could be stored for future retrieval. We also saw how data was stored in memory and retrieved from it via the data, address, and control busses. What we were describing was random access memory, abbreviated RAM. Any location may be accessed at random, and either read from or written into. We also mentioned that a program was nothing more than a sequence of binary numbers stored in memory locations. Under some circumstances it is not desirable to store programs in this type of memory because one of the instructions could be overwritten accidentally and lost. This is doubly true since *random access memory* has a characteristic of losing its data when power is disconnected. These problems can be solved by placing permanent programs in read-only memory, or memory which can be read from but not written into. *Read-only memory* is abbreviated as ROM. In a *read-only memory* the data in each memory location is permanently stored at the time of manufacture.

In your Instructor 50 when power is first applied, the message "Hello" is always displayed on the eight digit display. When you push a button some specific activity occurs. The reason your Instructor 50 responds this way is because it has been pre-programmed at the factory to perform these functions. This program, called the USE Monitor Program, is stored in *read-only memory* in your Instructor 50. It is this program which recognizes what buttons you have pushed and displays appropriate messages. Altogether your Instructor 50's Monitor Program occupies about 2000 memory locations. When a computer program is stored in *read-only memory* it is often referred to as firmware, that is, a *program* (software) stored in *read-only memory* (hardware).

Organization of the 2650 CPU

Earlier we discussed the general way in which a Central Processor Unit is organized. In this section we will look at the specific organization of the 2650 microprocessor used in your Instructor 50. From a user's point of view we may describe this organization completely by simply describing the registers contained in the Central Processor Unit and how each of them behaves. It is understood that a great deal of logic is required in the Central Processor Unit in addition to these registers to make them behave in the way described. The function of this additional logic, however, is simply to manipulate data and move it from one register to another. With this in mind, refer to the CPU register diagram of

Figure 4.7. This diagram does not show all the registers within the 2650 Central Processor Unit but it shows those that are important to our discussion. Other special purpose registers will be described later as they are needed in examples. (For a complete description of 2650 CPU registers see Chapter 9 of your User's Guide.)

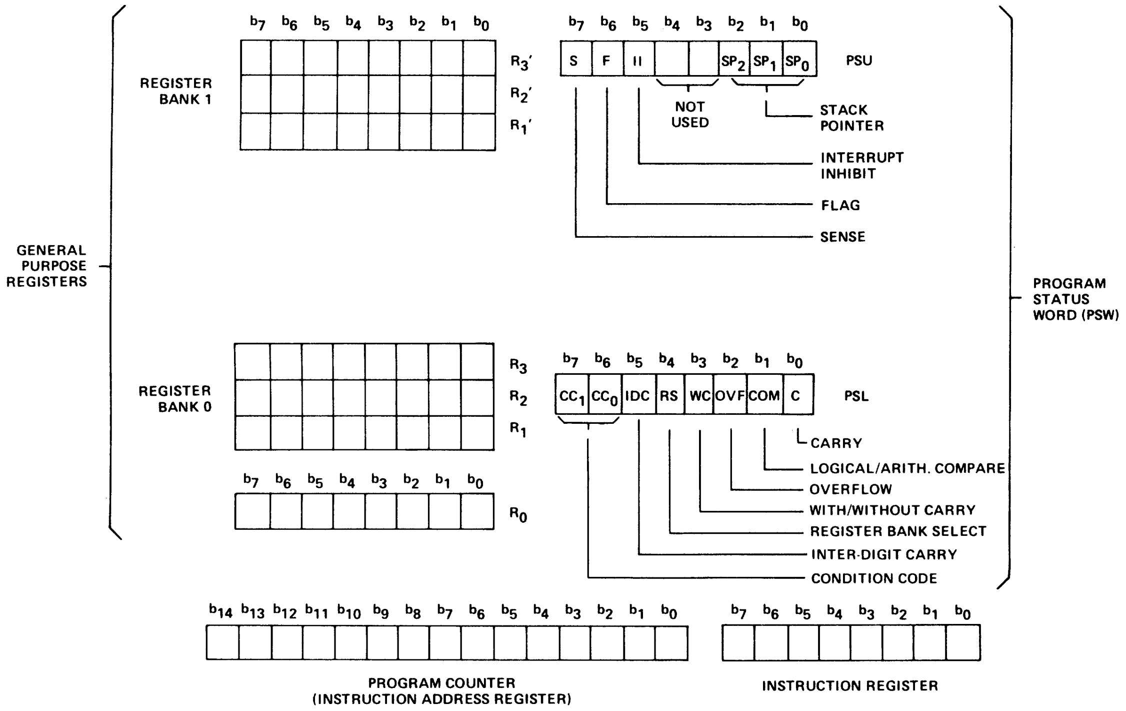


Figure 4.7 2650 CPU REGISTERS

In programming your Instructor 50 the register with which you will deal most often is Register Ø. This is an eight bit register and is called a General Purpose Register because it may be used for a number of purposes, such as temporarily storing data retrieved from or to be placed into memory. Or it may serve as a place to store the result of arithmetic or logical computations. Register Ø may also be correctly be called an *accumulator*, because it often accumulates (stores) the results of arithmetic calculations. It is possible to examine and alter the contents of this register through the keyboard and display on your Instructor 50. To see how this is done perform the following exercise:

1. Apply power or press the MON key to obtain the message "Hello" on the Instructor 50 display.

2. Press the Register Function button, REG, to obtain the message "r = "

3. Press the numerical button Ø. The display now reads ".rØ = XX," where XX is the current contents of Register Ø displayed in hexadecimal notation.

4. You may now change the contents of Register Ø by pressing any two numerical keys in sequence. For instance, if you wish to change the contents of Register Ø to $5A_{16}$ press 5 then A.

5. If you now again press REG you'll again see the message "r = "

6. Press the numerical key Ø. You will now see the message "rØ = 5A," the number you previously stored in Register Ø.

There are six more *general purpose registers* in your Instructor 50 organized as two banks of three registers each. Bank Ø contains Registers 1, 2, and 3. Bank 1 contains Registers 1', 2', and 3'.

Registers 1, 2, and 3 may be examined and altered by following the method described above except that instead of specifying RØ you must specify R1, R2, or R3. Registers 1', 2', and 3' may be similarly accessed from the keyboard by specifying Registers R4, R5, and R6, respectively.

You may at this point wonder why Registers 1, 2, and 3, and 1', 2', 3' are considered to be two banks of three registers each. While all seven of these registers are equally accessible from the keyboard, only RØ is always accessible by a 2650 instruction. A 2650 instruction may specify only RØ, R1, R2, or R3. Whether the R1, R2, or R3 referred to in the instruction comes from bank Ø or bank 1 is determined by the setting of a single bit in the Program Status Word (PSW).

The Program Status Word

The Program Status Word (PSW) is contained in a 16-bit register shown on the right hand side of Figure 4.7. For convenience, this 16-bit word is considered to be two separate 8-bit bytes, labeled PSU (Program Status Upper) and PSL (Program Status Lower). Each bit of the *Program Status Word* has significance for certain operations of the 2650. Bit 4 of PSL is the Register Bank Select bit. When this bit is a Ø, 2650 instructions specifying Register R1, R2, or R3 will act upon Register Bank Ø. When the Register Bank Selected bit is a 1, 2650 instructions referring to R1, R2, or R3 will operate on the registers in Register Bank 1.

The contents of the *Program Status Word* are also accessible from the function keys on your Instructor 50. The key sequence REG, 7 will display the contents of the *Program Status Word Upper* byte. These contents may then be altered as described above. Similarly, the key sequence REG, 8 will display the contents of PSL, the lower byte of the *Program Status Word*. Its contents may also be altered as described above. Try this out on your Instructor 50 keyboard. While each bit in a *Program Status Word* has a specific function significant to operation of the 2650 it is not necessary to describe each of these in detail here. This is done in Chapter 9 of your User's Guide.

The Instruction Register shown in Figure 4.7 may be considered part of the Timing and Control section of the Central Processor Unit. As we mentioned earlier each instruction consists of a binary code which is normally stored in memory until it is needed. When an instruction is needed it is temporarily stored in the *Instruction Register* of the Central Processor Unit, where the instruction code is available to the timing and control logic. The *Instruction Register* is not accessible from your Instructor 50 keyboard.

The Program Counter, also known as the Instruction Address Register, is another special purpose register of the 2650 microprocessor. Its function is to hold the memory address of the instruction which will be needed next by the timing and control section.

Since, as we mentioned earlier, the 2650's address buss is 15 bits in width, this must be a 15-bit register. It is accessible from the Instructor 50 keyboard by pressing the key sequence REG, C. When this key sequence is pushed the contents of the *Program Counter* is displayed as a 4 digit hexadecimal number. Its contents may also be altered from the keyboard as described above.

HOW IT ALL WORKS TOGETHER

In actual operation all of these registers perform their functions in a carefully timed sequence, much as the individual component parts of an automobile, the pistons, transmission, wheels, and so on, all operate together to move the automobile forward. Recall that we described a computer program as a sequence of instructions stored in sequential locations in the memory system.

Instruction is Fetched

Before any instruction can be carried out, it must be brought to the Central Processor Unit. This operation is known as fetching the instruction (from memory). Here is how it works.

1. As we said earlier the Program Counter contains the address (location in memory) of the next instruction to be *fetched*.

2. When it is time to *fetch* an instruction from memory the Timing and Control system connects the *Program Counter* to the address buss (refer to Figure 4.6).

3. The Timing and Control logic also specifies that a read operation is to be performed by placing the read/$\overline{\text{write}}$ control line to memory in the read position.

4. Next a *timing pulse* is issued to the memory system. The memory system responds by placing on the data buss the contents of the memory

location specified by the address buss (which is also the location specified by the Program Counter).

5. The Timing and Control logic then connects the data buss to the *Instruction Register*, where the code which appears on the data buss is stored.

6. Next the Timing and Control logic disconnects from the address buss and control busses.

7. The memory system disconnects itself from the data buss. While this is happening the value in the *Program Counter* is incremented by 1.

This completes the *fetch* operation. At this point the binary code for the instruction to be carried out is contained in the *Instruction Register*. The *Program Counter* has been incremented by 1, and thus contains the address of the next location in memory beyond that from which the current instruction was *fetched*.

Instruction is Executed

Once an instruction has been fetched from memory it may be executed. To *execute* an instruction simply means to carry it out or to perform the operation indicated by its binary code. To do this the Timing and Control logic examines the 1s and Øs contained in the Instruction Register. Based on the combination found, it issues timing pulses to other sections of the computer. In this manner the arithmetic and logic unit, memory system, or input/output system is manipulated to perform the required operation. When the instruction has been *executed*, the contents of the Program Counter are again placed on the address buss and another fetch cycle occurs. A new instruction from the next location in memory is transferred to the Instruction Register and is subsequently *executed*.

When a sequence of instructions (*program*) is *executed* in this fashion, we may say that we have *executed the program*.

Example: A One Instruction Program

Now that you've learned about the various functions and registers in the Instructor 50 you are ready to actually execute a simple program. Before the program can be executed, however, it must be entered (into memory) by pressing appropriate keys on the Instructor 50 keyboard.

The particular instruction that we will execute in this example is abbreviated "ADDZ R1." This is an abbreviated way of saying, "add to Register Ø the contents of Register 1." When this instruction is *executed*, the contents of Register Ø and Register 1 will be added together and the result will be stored in Register Ø. The hexadecimal code for this instruction is 81_{16}. To enter and execute this instruction follow the sequence indicated below.

Note: It is important that you not press the MON button except as specified during the key sequence indicated below. If you do, the contents of some internal registers may be altered and you will have to start again.

1. Press MON button to obtain the "Hello" message.

2. Press the sequence MEM, Ø, ENT/NXT, 8, 1. This enters the hexadecimal code for our instruction (81_{16}) in memory location Ø.

3. Next, initialize the value of Register Ø to a 1 by pressing the sequence REG, Ø, 1.

4. Initialize the contents of Register 1 to a 2 by pressing the key sequence, ENT/NXT, 2.

5. Initialize the contents of PSU to FF by pressing the sequence REG 7, F, F.

6. Initialize the contents of PSL to ØØ by pressing the key sequence ENT/NXT, Ø.

7. Initialize the Program Counter to Ø by pressing the key sequence REG, C, Ø.

Now everything is in readiness to execute your program. If you like, you may go back and examine the contents of all of the registers that you just initialized. But be careful not to press the MON button or the contents of some registers may be changed.

8. To execute your program push STEP. The display should now read "ØØØ1 XX," with XX indicating the contents of memory location 1. Again without pushing MON, let's go back and examine the contents of the various 2650 registers to see what effect our instruction had.

Since our instruction was supposed to add the contents of Register Ø and 1 and store the result in Register Ø, and since Register Ø initially contained 1 and Register 1 a 2, we should now find a 3 (1 + 2) stored in Register Ø. To find out if this is true, press the key sequence REG, Ø. The display should read ".rØ = Ø3."

Register 1 initially contained a 2. Since we've done nothing to alter this the 2 should still remain. To find out if it does press ENT/NXT. The display should now read ".r1 = Ø2." Finally, the Program Counter was initially a Ø. Recall that we said that the Program Counter is automatically incremented by 1 during execution of an instruction. To see if it contains a 1, press the key sequence REG, C. The display should read ".PC = ØØØ1." The instruction code 81 contained in memory location Ø should also still be there. To find it, press the key sequence MEM, Ø, ENT/NXT. The display should read ".ØØØØ 81." We may surmise that our instruction code 81 has indeed been fetched from memory by the Central Processor Unit. The contents of Registers Ø and 1, a 1 and a 2 respectively, have been added together to produce the result 3 stored in Register Ø. The Program Counter which was initally set to Ø (specifying the address of our instruction code) has now been incremented to contain a 1. Try executing this program again with different initial values in

Register Ø and Register 1, and verify that their sum is indeed stored in Register Ø by your program.

Congratulations! If you have followed the text so far, you've just executed your first computer program. From here on, it's all downhill.

Example: A Two Instruction Program

In this example we double the complexity of your program by adding another instruction. The instruction we will add is abbreviated "WRTD," which is short for "Write Data." This instruction transfers the contents of Register Ø to the "non-extended data port" of your Instructor 50. The 8 LEDs toward the left hand side of your Instructor 50 operator's panel may be connected to this output port, by placing the 3 position switch in the lower left hand corner to the full downward position labeled "NON-EXT Data Port." The binary code for the WRTD instruction is FØ. Thus, our two instruction program will read as follows.

ADDRESS	CODE	INSTRUCTION
ØØØØ	81	ADDZ RØ
ØØØ1	FØ	WRTD

To enter and execute this program follow the steps below.

1. As in the previous example enter the code for the ADDZ RØ instruction into memory location Ø by pressing the key sequence MEM, Ø, ENT/NXT, 8, 1.

2. Enter the code for the WRTD instruction into memory location 1, by pressing the sequence ENT/NXT, F, Ø.

3. This time let's initialize Register Ø to a 3 with the key sequence REG, Ø, 3.

4. Register 1 may be initialized to a 2 with the key sequence REG, 1, 2.

5. As before, we must initialize the Program Status Word with the key sequence REG, 7, F, F, ENT/NXT, Ø.

6. Finally, we initialize the Program Counter to Ø with the key sequence REG, C, Ø.

7. Now push the STEP button once. Notice that the display reads "ØØØ1 FØ." This means that the Program Counter is "pointing at" memory location 1 which contains the code for our WRTD instruction, FØ. Since the contents of Register Ø and Register 1 have now been added together and since they contained a 3 and a 2 respectively, the contents of Register Ø should now be a 5. You may verify this by pressing the key sequence REG, Ø.

8. Now press the STEP button again to execute the WRTD instruction. Note that the code ØØØØØ1Ø1 is displayed on the 8 LEDs connected to the non-extended data port. You should also be able to verify that Register Ø contains a 5, Register 1 contains a 2, and the Program Counter (Register C) contains a 2.

The STEP button is a very useful function of your Instructor 50. It allows you to execute instructions one at a time. After the STEP button is pressed and the instruction is executed, the display will indicate the contents of the memory location "pointed to" by the Program Counter. Since the contents of this location are always the next instruction that will be executed when you push the STEP button again, this provides a convenient method to verify that each instruction code is as it should be while you are stepping through a program.

TYPES OF INSTRUCTIONS (WHAT INSTRUCTIONS CAN DO)

In the above examples you saw two specific instructions and the operations they performed. Your Instructor 50 has a "vocabulary" of 75 such instructions, each of which performs a specific simple task. Each of these instructions is described in detail in your Instructor 50 User's Guide. Depending on the type of operation that a specific instruction performs it may be thought of as belonging to a class or type of similar instructions. These types are described briefly below.

Arithmetic Instructions

Arithmetic instructions are instructions which perform an *arithmetic operation* such as the ADDZ instruction described above. These instructions may *add* the contents any two registers or *subtract* the contents of one register from another.

Input/Output (I/O) Instructions

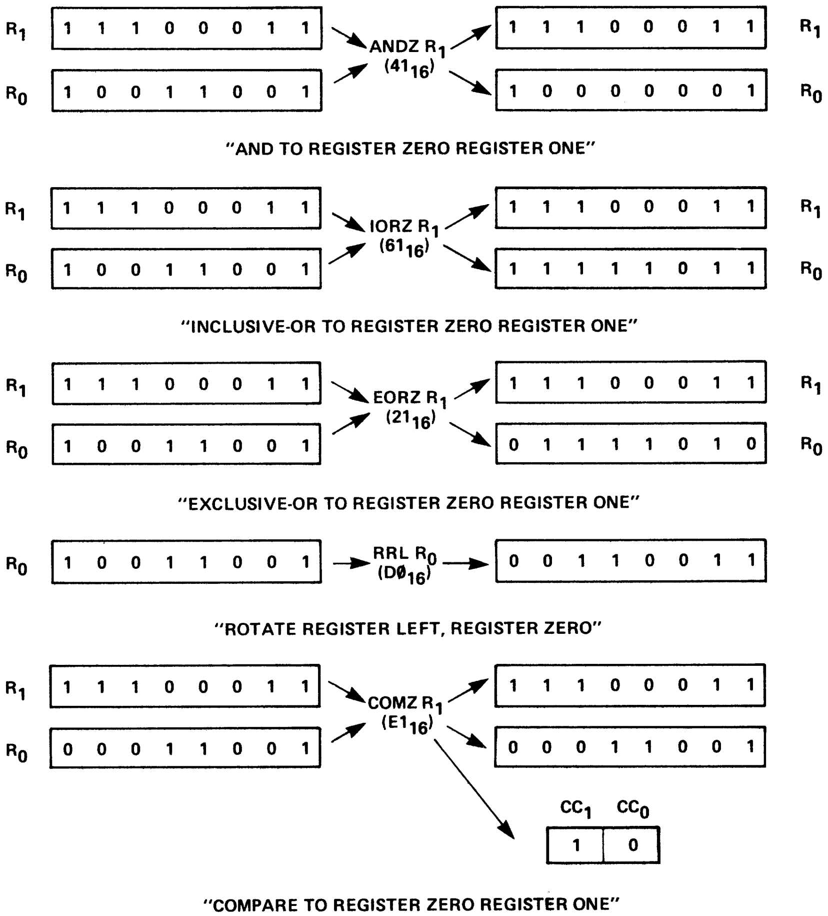
Input/output instructions are used for getting data into or out of the computer. They operate by copying a byte of data from an *input port* to a register or from a register to some *output port*. The WRTD instruction described above is an example of an *output instruction*.

Load and Store Instructions

Load and Store instructions are used to copy a byte of data in one location or register into another location or register. A register is said to be *loaded* when a byte of information is copied into it from some memory location or from some other register. The contents of a register are said to be *stored* when its contents are copied into a memory location or into some other register. *Load and store instructions* are somewhat similar to *input/output instructions* except that the operations are completely internal.

Logical Instructions

Logical instructions are similar to arithmetic instructions except that they perform *logical operations* such as AND, OR, Exclusive-OR, Rotate, and Compare. Examples of each of these types of instructions are given below, in Figure 4.8.



"AND TO REGISTER ZERO REGISTER ONE"

"INCLUSIVE-OR TO REGISTER ZERO REGISTER ONE"

"EXCLUSIVE-OR TO REGISTER ZERO REGISTER ONE"

"ROTATE REGISTER LEFT, REGISTER ZERO"

"COMPARE TO REGISTER ZERO REGISTER ONE"

Figure 4.8 EXAMPLES OF LOGICAL INSTRUCTIONS

The operators AND, OR (also known as Inclusive-OR) and Exclusive-OR are straightforward and require no further explanation. The operation of the Rotate and Compare instructions deserves further comment.

The operation of the Rotate Register Left instruction is affected by a bit in the Program Status Word, labeled WC or With Carry, in Figure 4.6. If this bit (bit 3 in the lower byte of the Program Status Word) is a Ø, the operation of the *Rotate Register Left RØ instruction* is as shown in Figure 4.8. That is, each bit of Register Ø is moved to the left one position with the most significant bit being rotated around and occupying the least significant position. If, however, the *With Carry* bit in the Program Status Word is set to a 1, the most significant bit is not shifted into the least significant bit position. Instead, it is shifted into the *Carry* bit which is another bit of the Program Status Word. The *Carry* bit in the PSW is shifted into the least significant position of Register Ø.

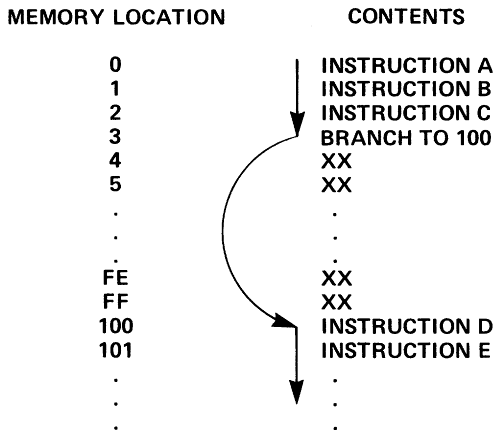
Operation of the *Compare instruction* involves three more bits in the PSW. When the *Compare instruction* is executed, neither of the two registers whose contents are being *compared* is altered. However, the contents of the two Condition Code bits in the PSW, CCØ and CC1, are set with a value which indicates whether the contents of Register 1 was greater than, equal to, or less than the contents of Register Ø. In the example shown, the contents of Register 1 is greater than that of Register Ø if the two are considered as straight binary numbers. Therefore, the Condition Code bits are set to 1 and Ø as shown.

In this example it was assumed that the COM bit in the PSW was a Ø. This is the logical/arithmetic Compare bit. If this bit were a 1, the contents of Register Ø and Register 1 would have been interpreted by the Compare instruction as binary *two's complement* numbers. Since the most significant bit of Register 1 in the example is a 1, this represents a negative number. The contents of Register Ø represents a positive number. Thus considered as two's complement numbers, the contents of Register Ø is greater in value than the contents of Register 1, and the resulting Condition Code setting would be Ø1 rather than 1Ø.

A Compare operation is particularly useful when executed just before a branch instruction, which is described below.

Branch Instructions

Normally, the sequence of instructions for a program is drawn from sequential locations in memory. As each instruction is executed, the Program Counter is incremented by 1. Thus the Program Counter keeps track of the location in memory from which the next instruction is to come. The function of a branch instruction is to alter the contents of the Program Counter. The effect of doing this is that the next instruc tion for the program will not be taken from the next sequential location in memory but will be taken instead from an address which is specified by the *branch instruction*. The operation of a *branch instruction* is illustrated in Figure 4.9.



MEMORY LOCATION	CONTENTS
0	INSTRUCTION A
1	INSTRUCTION B
2	INSTRUCTION C
3	BRANCH TO 100
4	XX
5	XX
.	.
.	.
.	.
FE	XX
FF	XX
100	INSTRUCTION D
101	INSTRUCTION E
.	.
.	.
.	.

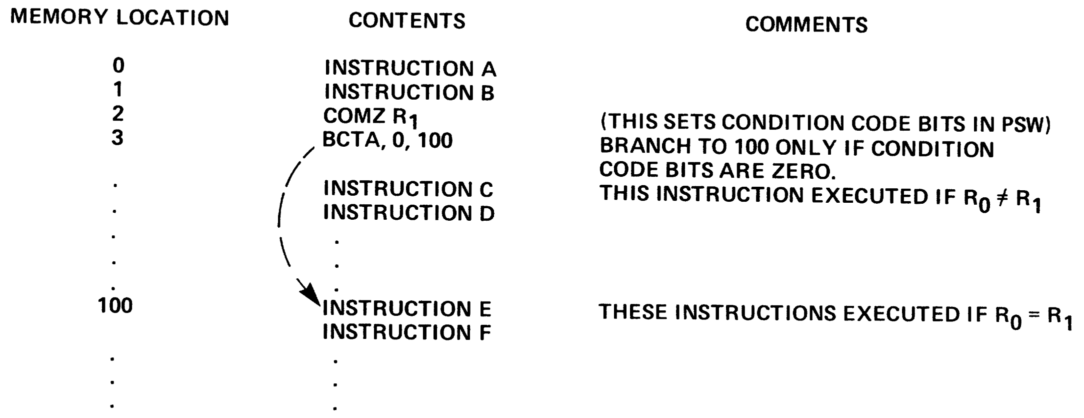
Figure 4.9 OPERATION OF BRANCH INSTRUCTION

We begin executing our program, for instance, at memory location Ø. Instructions are executed in sequence until the *branch instruction* is encountered. When the *branch instruction* is encountered the next instruction is taken not from the next sequential location in memory but from the location specified by the *branch instruction*, location 1ØØ in the example. From·this point instructions are again executed in sequence.

In many cases it is desirable for the program to *branch* to another location in memory only if some condition is met. An instruction to perform this operation (alter the contents of the Program Counter only if some condition is satisfied) is called a conditional branch instruction. In your Instructor 50 the Condition Code bits (CCØ and CC1) in the PSW may be examined by a *branch instruction* to see if a specified condition is satisfied.

For instance, let us say that we wish to compare the contents of Register 1 and Register Ø and *branch* to another location in memory only if the value in Register 1 is equal to the value in Register Ø. This would be done as illustrated in the Figure 4.10. Again, we assume that our program starts at memory location Ø and instructions A and B are executed.

We then *compare* the contents of Register 1 to those of Register Ø with the COMZ R1 instruction. The result of this instruction as discussed above is to set the Condition Code bits in the PSW. These two bits will both be set to Ø if Register 1 and Register Ø contain equal values.



MEMORY LOCATION	CONTENTS	COMMENTS
0	INSTRUCTION A	
1	INSTRUCTION B	
2	COMZ R_1	(THIS SETS CONDITION CODE BITS IN PSW)
3	BCTA, 0, 100	BRANCH TO 100 ONLY IF CONDITION CODE BITS ARE ZERO.
.	INSTRUCTION C	THIS INSTRUCTION EXECUTED IF $R_0 \neq R_1$
.	INSTRUCTION D	
.	.	
.	.	
.		
100	INSTRUCTION E	THESE INSTRUCTIONS EXECUTED IF $R_0 = R_1$
	INSTRUCTION F	
.	.	
.	.	
.	.	

Figure 4.10 OPERATION OF CONDITIONAL BRANCH INSTRUCTION

The next instruction is the BCTA or Branch on Condition True Absolute instruction. The *branch* will occur only if the contents of the Condition Code bits are ØØ. If the contents of Register 1 are equal to the contents of Register Ø when the *Compare instruction* occurs, instruction E will be executed immediately following the *branch instruction*. If the *branch* does not occur, that is, if the contents of Register 1 did not equal the contents of Register Ø, instructions C, D, and so on will be executed. *Branch instructions* are also sometimes called jump instructions because they cause the normal execution of the program to *jump* from one location in memory to another.

Multi-Byte Instructions

In our discussions so far, we have assumed that each computer instruction occupied one byte in memory. Some instructions, however, require that more information be specified than the amount which can be contained in a single byte. For instance, the *Branch on Condition True Absolute* instruction, discussed in the previous example, must not only specify that a branch must occur if some condition is satisfied, but it must also specify the value of the Condition Code bits and an address from which the next instruction is to be fetched if the specified condition is satisfied. Since the address requires fifteen bits of information it is specified in two successive bytes following the first byte of the *Branch on Condition True Absolute instruction*. This instruction therefore occupies three sequential bytes in memory. Its format is illustrated in Figure 4.11.

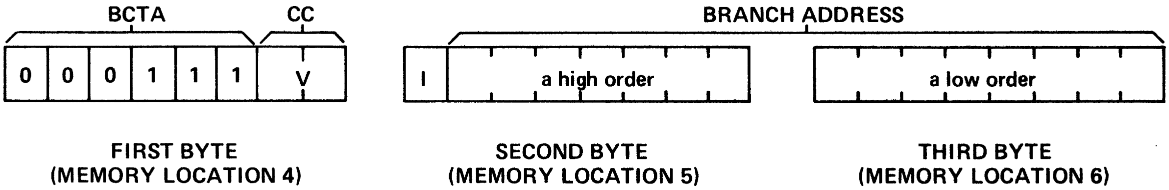


Figure 4.11 FORMAT OF BCTA INSTRUCTION

This type of instruction is executed in a similar fashion to the single byte instructions discussed above, except that three fetch cycles are required before the execution cycle. This works as follows: In the first fetch cycle the first byte of the instruction is fetched from memory and put into the CPU's Instruction Register in the normal fashion. The Timing and Control logic of the CPU then examines the combination of 1s and 0s in the first byte of the instruction and determines that two additional bytes must be fetched. Since the Program Counter has been incremented automatically during the fetch cycle, it already contains the address of the second byte of this instruction. The second byte is then fetched and placed in an auxiliary register in the Timing and Control section of the CPU. While this fetch cycle is taking place the Program Counter is again incremented so that it now contains the address of the third byte of the instruction. The third byte is fetched and placed in another auxiliary register in the Timing and Control logic. At this point, the entire instruction has been moved into the Central

Processor Unit and the Program Counter contains the address of the next instruction following the branch instruction.

The two least significant bits in the first byte of the instruction are compared with the two Condition Code bits in the PSW to determine if their settings are equal. If they are equal the address specified by the second and third byte of the BCTA instruction is placed in the Program Counter. Thus the next instruction is fetched from that location. If the settings of the Condition Code bits do not match those specified by the two least significant bits of the first byte of BCTA instruction the Program Counter is not altered and the next instruction is fetched sequentially from memory. In every case for a multi-byte instruction the first byte contains enough information to identify the type of instruction and the fact that additional bytes must be fetched. Allowing instructions to be two or three bytes in length permits a great deal of flexibility in specifying the source of information on which the instruction is to operate and its destination.

Addressing Modes

The way in which the source or destination of information is specified by an instruction is called an addressing mode. There are four basic addressing modes which we will describe below in detail. These are called Register, Immediate, Absolute, and Relative.

1. Register Addressing: An instruction which specifies internal 2650 registers as both the source and destination of data is said to use the *register addressing* mode. These instructions require only one byte of information. For example, the format of the *Add to Register Zero (ADDZ)* instruction, is shown in Figure 4.12.

In the *Add to Register Zero instruction*, the fact that an addition is to take place whose contents will be stored in Register Ø is specified by the most significant six bits of the instruction. The least significant two bits specify which internal register will be added to Register Ø to produce the result. For instance, if the Register field (as shown in Figure 4.12) contains 1Ø then the contents of Register 2 will be added to Register Ø and the results stored in Register Ø. (1Ø$_2$ is the binary equivalent of 2$_{1Ø}$).

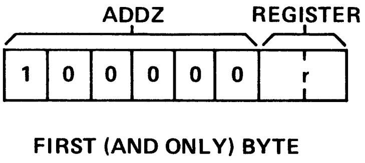


Figure 4.12 FORMAT OF ADDZ INSTRUCTION--REGISTER ADDRESSING

2. Immediate Addressing: An instruction is said to be in the *immediate addressing* mode if the actual data to be operated on is contained in the instruction. These instructions require two bytes of information. As an example, the format for the *Add Immediate instruction* is shown in Figure 4.13.

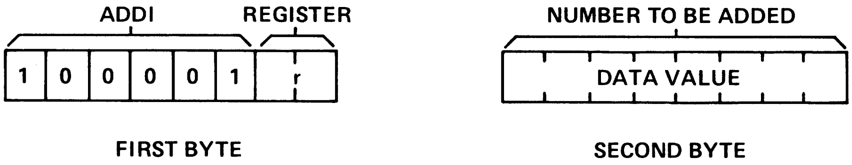


Figure 4.13 FORMAT OF ADDI INSTRUCTION--IMMEDIATE ADDRESSING

When this instruction is executed, the first byte of the instruction will first be fetched and placed in the Instruction Register in the CPU. The most significant six bits of this byte indicate that the operation to be performed is *Add Immediate*. The two least significant bits designate the Register (RØ, R1, R2 or R3), where the result will be stored.

After the Timing and Control logic obtains this information, it fetches the second byte of the instruction from memory. This byte contains an immediate value to be added to the contents of the register specified by the first byte. For example, let us say that the R field in the first byte contains a Ø1 specifying Register 1, and that before execution of this instruction the contents of Register 1 in the CPU is 5. Let's also assume that the second byte of the instruction is ØØØØØØ11 or $3_{1\emptyset}$. After execution of this instruction the contents of Register 1 in the CPU will be 8. That is, the $5_{1\emptyset}$ currently in Register 1 was added to the $3_{1\emptyset}$ specified by the second byte of the instruction to produce the result, $8_{1\emptyset}$, stored in Register 1.

3. Absolute Addressing: An instruction is said to be in the *absolute addressing* mode if it requires data stored in a specific location in memory. This type of instruction requires three bytes, one to specify the type of instruction and two to specify the specific address in memory where the required data will be found. The BCTA instruction discussed above and whose format is shown in Figure 4.11, is an example of an *absolute addressed* instruction.

4. Relative Addressing: An instruction using the *relative addressing* mode is similar to an instruction using absolute addressing in the sense that it requires data from some specific location in memory. Instead of specifying this address absolutely however, the address is specified as a position relative to the current contents of the Program Counter. To better understand the concept of *relative addressing*, consider the format of the Add Relative (ADDR) instruction shown in Figure 4.14. In the Add Relative instruction the most significant six bits of the first byte indicate to the Timing and Control logic that the operation to be performed is Add Relative. The least significant two bits of the first byte indicate which register is to be the destination of (hold the results of) the operation. The second byte of the instruction specifies a number which is to be added to the current contents of the Program Counter to determine the location in memory which contains the data to be added to the specified register.

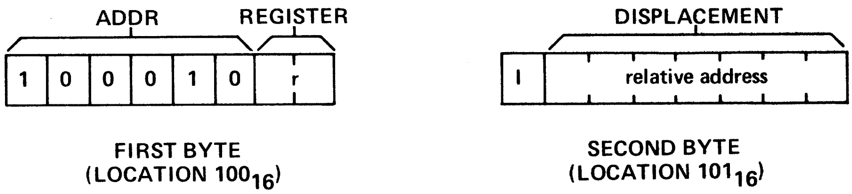


Figure 4.14 FORMAT OF ADDR INSTRUCTION--RELATIVE ADDRESSING

For instance, let us assume that the *relative address* of this instruction is a 5. Also assume that the instruction itself is stored at location 1ØØ in memory, and that this instruction has just been fetched. The Program Counter will contain 1Ø2, the address of the next memory location beyond this instruction. Thus, the contents of the Program Counter 1Ø2 will be added to the displacement, 5, to obtain the result 1Ø7. Data to be added to Register R then will be obtained from location 1Ø7$_{16}$. This is somewhat more complicated than the absolute addressing method shown above. Its advantage lies in that it requires only two bytes to specify an address rather than three. The displacement value is considered to be a 7 bit two's complement number, and thus may take on values from +63$_{1Ø}$ to -64$_{1Ø}$.

5. Indirect Addressing: In addition to the four basic addressing modes described above, *Register, Immediate, Absolute, and Relative*, there are two variations which may be used under some circumstances. These are called indirect addressing and indexed addressing. In your Instructor 50, *indirect addressing* may be specified for instructions in the relative and absolute addressing modes. *Indirect addressing* is specified by the most significant bit of the second byte of these instructions, labeled "I" in Figures 4.11 and 4.14 above. When *indirect addressing* is specified, the absolute address or relative address specified by the instruction does not contain the data to be operated upon. Instead, it contains the first byte of the *address* of the data to be operated upon. The second byte of the address of the data to be operated upon is found in the next sequential location in memory. For our example of Figure 4.14 above, if the Indirect bit is a 1, data would be obtained not from address

1Ø7 in memory but from the address in memory specified by the contents of locations 1Ø7 and 1Ø8. While this is a bit complicated, it turns out to be useful in some programming applications.

6. Indexed Addressing: Indexing is another address modification technique which may be used with some instructions in the absolute addressing mode. It may be illustrated by considering the format of the Add Absolute instruction as shown in Figure 4.15.

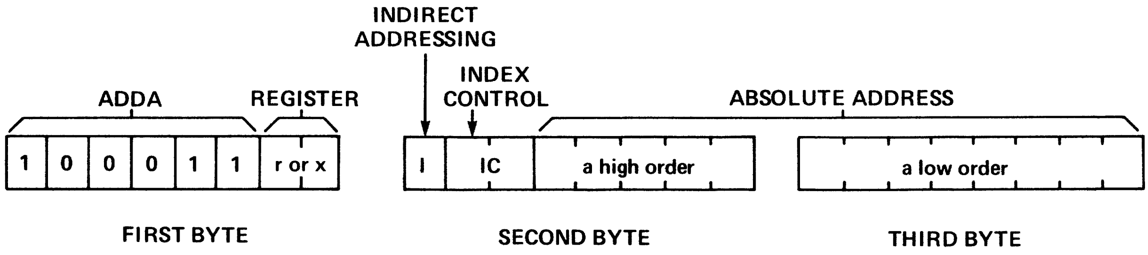


Figure 4.15 FORMAT OF ADDA INSTRUCTION
ABSOLUTE ADDRESSING WITH INDEXING

In this instruction the first byte specifies that the instruction is to be an *Add Absolute* operation. Let's first consider the case where both the *Indirect Addressing* and *Index Control* bits are zeroes. In this case, the register which will contain the result of the operation is specified by the least significant two bits of the first byte. The address in memory from which data is to be obtained to add to this register is specified by the *Absolute Address* section of the second and third bytes. Since only thirteen bits of address information may be specified and the 2650 address buss is fifteen bits wide, it is assumed that the most significant two bits of the address are the same as the most significant two bits of the address in which this instruction itself is located.

In the event that *Indirect Addressing* is specified (by placing a 1 in the most significant bit of the second byte of the instruction) the contents of the location specified by the *Absolute Address* contains the first byte of the address where the data to be added to the specified register is stored. The second byte of this address is found in the next sequential location. If the *Index Control* bits are non-zero, the two least significant bits of the first byte no longer specify the register in which the results of the instruction will be stored. Instead they specify a register to be used for indexing purposes, or an *Index Register*. It is then assumed that the results of the operation will be stored in Register Ø.

When *Indexing* is specified (by having one or both of the *Index Control* bits set to a 1,) an Effective Address is formed by adding the contents of the specified *Index Register* to the *Absolute Address* specified in the second and third bytes of the instruction. For instance, let us assume an example in which both *Index Control* bits are 1s, the register specified as an *Index Register* is Register 3, the *Absolute Address* portion of the instruction contains a 5, and the contents of Register 3 is a 1. We now execute the ADDA instruction. The data to be added to Register Ø will be obtained from location 6 (= 5 + 1). The data contained in location 6 will be added to the data contained in Register Ø, and the result will be stored in Register Ø.

While all this is happening, it may be desirable to automatically increment or decrement the specified *Index Register*. The *Index Control* bits are set to Ø1 to accomplish this, instead of being set to 11. The contents of the specified *Index Register* will be incremented before the operation is performed. In our example, the contents of Register 3 would become 2. If the *Index Control* bits are set to 1Ø, the contents of the specified *Index Register* will be decremented before the operation is performed. In our example, the contents of Register 3 would become Ø. This is also complicated but is useful in some programming applications.

Let's say, for example, that you've stored a table of data beginning at location 1ØØ in memory. You may use an *Index Register* to keep track of which entry in this table you wish to look at. If you set the *Absolute Address* portion of an *Indexed instruction* to 1ØØ$_{16}$, and specify the entry number within your table with an *Index Register*, this becomes a convenient method of accessing a particular entry within a table. Don't be too discouraged if this is not all obvious to you right now. As you gain more familiarity with programming and programming techniques, these concepts will become easier to grasp.

INTERPRETING INSTRUCTION DESCRIPTIONS

In the foregoing discussion we've spoken briefly about several instructions and their various formats. In Chapter 9 of your Instructor 50 User's Guide, you'll find detailed descriptions of the operation of each instruction which your Instructor 50 is capable of performing. In describing these instructions some notation conventions have been developed which make it easy and compact to represent certain ideas. For instance, a small "r" designates a particular register. It stands for Registers RØ, R1, R2, or R3 in the Instructor 50. Parentheses around a quantity designate the concept, "contents of." For instance, (RØ) means "the contents of Register Ø" or (r) means "the contents of Register r." A small "a" is often used to denote an address. Therefore (a) would be read, "the contents of memory location a." A small "v" is used to stand for a data value. A left-pointing arrow, is used to stand for "is replaced by." For instance, we might write

"(r) ⟵ (r) + v."

This would be read, "the contents of Register r is replaced by the contents of Register r added to the value v." Let us say, for example, that RØ contains a 5 and value v is a 7. If we wrote

(RØ) ⟵ (RØ) + v

it would mean in our example

(RØ) ⟵ 5 + 7 or (RØ) = ØC$_{16}$.

This type of notation is used in describing instructions in your User's Guide to make the instruction descriptions more compact and easier to refer to.

As an example, let's take a look at the description for the Add Absolute instruction, which you will find in your User's Guide described on page

37 of Chapter 9. On the top line of this page, you see in the left hand side the mnemonic for the instruction, "ADDA," followed by some other notation. On the right hand side you see a brief description of this instruction, in this case, Add Absolute. When writing instructions to be interpreted by humans, it is much more understandable to write them in *mnemonic* form rather than as hexadecimal or binary codes. After the instruction's *mnemonic* in this example we see, "r (*)a(,x)." The portions enclosed in parentheses are optional and may or may not be included in the notation for the instruction as it is written. The little "r" is meant to indicate a specific register. The little "a" is meant to indicate a specific address. For instance, if we wanted to specify adding the contents of location 5Ø$_{16}$ to the contents of Register 1 we would write "ADDA,R1 5Ø." This would specify that the contents of memory location 5Ø are to be added to the contents of Register 1, with the result stored in Register 1. Optionally, we could write, "ADDA,R1 *5Ø." The addition of the optional "*" would indicate Indirect Addressing in this example. This would mean that the contents of the location specified by the address contained in locations 5Ø and 51 would be added to the contents of Register 1, with the results stored in Register 1.

Also, we may optionally specify that *indexing* is to be used with this instruction. If we write "ADDA,RØ 5Ø, R1" this means that the contents of Register 1 is to be added to the specified address value, 5Ø, to calculate the *Effective Address* where the data to be added to RØ will be found. Thus we have specified Indexed Absolute Addressing. In the same shorthand, to denote that we would like to have the Index Register specified incremented or decremented before indexing occurs, we would write "ADDA,RØ 5Ø, R1+" or "ADDA,RØ 5Ø, R1-."

The next line tells us that this instruction occurs in the Absolute Addressing mode. The third line tells us that the operation codes possible for this instruction are 8C, 8D, 8E and 8F. This is a hexadecimal form of expressing the possible binary codes for the first byte of this instruction.

Below this we see the format for the binary code for the ADDA instruction, which is similar to that discussed in Figure 4.15 above. Notice that if the two least significant bits of the first byte of the instruction are ØØ, the hexadecimal notation for this binary code would be 8C. If the two least significant bits of the first byte were 11, the hexadecimal coding for the first byte of this instruction would be 8F.

The next line tells how much time it takes to execute this particular instruction. This is described in terms of *machine cycles*. In our example, four *machine cycles* are required. The first three *machine cycles* are to fetch the three bytes of the instruction. The instruction is then executed during the fourth *machine cycle*. Since each *machine cycle* is equivalent to three cycles of the clock signal applied to the 2650 microprocessor, 12 cycles or <u>periods</u> of this clock signal are required. Your Instructor 50's clock signal cycles 895,000 times each second. Therefore each *machine cycle* takes about 3.35 microseconds (millionths of a second) to execute. Since the Add-Absolute instruction requires four *machine cycles*, it takes approximately 13.4 microseconds to execute in your Instructor 50.

The "Operation" section of the page describes the instruction in terms of the notation we discussed earlier. For instance, the "Operation" line, the first line, may be read, "the contents of the specified Register R is replaced by the current contents of Register R plus the contents of the address specified by the Effective Address of the instruction." All of this is true if the contents of the With-Carry bit in the Program Status Word is a Ø. If the With-Carry bit in the Program Status Word contains a 1 then the contents of the specified Register R is replaced by its current contents added to the contents of the memory location specified by the Effective Address plus the contents of the Carry bit in the Program Status Word. The Effective Address in this case is calculated by taking the a field of the second two bytes of the instruction and adding the contents of the specified Index Register (if indexing is specified). This is the Effective Address if the Indirect bit (bit 7 of Byte 2 of the instruction) is a Ø. If the Indirect bit is a 1 the Effective Address is found in two bytes beginning at the byte specified by the address calculated above. Note on page 9-36 of your User's Guide the formulas for calculating the Effective Address of an instruction based on the states of the Indirect bit and the Index Control bits in the instruction. The symbols and abbreviations used in the instruction descriptions are listed on page 9-35 of the User's Guide.

Below the description of the instruction we see a line that says "PSW bits affected." When this particular instruction (Add-Absolute) is executed the Carry, Condition Code, Inter-Digit Carry, and Overflow bits of the Program Status Word will be set or reset depending on the results of executing the instruction. In particular, the Condition Code bits, CCØ and CC1, will be set as shown in the Condition Code setting table. The first part of Chapter 9 of the User's Guide contains a detailed description of the 2650 Processor Registers and the function of each bit in the Program Status Word.

Making It Work

Based on what we have just learned about the Add-Absolute instruction, let us go through the exercise of actually coding this instruction into binary form (machine language). You may then execute it on your Instructor 50 to prove that it works as described.

Step 1. Write the entire command in mnemonic form. For our example, let's take the most complicated possible case, Add-Absolute Indirect with Auto-Incremented Indexing. (Note: Most instructions will be much easier than this to code because the complications of Indexing, Indirect Addressing, and Auto-Incrementing are used only a small percentage of the time. They are seldom used all at once.) We must first decide which location in memory will contain the address which will contain our data to be added to Register Ø. Let us select location 1 in memory. Also let's select Register 3 as our Index Register. In mnemonic form then our instruction would look like this: ADDA,RØ *ØØØ1,R3+.

Step 2. We can now write down the binary code or machine code for our instruction. The first byte will be:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	$b_{\text{Ø}}$
1	Ø	Ø	Ø	1	1	1	1

where the two least-significant bits specify Register 3 and the six most-significant bits specify the Add Absolute instruction. In the second byte of our instruction, the most significant bit must be a 1 because we are using Indirect Addressing. The next two bits are the Index Control bits. Since we've elected to use Indexing with Auto-Incrementing we see from the table on page 9-36 that the proper code for this option is Ø1 for the Index Control bits. The next five bits of the second byte of our instruction are all Øs since the high order bits of address 1 in memory are all Øs. Thus the second byte becomes:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	$b_{\text{Ø}}$
1	Ø	1	Ø	Ø	Ø	Ø	Ø
Indirect Bit	Index Control Bits		High Order Address Bits				

Finally the third byte contains the low order address bits or

b_7	b_6	b_5	b_4	b_3	b_2	b_1	$b_{\text{Ø}}$
Ø	Ø	Ø	Ø	Ø	Ø	Ø	1

Thus our final instruction looks like this:

1 Ø Ø Ø 1 1 1 1	1 Ø 1 Ø Ø Ø Ø Ø	Ø Ø Ø Ø Ø Ø Ø 1
first byte	second byte	third byte

Step 3. The next step is to convert our binary coded instruction into hexadecimal form. As diagrammed below the first byte converts into 8F, the second byte into AØ, and the third byte into Ø1.

first byte		second byte		third byte		
1ØØØ	1111	1Ø1Ø	ØØØØ	ØØØØ	ØØØ1	(binary)
8	F	A	Ø	Ø	1	(hexadecimal)

Now let's see if our instruction works. To do this we must first enter it into some memory location. Let's arbitrarily pick location 1ØØ. To enter the three bytes of this instruction into three successive memory locations beginning at location 1ØØ, press the following sequence of buttons on your Instructor 50 keyboard: MEM, 1, Ø, Ø, ENT/NXT, 8, F, ENT/NXT, A, Ø, ENT/NXT, Ø, 1, ENT/NXT. Next, let's set up the 2650's internal registers with some initial data. For some initial data in Register Ø let's place a 2. To do this, press the button sequence REG, Ø, 2. In Register 3 let's place a 7. To do this press REG, 3, 7. Next we must initialize the Program Status Word. We do this by pressing REG, 7, F, F, ENT/NXT, Ø. Since we have placed our instruction in memory location 1ØØ, we must initialize the Program Counter to this value. This is done by pressing REG, C, 1, Ø, Ø. Since we have placed a 7 in Register 3 and have specified that Register 3 be incremented before this instruction is executed, the Effective Address will be found in the following manner.

The address specified in the instruction (Ø1) is expected to contain the first byte of an address. The next sequential location (Ø2) is expected to contain the second byte of an address. To this address (found in locations 1 and 2), the incremented value of our Index Register, R3, will be added. Let us place the data for our instruction in location FF by pressing the button sequence MEM, FF, ENT/NXT, 5. This places a 5 in location FF. When our instruction is executed we expect that this 5 will be added to the 2 already in Register Ø to produce 7. So that our Effective Address will be location FF, we must enter the following information in locations 1 and 2: MEM, 1, ENT/NXT, Ø, ENT/NXT, F, 7. Now when the ØØF7 contained in locations 1 and 2 is added to 8, the incremented value of our Index Register (R3), the result will be FF, the location of our data in memory. Next press STEP. The instruction has now executed and Register Ø should contain a 7. To find out if it does, press REG, Ø. Also note that Register 3 should contain an 8 and the Program Counter should contain 1Ø3.

5. PROGRAMMING TECHNIQUES

ORGANIZING YOUR PROGRAM

Now that you know how instructions work, how to interpret their descriptions and what they can do, you are ready to start writing programs for your Instructor 50. When you write a program, what you are trying to do is define a sequence of numbers which will be entered into memory locations in your Instructor 50, and which will perform some desired task when they are executed as a program. This chapter discusses some of the techniques which have been evolved for progressing from a vague idea of the task to be accomplished to a completed listing of *machine code* which can be entered into your Instructor 50.

The first thing you should do is diagram your program in flow chart form. The *flow chart* is your first cut at reducing the solution of your problem to an algorithm, that is a specific sequence of manipulations which, taken together, will solve the problem. A simple example of a *flow chart* is shown in Figure 5.1.

This *flow chart* describes a program which adds together two bytes stored in memory. If their sum is greater than 5, the program is to display the message "HI" on the 8-digit display. If the sum is 5 or less, the program is to display the message "LO" on the Instructor 50 display.

The program starts in the upper left hand corner labeled Start. The first operation performed is to get a byte of data from memory and place it in Register Ø in the Central Processing Unit. The next task is to add to the number stored in Register Ø some other data byte from memory. Next, we must make a decision. Is the result of this addition greater than 5 or is it not? If the result is not greater than 5, the message

"LO" is displayed. If the result is greater than 5, the message "HI" is displayed. The little circle containing the "A" is a connector. In the upper right hand corner you see another circle with an "A" meaning that the *flow chart* continues from this point.

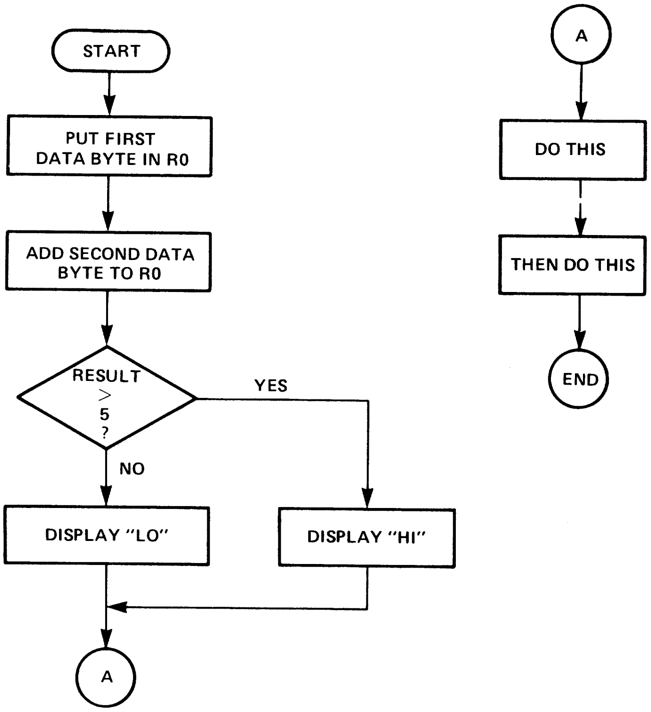
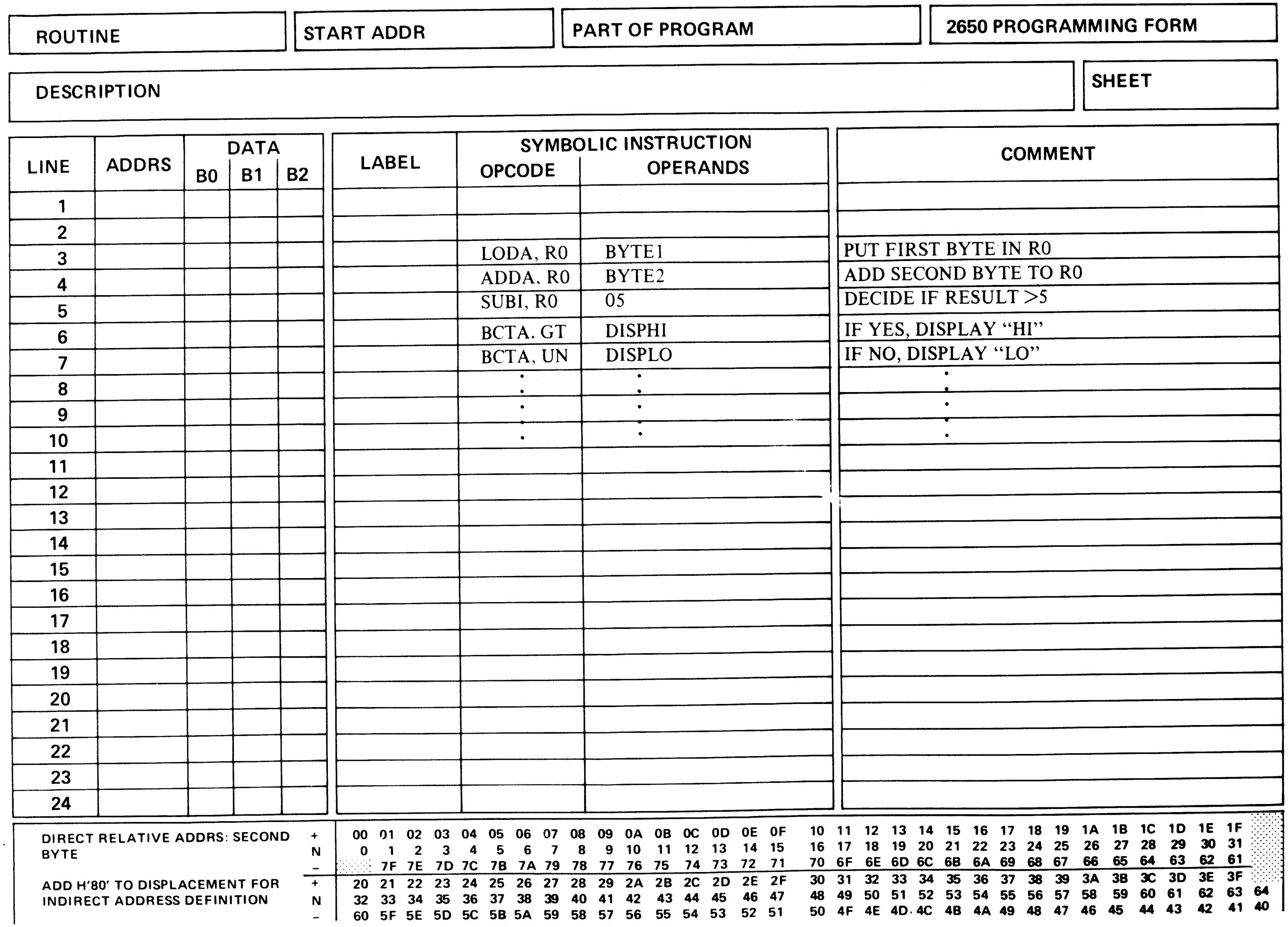


Figure 5.1 A SIMPLE FLOW CHART

Additional sequential blocks may be defined until the program comes to an end. If the flow chart is written in sufficient detail, it is then an easy matter to convert the contents of each block of the flow chart to a few corresponding machine instructions. The sequence of instructions is normally written first in *mnemonic* form on a coding form similar to that shown in Figure 5.2.

For our example program, which is flow charted in Figure 5.1, we might begin filling out the symbolic instruction columns of this *coding form* as shown in Figure 5.2.

Notice that we have begun filling out mnemonics for each instruction required by our flow chart in the column labeled "OP CODE." Under the



ROUTINE

START ADDR

PART OF PROGRAM

2650 PROGRAMMING FORM

DESCRIPTION

SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3						LODA, R0	BYTE1	PUT FIRST BYTE IN R0
4						ADDA. R0	BYTE2	ADD SECOND BYTE TO R0
5						SUBI, R0	05	DECIDE IF RESULT >5
6						BCTA. GT	DISPHI	IF YES, DISPLAY "HI"
7						BCTA, UN	DISPLO	IF NO, DISPLAY "LO"
8						.	.	.
9						.	.	.
10						.	.	.
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	–		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	–	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.2 MNEMONIC LISTING ON CODING FORM

column labeled "OPERANDS," we have listed symbols for any additional information required by the instruction. In the "COMMENT" column, we placed a short description of what the instructions we've selected are expected to do. Notice that our first instruction, Load Absolute into Register Ø, requires an address which specifies where the byte of data to be loaded will be found. Since we haven't yet picked a specific address, we substituted for the address the label "BYTE1."

The second instruction, Add Absolute to Register Ø, also requires an address. Again, not knowing what the address would be we have labeled it "BYTE2." The next instruction, Subtract Immediate from Register Ø, will have the effect of subtracting the constant 5 from the sum of the contents of locations BYTE1 and BYTE2. As a result of this operation the Condition Code bits (CCØ and CC1) in the Program Status Word will be set to Ø1 for positive, ØØ for zero, or 1Ø for negative. The next instruction, Branch on Condition True Absolute if Greater Than, will cause the normal sequence of instructions to be changed. The next instruction will not be taken from the next location in memory but rather from a location which we've labeled "DISPHI." Finally, our last instruction Branch on Condition True Absolute Unconditional, will take the next instruction from the location in memory which we've labeled "DISPLO."

The *labels* that we've used above, "BYTE1," "BYTE2," "DISPHI," and "DISPLO," must be translated into specific addresses before we can develop the machine code for our program. Let us arbitrarily select locations 1ØØ and 1Ø1 as storage places for our two data bytes, BYTE1 and BYTE2. We add this to our coding form as shown in Figure 5.3.

In the "LABEL" column, we have added the labels BYTE1 and BYTE2. Under the "OP CODE" column for these two bytes we've written "RES," which is simply a note to tell us to reserve a location. (This is not strictly a microprocessor instruction.) Under the "OPERANDS" column we've written a 1, indicating that we wish to reserve only one location in memory for the label "BYTE1" rather than a number of sequential locations. Finally,

ROUTINE	START ADDR	PART OF PROGRAM	2650 PROGRAMMING FORM
DESCRIPTION			SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3						LODA, R0	BYTE1	PUT FIRST BYTE IN R0
4						ADDA, R0	BYTE2	ADD SECOND BYTE TO R0
5						SUBI, R0	05	DECIDE IF RESULT >5
6						BCTA. GT	DISPHI	IF YES, DISPLAY "HI"
7						BCTA, UN	DISPLO	IF NO, DISPLAY "LO"
8						.	.	.
9						.	.	.
10						.	.	.
11								
12								
13								
14								
15								
16	100				BYTE1	RES	1	RESERVE LOCATION 100
17	101				BYTE2	RES	1	RESERVE LOCATION 101
18						.		
19	102				DISPHI	.		
20	.				.	.		
21	.				.	.		
22	150				DISPLO	.		
23					.	.		
24					.			

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.3 DEFINING LABELS

in the "ADDRESS" column ("ADDRS") toward the left hand side of the coding sheet, we've written 1ØØ and 1Ø1, the actual addresses in which we wish to store Byte 1 and Byte 2.

Let us also arbitrarily pick location 1Ø2 as the address for the program segment labeled "DISPHI," and 15Ø as the address for the program segment labeled "DISPLO." We are now ready to write down the actual machine code for the instructions which we've written in mnemonic form in our example. There is just one decision left to be made. We must determine where in memory we wish our program to begin.

Since the Instructor 50 takes its first instruction from location Ø after the RESET button is pushed, let us begin our program at memory location Ø. Now in the "ADDRS" column opposite our first instruction, Load Absolute RØ, we may write the address of this instruction, ØØØØ, as shown in Figure 5.4.

Next, we look up the format for the Load Absolute instruction in the Instructor 50 User's Guide. We see that this is a 3-byte instruction, and that if the label "BYTE1" refers to location 1ØØ and we are not using Indirect Addressing or Indexing, the hexadecimal code for this instruction must then be ØC Ø1 ØØ. These three hexadecimal codes are written in the "DATA" columns labeled "BØ," "B1" and "B2" on the coding form. Since they are to be placed in three sequential memory locations beginning with Address Ø, they will occupy locations Ø, 1 and 2 in memory. Our next instruction therefore will begin at memory location 3, and we may write this value in the "ADDRS" column. Looking up the format for the Add Absolute instruction in the Instructor 50 User's Guide we see that this is again a 3 byte instruction, and that the hexadecimal codes for it are 8C, Ø1, Ø1. The first byte specifies that we are Adding-Absolute to Register Ø and the second two bytes specify the location in memory of the data we have labeled "BYTE2." Since this is a three byte instruction, it will occupy locations 3, 4 and 5 in memory, and the next instruction will begin at location ØØØ6, which we write in the "ADDRS" column.

ROUTINE	START ADDR	PART OF PROGRAM	2650 PROGRAMMING FORM

DESCRIPTION	SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3	0000	0C	01	00		LODA, R0	BYTE1	PUT FIRST BYTE IN R0
4	0003	8C	01	01		ADDA, R0	BYTE2	ADD SECOND BYTE TO R0
5	0006	A4	05			SUBI, R0	05	DECIDE IF RESULT >5
6	0008	1D	01	02		BCTA. GT	DISPHI	IF YES, DISPLAY "HI"
7	000B	1F	01	50		BCTA, UN	DISPLO	IF NO, DISPLAY "LO"
8	000E					·	·	·
9						·	·	·
10						·	·	·
11								
12								
13								
14								
15								
16	100				BYTE1	RES	1	RESERVE LOCATION 100
17	101				BYTE2	RES	1	RESERVE LOCATION 101
18						·		
19	102				DISPHI	·		
20	·				·	·		
21	·				·	·		
22	150				DISPLO	·		
23					·	·		
24					·			

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	–		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	–	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.4 CODING THE INSTRUCTIONS

The description of the Subtract-Immediate instruction in the User's Guide indicates that the hexadecimal code for this instruction will be A4 Ø5. These values are written in the "BØ" and "B1" columns on the coding form and will occupy locations 6 and 7 in memory.

Our next instruction will then appear at location 8 in memory which we may write down in the "ADDRS" column followed by the hexadecimal code for our Branch on Condition True Absolute if Greater Than instruction.

The format shown in the User's Guide for this instruction indicates it is a three byte instruction whose hexadecimal codes are 1D, Ø1, Ø2. These three bytes will occupy locations 8, 9 and A in memory, and our next address on the coding form will be address B. This instruction will be coded for unconditional branching to the address of "DISPLO" Therefore the three hexadecimal bytes of code for this instruction are 1F, Ø1, 5Ø. These three bytes will occupy locations B, C and D in memory. The next instruction would begin at location E in memory. We have now completed the coding for this segment of our program.

This code may now be entered directly into the Instructor 50's memory and executed. You are encouraged to try this out by entering the code indicated through the keyboards on your Instructor 50. Since we have not completely coded our program however, that is, we have not decided which instructions to use at the locations labeled "DISP HI" and "DISP LO," simply pressing the RESET button will produce unpredictable results. Therefore, simply set the Program Counter to location Ø and step through the program one instruction at a time until the Program Counter contains either 1Ø2 or 15Ø. Enter some trial bytes of data in locations 1ØØ and 1Ø1 and confirm that the program branches to location 1Ø2 or 15Ø depending on whether the sum of these two numbers is greater than 5 or not. To avoid difficulty during this exercise make sure that the INTERRUPT SELECTOR switch on the bottom of your Instructor 50 is placed in the KEYBOARD position.

To sum up the process used in writing a program for your Instructor 50, first, define the algorithm in flow chart form. The flow chart should be sufficiently detailed so that each block may be converted into just a few 2650 instructions. Naturally, as you become more and more familiar with the instructions which are available to you, this will become easier and easier to do.

Next, write down the list of instructions in mnemonic form on a coding form. You may use *labels* to substitute for addresses when addresses are not to be known until later. It is also useful to fill out the "COMMENT" column at this point to remind yourself (and others who may read your program) of what actions you intended. Finally, the "ADDRS" and "DATA" columns may be filled out using the information you have already generated and the instruction descriptions in the User's Guide. As you become more familiar with the descriptions of the instructions, you will find that the short coding reference table provided on the operator panel of your Instructor 50 is sufficient in most cases and you will not need to refer to the detailed descriptions contained in the User's Guide.

LOOPING AND BRANCHING

In our example flow chart of Figure 5.1 we saw a diamond shaped decision block which caused the program to branch to one location in memory or another depending on whether the result of the calculation was greater than 5 or not. Whenever such a *decision block* occurs in a flow chart you may be certain that the actual program involves some sort of *conditional branch* instruction. When a branch instruction causes a previously executed instruction to be executed again, a loop is said to be formed. The reason for this may be seen by examining the flow chart diagrammed in Figure 5.5, a conditional loop.

On this flow chart the instructions indicated by Block 1 are executed followed by those in Blocks 2, 3 and 4. Block 4 is a *decision block* which contains a conditional branch instruction. Depending on whether some condition is satisfied, Block 5 or Block 2 may be executed next. Blocks 2, 3 and 4 then, may be said to be contained in a *loop*. The program will continue going around and around this *loop* until the condition specified in the *decision block* is satisfied. This then is said to be a *conditional loop*.

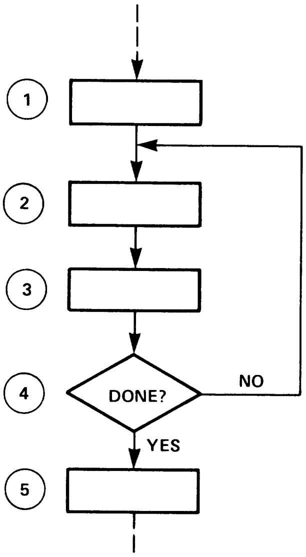


Figure 5.5 A CONDITIONAL LOOP

Looping is a very useful programming technique. The instructions within the loop may be carried out thousands of times, whereas they need only be written down once and only occupy a small number of locations in memory. For instance, suppose our computer had just accomplished some task which we had programmed it to perform and we wanted it to signal when it was finished by flashing a light on and off. This segment of our program might be diagrammed by the flow chart in Figure 5.6.

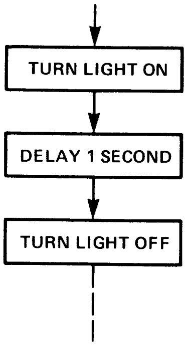


Figure 5.6 FLOWCHART TO FLASH LIGHT

Knowing that each instruction takes a little while to execute, the "DELAY" Block could be implemented by executing a number of instructions which serve no function other than to consume time. Looking around for such an instruction we find that "NOP" (No Operation) satisfies this criteria. Since a "NOP" does nothing but occupy two machine cycles of execution time, we conclude that a sequence of 149,167 "NOP" instructions would accomplish our one second delay. That is if we had enough patience to key all these "NOP" instructions into our Instructor 50 and it had enough memory to hold them all. A much better choice would be the sequence of instructions shown in Figure 5.7.

In this sequence of instructions, Register 1 is first loaded with the hexadecimal value "E9" and Register Ø with the hexadecimal value "FF." When the Subtract Immediate instruction is executed, one is subtracted from the value in Register Ø, leaving FE. We then come to the conditional branch instruction, Branch on Condition False Relative, the condition being if Register Ø is equal to a Ø. The first time through it is not, so we branch back to the previous instruction labeled "INLOOP" and subtract 1 from the contents of Register Ø again, this time obtaining FD as a result. This small loop is repeated until the contents of Register Ø have been decremented to a Ø. It will take $255_{1Ø}$ times through this loop to make the contents of Register Ø equal to Ø. Therefore these two instructions beginning with the label "INLOOP" will be executed 255 times each. Since the Subtract Immediate instruction takes 6.7 microseconds to execute, and the Branch on Condition False Relative instruction takes 1Ø.Ø6 microseconds to execute, this small loop will occupy 16.76 x 255 or 4,273.8 microseconds of execution time as noted in the "COMMENTS" column. After we have been delayed for 4,273.8 microseconds we will subtract a 1 from the contents of Register 1, which began as E9. If the result is not equal to Ø, the next instruction will cause us to branch back to the instruction labeled "OUTLOP" which again loads Register Ø with an FF. We then begin another 4,273.8 microsecond delay. After this time, 1 will again be subtracted from Register 1 bringing the contents of Register 1 to E7 and causing a branch back for another 4,273.8 microsecond delay. When all this happens $233_{1Ø}$ (equals

ROUTINE | START ADDR | PART OF PROGRAM | 2650 PROGRAMMING FORM

DESCRIPTION | SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3								
4		05	E9			LODI, R1	$E9_{16}$	(233_{10})
5		04	FF		OUTLOP	LODI, R0	FF_{16}	6.7μS (255_{10})
6		A4	01		INLOOP	SUBI, R0	01	6.7μS } 16.76μS x 255=4273.8μS
7		98	7C			BCFR, EQ	INLOOP	10.06μS
8		A5	01			SUBI, R1	01	6.7μS
9		98	76			BCFR, EQ	OUTLOP	10.06μS
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.7 A ONE SECOND TIME DELAY

$E9_{16}$) times, Register 1 will contain a Ø and the branch to "OUTLOP" will not occur. We will have by then delayed approximately one second (233 x 4273.8 = 995,795.4 microseconds) and the next instruction in sequence will be executed.

In this example, Registers Ø and 1 were used as loop counters. That is, they were used to keep track of the number of times the program executed their respective *loops*. The inner loop (program segment beginning with the label "INLOOP") was executed $255_{1Ø}$ or FF_{16} times because FF was the number initially loaded into Register Ø. The outer loop (program segment beginning with "OUTLOP") was executed $E9_{16}$ or $233_{1Ø}$ times since E9 was the number originally loaded into Register 1. The actual machine code for these instructions is shown in the "DATA" columns of the coding form. Notice that only twelve bytes of memory are required, versus the nearly 150,000 bytes which would have been required had we elected to use a series of NOP instructions to form our one second delay.

In this example, it is particularly instructive to note the second byte of each of the two Branch on Condition False Relative (BCFR) instructions. These values were obtained directly from the Relative Address table provided at the bottom of the coding form. The BCFR instruction which branches to "INLOOP" must have a second byte which represents -4, as this is the value by which the program counter must be altered to bring it to the address "INLOOP." We see from the table the code for "-4" is "7C."

The second BCFR instruction contains a value 76 in its second byte. This is the hex code for "-10", the value by which the Program Counter must be altered after executing this instruction to effect a branch to the location labeled "OUTLOP."

SUBROUTINES

In the example above, we developed a short program segment which will delay execution by one second. If we had a program which was to flash several lights on and off for one second each, this program segment could be inserted in our program each time we wish to flash a light on and off. Rather than having to rewrite this program segment each time we wish to use it, it would be convenient if we could write it only once and branch to it from any place in our program, and then branch back to the main program when this program segment is complete. Such a program segment that is used by various parts of the main program is called a subroutine.

A *subroutine* may be any program segment that does some useful function which may be required often, such as a time delay or calculating the square root of a number, or reading an entry from the keyboard or displaying a message on the 8-digit display of your Instructor 50.

Subroutines to perform these last two functions are part of the monitor program contained in your Instructor 50's Read Only Memory. These monitor subroutines may also be used by programs that you write. What each of the *monitor program's subroutines* does and how each may be used by your programs is explained in your User's Guide.

The use of *subroutines* is diagrammed in Figure 5.8.

Instructions of the main program are executed until a Branch to Subroutine instruction is encountered. The Branch to Subroutine instruction is similar to other branch instructions in that it will specify the address of the first instruction of the subroutine. A branch will then occur to the subroutine as indicated by the arrow labeled 1 in Figure 5.8. The instructions of the subroutine are executed until a Return from Subroutine instruction is encountered, signalling the end of the subroutine.

The next instruction must then be taken from the main program, and a branch must occur along path 1' as indicated in Figure 5.8. The main program instructions are again executed until another Branch to Subroutine instruction is encountered. The branch then occurs along path 2 and returns along path 2'.

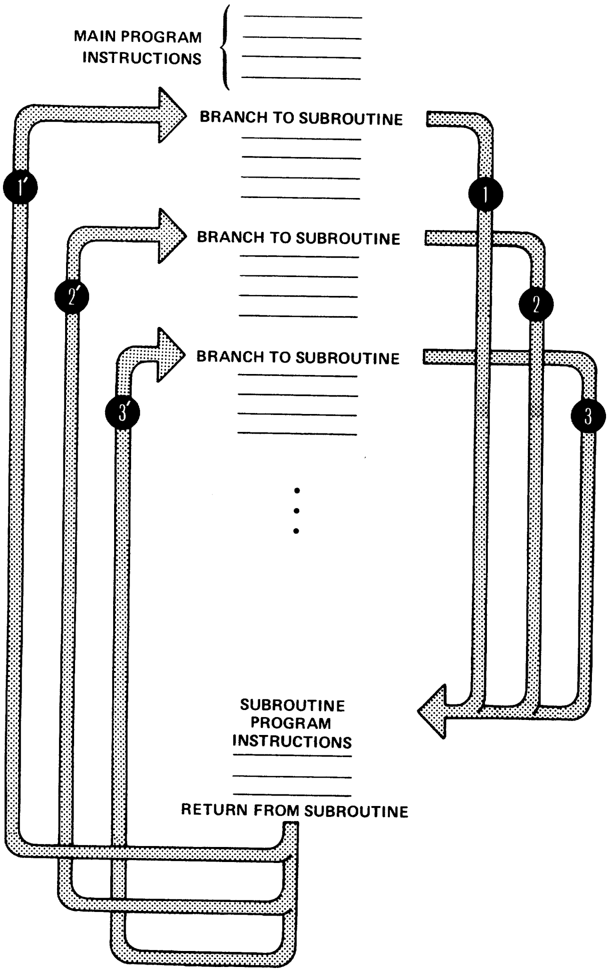


Figure 5.8 PROGRAM WITH SUBROUTINE

Subsequently, a third Branch to Subroutine instruction may cause a branch along path 3 and return along path 3'.

There is an important difference between the Branch to Subroutine instruction and Return from Subroutine instruction and other branch instructions which we have discussed earlier. This comes from the fact that when the subroutine is completed it is impossible for the Return from Subroutine instruction to specify the location of the next instruction in the main program. This is because this location may change depending on where the Branch to Subroutine was encountered in the main program.

To circumvent this complication the Branch to Subroutine instruction operates in a special way. Instead of simply altering the Program Counter as is done in a normal branch instruction, the current contents of the Program Counter are first stored in a special register reserved for this purpose in the Central Processor Unit, called a Return Address Register. The contents of the Program Counter are then altered in the normal way to indicate the location of the first instruction in the subroutine. When the Return from Subroutine instruction is encountered the Central Processor Unit retrieves the location which has been stored in the Return Address Register and places it back in the Program Counter. This happens automatically as a result of executing the Return from Subroutine instruction. Thus, the next instruction to be executed will be the next instruction in the main program following the Branch to Subroutine instruction.

The process of branching to a subroutine is often called Calling a subroutine, and the Branch to Subroutine instruction is often called a Subroutine Call instruction. If a subroutine is to be used it is important to access it via the Branch to Subroutine instruction rather than some other jump instruction. Each subroutine must end with a Return from Subroutine instruction, which replaces the contents of the Program Counter with the contents which were stored during the corresponding *subroutine call*.

As an example of the use of a subroutine, let's convert our time delay program segment into a subroutine, and locate the subroutine at memory location 1ØØ. We've labeled the subroutine "TIMDLY." In the "ADDRS" column we've inserted the addresses in which our instructions codes are to be placed beginning with address 1ØØ. See Figure 5.9. Next, let's write a simple program which will make use of this subroutine to flash the 8 LED's on your Instructor 50 operator's panel on for one second and then off for one second and then back on again.

The flow chart for this program is shown in Figure 5.10.

ROUTINE	START ADDR	PART OF PROGRAM	2650 PROGRAMMING FORM
DESCRIPTION			SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3								
4	0100	05	E9		TIMDLY	LODI, R1	E9$_{16}$	(233$_{10}$)
5	0102	04	FF		OUTLOP	LODI, R0	FF$_{16}$	6.7μS (255$_{10}$)
6	0104	A4	01		INLOOP	SURI, R0	01	6.7μS } 16.76μS x 255=4273.8μS
7	0106	98	7C			BCFR, EQ	INLOOP	10.06μS
8	0108	A5	01			SURI, R1	01	6.7μS
9	010A	98	76			BCFR, EQ	OUTLOP	10.06μS
10	010C	17				RETC, UN		RETURN FROM SUBROUTINE
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.9 TIME DELAY SUBROUTINE

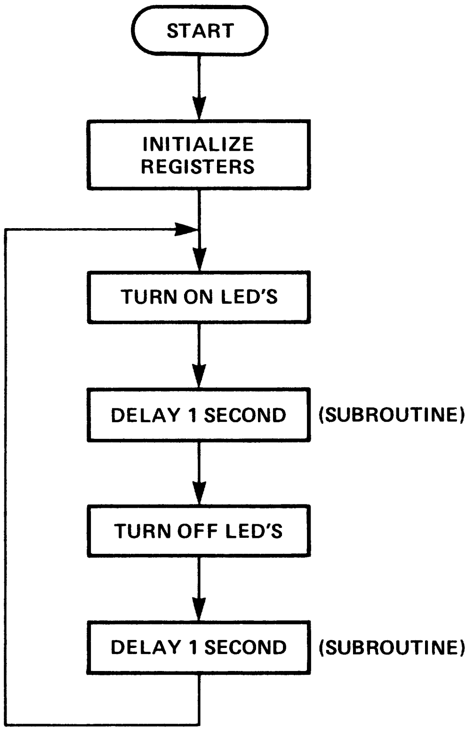
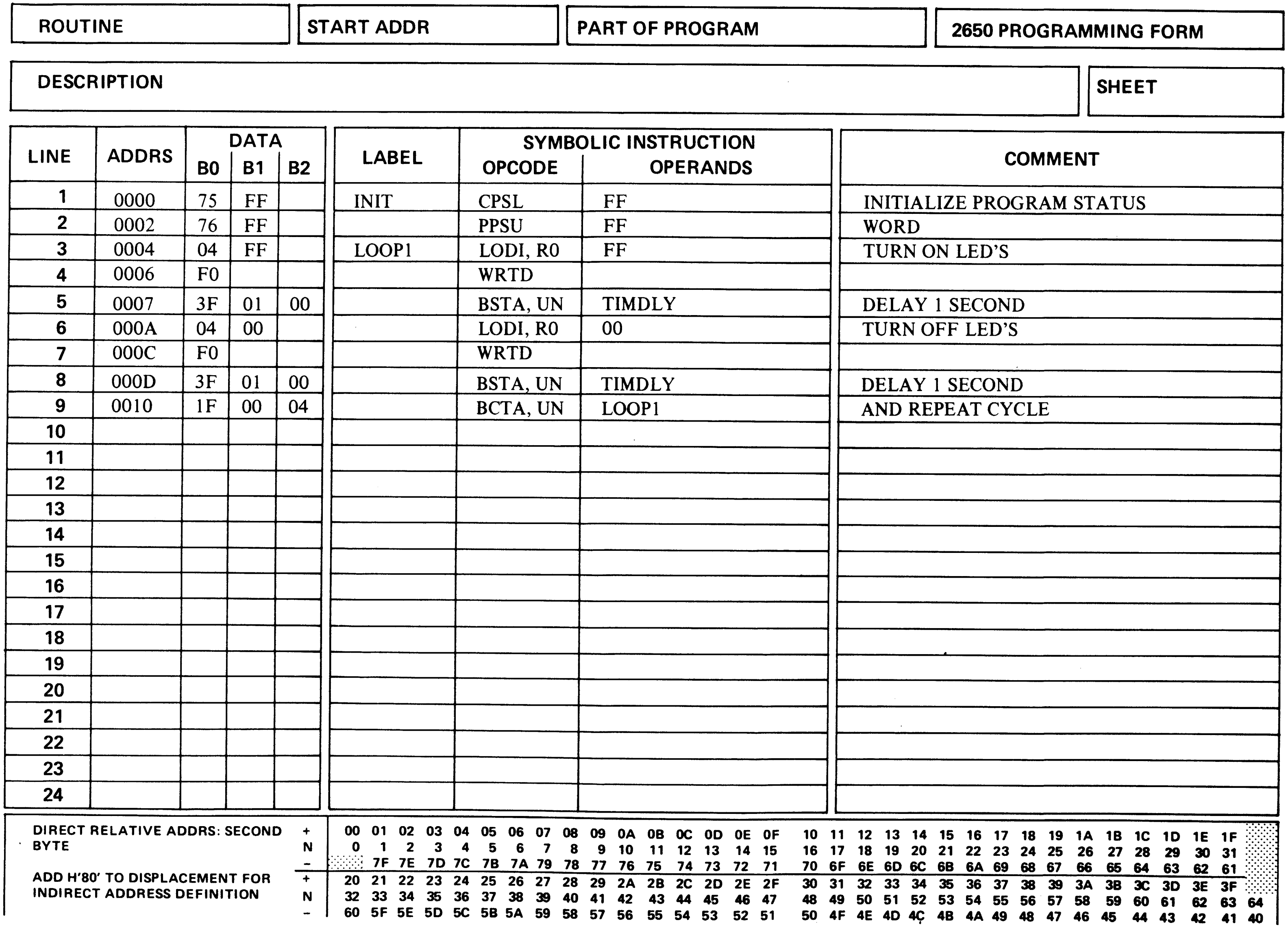


Figure 5.10 FLOW CHART TO FLASH LED'S

This flow chart may be converted into a program as shown in the coding form of Figure 5.11. Notice that the first two instructions in this program initialize the Program Status Word to a known value.

This is generally good practice as the settings of bits in the Program Status Word can affect operations of other instructions in the program. It may be desirable, depending on the program, to initialize other CPU registers to known values as well. Our initialization procedure simply clears all the bits in the lower byte of the Program Status Word and sets all the bits in the upper byte of the Program Status Word.

The next instruction, which we have labeled "LOOP1" loads Register Ø with all ones. This value is then output to the Data I/O port with the WRTD instruction. We then branch to the time delay subroutine ("TIMDLY") at location 1ØØ. After a one second delay we load Register Ø with an all Ø bit pattern and write this pattern to the Data port.



ROUTINE	START ADDR	PART OF PROGRAM	2650 PROGRAMMING FORM
DESCRIPTION			SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1	0000	75	FF		INIT	CPSL	FF	INITIALIZE PROGRAM STATUS
2	0002	76	FF			PPSU	FF	WORD
3	0004	04	FF		LOOP1	LODI, R0	FF	TURN ON LED'S
4	0006	F0				WRTD		
5	0007	3F	01	00		BSTA, UN	TIMDLY	DELAY 1 SECOND
6	000A	04	00			LODI, R0	00	TURN OFF LED'S
7	000C	F0				WRTD		
8	000D	3F	01	00		BSTA, UN	TIMDLY	DELAY 1 SECOND
9	0010	1F	00	04		BCTA, UN	LOOP1	AND REPEAT CYCLE
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.11 PROGRAM TO FLASH LED'S

Again we delay for one second and then branch to the instruction labeled "LOOP1" which begins the sequence again. You may try this out on your Instructor 50 by loading the code shown in Figures 5.9 and 5.11 into memory and then pressing the RST button. Make sure that the three-position switch below the "Parallel Input/Output" port toward the left-hand of your Instructor 50 operator's panel is in the lower position, labeled "Non-Extended Data Port."

Nested Subroutines

Sometimes, it is convenient for a subroutine to call another subroutine, as illustrated in the diagram of Figure 5.12.

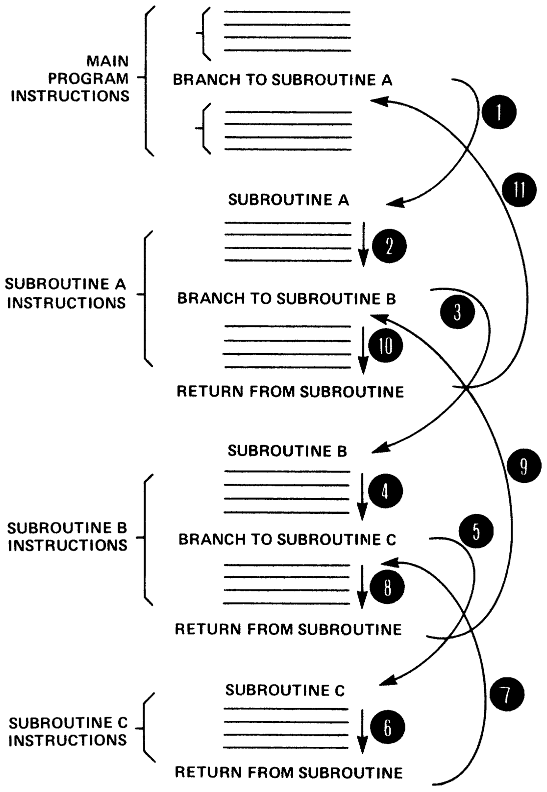


Figure 5.12 NESTED SUBROUTINES

As the main program executes, a Branch to Subroutine is encountered which branches the program to Subroutine A along branch path 1. Subroutine A is then executed along path 2 until another Branch to Subroutine instruction transfers program control along path 3, at which time Subroutine B begins execution along path 4.

Soon another Branch to Subroutine instruction is encountered which transfers control along path 5 to Subroutine C. When Subroutine C is completed a branch along path 7 transfers control back to the remainder of Subroutine B which upon its completion transfers control along path 9 to the remainder of Subroutine A. At the completion of Subroutine A control is finally transferred along path 11 to the remainder of the main program.

In this example Subroutines B and C are said to be nested subroutines. That is, Subroutine B is *nested* within Subroutine A. And Subroutine C is *nested* within Subroutine B.

When the Branch to Subroutine B occurs in Figure 5.12, the contents of the Program Counter must be stored in a Return Address Register. If there were only one Return Address Register in your 2650 microprocessor this would mean that the return address for Subroutine A would have to be lost. This is prevented in the 2650 microprocessor by providing eight Return Address Registers which together are called the Return Address Stack. Which register of the *Return Address Stack* is currently in use is kept track of by three bits in the Program Status Word called SPØ, SP1, and SP2. These three bits taken together may be considered to be a 3-bit binary number which indicates the number of the Return Address Register currently in use. They may be said to point to a particular Return Address Register, and their name, "SP," is an abbreviation for Stack Pointer.

In operation, when a Branch to Subroutine instruction is encountered by the 2650 microprocessor, the processor first increments the value contained in the three bits of in the *Stack Pointer*. The Return Address for the subroutine is then stored in the Return Address Register indicated by the contents of the *Stack Pointer*. When a Return from Subroutine instruction is encountered the Return Address is retrieved from the

Return Instruction Register indicated by the *Stack Pointer* and the *Stack Pointer* is automatically decremented by 1. This allows nesting of subroutines up to 8 levels.

INTERRUPTS

An interrupt is similar to a subroutine call except that it is not initiated by a Branch to Subroutine instruction. It is instead, initiated by a logic signal applied to one of the pins of the 2650 microprocessor. At any time and during the execution of any instruction this pin which is called "$\overline{\text{INTREQ}}$" for "Interrupt Request," may make a transition from a logic 1 to a logic Ø. The effect of this is similar to the effect of inserting a Branch to Subroutine instruction following the instruction during which the transition from a logic 1 to a logic Ø occurs.

In your Instructor 50, the subroutine is expected to start at location 7 in memory. Since this subroutine occurs in response to an *interrupt request*, it is called an interrupt service routine. On your Instructor 50 the *interrupt* signal may be activated by pushing the button labeled INT on the Instructor 50 function keyboard. This is true so long as the INTERRUPT SELECTOR switch on the bottom of your Instructor 50 is in the KEYBOARD position. If the INTERRUPT SELECTOR switch is in the AC LINE position an *interrupt* signal will be generated for each cycle of the power line applied to your Instructor 50. In other words, in the United States an *interrupt request* will be generated 60 times each second, and in Europe 50 times each second, if the *Interrupt Selector* switch is in the AC LINE position.

Response to an Interrupt Request will only occur if the Interrupt Inhibit bit in the Program Status Word is a Ø. If this bit contains a 1 and an Interrupt Request occurs, it will not be serviced until the Interrupt Inhibit bit is changed to a Ø, as by execution of a CPSU instruction.

To illustrate the use of interrupts in your Instructor 50 let's write a simple program to change the state of the LED's of the parallel I/O port whenever the Interrupt button is depressed. Our flow chart might look like the one shown in Figure 5.13.

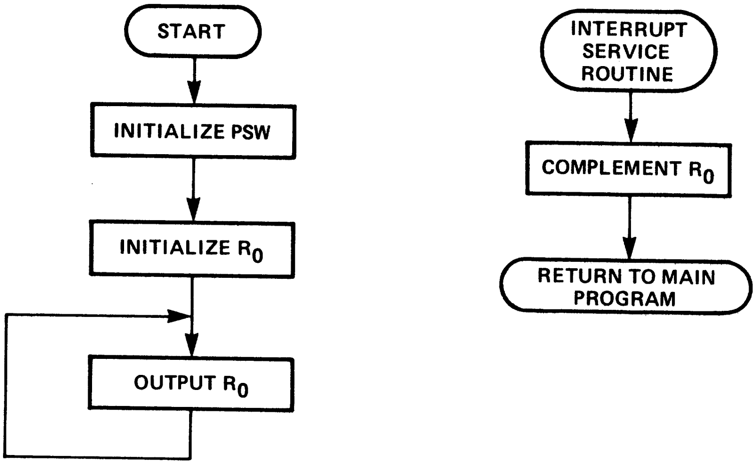


Figure 5.13 PROGRAM USING INTERRUPT

First, we will initialize the Program Status Word to make sure that the Interrupt Inhibit bit is set to a Ø. Then we will place the value FF in Register Ø, and output the contents of Register Ø to the "Non-Extended Data" port, thus turning on all the LED's. We will continue looping on the last block of this flow chart waiting for an interrupt to occur. When the interrupt occurs, the *Interrupt Service Routine* will complement the value in Register Ø and return to the main program. Thus, when the Interrupt button is pressed the first time all of the LED's will go off. When it is pressed the second time they will go on. Figure 5.14 shows this program converted into actual machine code on a coding form.

The instructions in locations Ø and 2 initialize the Program Status Word. The instruction at location 4 branches beyond the Interrupt Service Routine which must begin at location 7. The main program begins at location A which we have labeled "Resume." It initializes Register Ø to contain an FF and then outputs the contents of Register Ø to the "Non-Extended Data" port. Finally, the program branches back to the output instruction, whose location we have labeled "OUTPUT." The WRTD and BCTA instructions will be executed in a loop until an interrupt occurs. At that time the Exclusive-Or Immediate instruction, (EORI) will complement all of the bits in Register Ø. The Return and Enable

ROUTINE | START ADDR | PART OF PROGRAM | 2650 PROGRAMMING FORM

DESCRIPTION | SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1	0000	74	FF			CPSU	FF	INITIALIZE PSW
2	0002	75	FF			CPSL	FF	
3	0004	1F	00	0A		BCTA, UN	RESUME	BRANCH AROUND SERVICE ROUTINE
4	0007	24	FF		INTSRV	EORI , R0	FF	INTERRUPT SERVICE
5	0009	37				RETE, UN		ROUTINE
6	000A	04	FF		RESUME	LODI, R0	FF	INITIALIZE R0 TO FF
7	000C	F0			OUTPUT	WRTD		OUTPUT CONTENTS OF R0
8	000D	1F	00	0C		BCTA, UN	OUTPUT	AND WAIT FOR INTERRUPT
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.14 CODE FOR PROGRAM USING INTERRUPT

instruction is the same instruction used to return from subroutines. It will allow the main program to continue with the Inhibit Interrupt bit in the Program Status Word set to a Ø.

You are encouraged to load this code into your Instructor 50's memory and try it out. Make sure that the "Direct-Indirect Interrupt" switch is placed in the "Direct" position for this program to work. Also, be sure that the INTERRUPT SELECTOR switch on the bottom of your Instructor 50 is placed in the KEYBOARD position, and that the 3-position switch below the eight LED's on the left hand side of your Instructor 50 operator's panel is placed in the downward position, labeled "Non-Extended Data" port. This allows the LED's to be activated by the WRTD instruction.

Each time you press the Interrupt button, the eight LED's should assume the condition opposite to what they have previously been. While the program is running place the switch on the botton of your Instructor 50 into the AC LINE position. You should see the LEDs flickering rapidly, turning on and off as each cycle of the power line generates an Interrupt Request signal. The DESKCLOCK program on your Instructor 50 Introductory Cassette makes use of the AC LINE Interrupt to keep track of time. Each 60 interrupts (50 in Europe) mark the passage of one second of time.

If we wish to modify this program to use Indirect Interrupting, this may be done as shown in Figure 5.15. This is similar to the coding for Figure 5.14, except that in location 7 we have placed not the first instruction of our Interrupt Service Routine but the address of the first instruction of our Interrupt Service Routine. We have selected Ø1ØØ as this address. Then at location 1ØØ in memory we have placed the code for the actual Interrupt Service Routine. For this version of the program to execute properly on your Instructor 50 the INTERRUPT switch just to the lower left hand side of the function keyboard must be in the INDIRECT position. Try this modification and verify that it works as well.

ROUTINE	START ADDR	PART OF PROGRAM	2650 PROGRAMMING FORM

DESCRIPTION	SHEET

LINE	ADDRS	DATA			LABEL	SYMBOLIC INSTRUCTION		COMMENT
		B0	B1	B2		OPCODE	OPERANDS	
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11	0000	74	FF			CPS U	FF	INITIALIZE PSW
12	0002	75	FF			CPS L	FF	
13	0004	1F	00	0A		BCTA, UN	RESUME	BRANCH AROUND SERVICE ROUTINE
14	0007	01	00		INTSRV	EORI , R0	FF	ADDRESS OF INTERRUPT
15								ROUTINE
16	000A	04	FF		RESUME	LODI, R0	FF	INITIALIZE R0 TO FF
17	000C	F0			OUTPUT	WRTD		OUTPUT CONTENTS OF R0
18	000D	1F	00	0C		BCTA, UN	OUTPUT	AND WAIT FOR INTERRUPT
19								
20	0100	24	FF		INTSRV	EORI, R0	FF	INTERRUPT SERVICE
21	0102	37				RETE, UN		ROUTINE
22								
23								
24								

DIRECT RELATIVE ADDRS: SECOND BYTE	+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
	N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
	-		7F	7E	7D	7C	7B	7A	79	78	77	76	75	74	73	72	71	70	6F	6E	6D	6C	6B	6A	69	68	67	66	65	64	63	62	61	
ADD H'80' TO DISPLACEMENT FOR INDIRECT ADDRESS DEFINITION	+	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F	
	N	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	-	60	5F	5E	5D	5C	5B	5A	59	58	57	56	55	54	53	52	51	50	4F	4E	4D	4C	4B	4A	49	48	47	46	45	44	43	42	41	40

Figure 5.15 CODE FOR PROGRAM USING INDIRECT INTERRUPT

CHAPTER VI

Note: Since its inception, Chapter Six of this introductory manual has been expanded considerably in scope. As a convenience to our customers, it is now being printed as a separate manual entitled, "The Instructor 50™ Computer Applications Manual."

GLOSSARY

ACCUMULATOR

A register of the Arithmetic Logic Unit (ALU) of a central processor used as intermediate storage during the formation of algebraic sums, or for other intermediate logical and arithmetic operations.
Register and related circuitry that holds one operand for arithmetic and logical operations.

ADDRESS

A unique label, name, or number that identifies a memory location or a device register for access by a computer.
A number used by the CPU to specify a location in memory.

ADDRESSING MODE

The method of specifying the memory location of an operand. Common addressing modes are: REGISTER, IMMEDIATE, ABSOLUTE, RELATIVE, INDIRECT, AND INDEXED. These modes are important factors in program efficiency.

ALGORITHM

A prescribed set of well-defined rules or processes for the solution of a problem. Algorithms are implemented on a computer by a programmed sequence of instructions.

ASSEMBLER PROGRAM

A computer program which converts a symbolic assembly language program into an executable object (binary-coded) program.

ASSEMBLY LANGUAGE

A human oriented symbolic-mnemonic source language which is used by the programmer to encode programs and associated data bases. Assembly language programs are read by the assembler and converted to executable machine language programs during the assembly processes. Assembly language is easier to remember and manipulate than machine language.

BIT

A minimum logic element. A binary number of either Ø or 1. A bit is the smallest unit of information in a binary system of notation. It is the choice between two possible states, usually designated one (1) and zero (Ø).

BYTE

A set of contiguous binary bits, usually eight, which are operated on as a unit. A byte can also be a subset of a computer word.

BUSS

An electrical conductor or group of electrical conductors which provide a communication path between two or more devices, such as between a central processor, memory, and peripherals.

CALLING

See "SUBROUTINE CALL."

CLEARING

Setting to binary zero, as with a register or an individual bit.

CONDITIONAL BRANCH INSTRUCTION (Also Conditional Jump)

An instruction causing a program transfer to an instruction location other than the next sequential instruction only if a specific condition tested by the instruction is satisfied. If the condition is not satisfied, the next sequential instruction in the program line is executed.

CONDITIONAL LOOP

A loop which may or may not be executed depending on a condition tested by a Conditional Branch Instruction.

CONTENTS

The binary value stored in a flip/flop, register, or memory location.

CPU
Central Processing Unit. Usually contains the Timing and Control section and the Arithmetic and Logic section of a computer.

DECISION BLOCK
A diamond shaped element in a flow chart which initiates one of two possible actions based on the results of a decision. When translated to assembly language, it will include a Conditional Branch Instruction.

EXECUTE
To perform a specified computer instruction. To run a program.

FETCH
1) The action of obtaining an instruction from a stored program and decoding that instruction. Also refers to that portion of a computer's instruction cycle when that action is performed.
2) A process of addressing the memory and reading into the CPU the information word, or byte, stored at the addressed location. Most often, fetch refers to the reading out of an instruction from the memory.

FIRMWARE
A computer program stored in Read-Only-Memory (ROM).

FLIP/FLOP
A circuit whose output may be set to either a 1 or a Ø by manipulating its inputs in a specific sequence. The output is then retained until this specific manipulation is performed again. A flip/flop is used to store one bit of information.

FLOWCHART
A graphical representation of the processing steps performed by a computer program or of the sequence of logic operations implemented in hardware.

HARDWARE

The physical material comprising a computer or other electronic system.

HEXADECIMAL

The base 16 number system using Ø, 1, . . . , A, B, C, D, E, F to represent all the possible values of a 4-bit digit. The decimal equivalent is Ø to 15. Two hexadecimal digits can be used to specify a byte.

INDEX REGISTER

A register that contains a quantity which may be used to modify the memory address specified by an instruction.

INDEXED ADDRESSING

An addressing mode in which the address part of an instruction is modified by the contents in an auxiliary (index) register during the execution of that instruction.

INDEXING

The process of using an Index Register.

INDIRECT ADDRESSING

An addressing mode in which the address of the operand of an instruction is specified by the contents of memory location(s). The address of the memory location(s) is, in turn, specified by the instruction.

INPUT

Signals entering a computer, circuit, or system from the outside world.

INPUT/OUTPUT PORT

A device for connecting a computer to the outside world.

INSTRUCTION

A set of bits that defines a computer operation, and is a basic command understood by the CPU. It may move data, do arithmetic and logic functions, control I/O devices, or make decisions as to which instruction to execute next.

INTERRUPT

An interrupt involves the suspension of the normal programming routine of a microprocessor in order to handle a sudden request for service. The importance of the interrupt capability of a microprocessor depends on the kind of applications to which it will be exposed. When a number of peripheral devices interface the microprocessor, one or several simultaneous interrupts may occur on a frequent basis. Multiple interrupt requests require the processor to be able to accomplish the following: to delay or prevent further interrupts; to break into an interrupt in order to handle a more urgent interrupt; to establish a method of handling interrupt priorities; and, after completion of interrupt service, to resume the interrupted program from the point where it was interrupted.

INTERRUPT SERVICE ROUTINE

Similar to a Subroutine, except that it is initiated by a physical signal, called an Interrupt Request, rather than a Branch to Subroutine Instruction.

INTERRUPT REQUEST

A logic signal applied to the CPU which initiates an Interrupt Service Routine.

JUMP

1) An instruction which, when executed, can cause the computer to fetch the next instruction to be executed from a location other than the next sequential location. Synonymous with branch.

2) A departure from the normal one-step incrementing of the program counter. By forcing a new value (address) into the program counter, the next instruction can be fetched from an arbitrary location (either further ahead or back). For example, a program jump can be used to go from the main program to a subroutine, from a subroutine back to the main program, or from the end of a short routine back to the beginning of the same routine to form a loop. See also BRANCH INSTRUCTION. If you reached this point from branch, you have executed a jump. Now return.

LABEL

A name used to represent a specific location in memory. Labels are often used in writing assembly language programs because a) the specific location may not be known while the program is being written but may be easily identified later by its label and b) the use of labels makes assembly language programs easier to read and understand.

LATCHES

Flip/flops.

LEAST SIGNIFICANT

Of lowest value. The right-most digits of a number.

LISTING

A printout of a computer program or data.

LOCATION

A specific storage position in memory, identified by its address.

LOGIC SIGNAL

An information signal which transmits one of two possible states or meanings.

LOOP

A program segment which is executed repeatedly, usually until some condition is satisfied.

LOOP COUNTER

A register which keeps track of the number of times a loop has been executed.

LOWEST ORDER

Least significant.

MACHINE LANGUAGE

The numeric form of specifying instructions ready for loading into memory and execution by the machine. This is the lowest level language in which to write programs. The value of every bit in every instruction in the program must be specified (e.g., by giving a string of binary, octal, or hexadecimal digits for the contents of each word in memory).

MEMORY

A general term which refers to any storage media for binary data. Basic memory functional types include read/write and read-only.

That part of a computer that holds data and instructions. Each instruction or datum is assigned a unique address that is used by the CPU when fetching or storing the information.

MNEMONIC

Computer instructions written in brief, easy-to-learn, symbolic or abbreviated form. Mnemonic code is also recognizable by the assembly program. For example, ADD, SUB, CLR, and MOV are mnemonic codes for instructions which will be executed as machine code.

MULTIPLEXOR

A device which allows several signals to share a single transmission path. Typically, a multiplexor is similar to a switch which allows any one of several inputs to be connected to its output.

NESTED SUBROUTINE

A subroutine which is called by another subroutine. The called subroutine is said to be nested within the calling subroutine.

OBJECT CODE/OBJECT PROGRAM

The binary form of a source program produced by an assembler or a compiler. The object program is composed of machine-coded instructions that the computer can execute.

OPERAND

The data on which an instruction is to operate.

POINT TO

The act of specifying a particular memory location or register. For instance, a register which contains the address of a memory location is said to point to that location.

PORT

See "INPUT/OUTPUT PORT."

PROCESSING THE DATA

Manipulating or operating on data as directed by a computer program.

PROGRAM

A complete sequence of computer instructions necessary to solve a specific problem, perform a specific action, or respond to external stimuli in a prescribed manner. As a verb, it means to develop a program.

PROGRAM STATUS WORD (PSW)

A special-purpose register within the 2650 processor that contains status and control bits. It is 16 bits wide and is divided into two bytes called Program Status Upper (PSU) and the Program Status Lower (PSL).

PSEUDO-OPERATION

Also called "pseudo instruction," "assembler directive," or "Pseudo-Op." This appears to be an instruction written in assembly language. However, instead of translating directly into a machine instruction, it provides information for the assembler to use in the process of translating other assembly language instructions into machine language.

RANDOM ACCESS MEMORY

1) Memory in which any random location may be easily accessed.

2) Usually implies that any location may be written into, as well as read from. (See Read-Only Memory)

READ-ONLY MEMORY

A memory whose data is permanently stored at the time of manufacture. Data contained in it may be read, but new data may not be stored.

REGISTER

A storage location for a group of one or more bits; especially within a CPU.

RETURN ADDRESS REGISTER

A register used to save the address of the instruction to be executed following the execution of (and return from) a subroutine.

SOFTWARE

Computer programs.

SOURCE CODE/SOURCE PROGRAM

A program coded in other than machine language (in assembly or compiler language) that must be translated into machine language for use. Assembly and compiler language programs are human readable whereas object programs are machine readable.

STACK POINTER

In the 2650 microprocessor, a 3-bit value stored in the Program Status Word which points to the Return Address Register which is currently in use on the Return Address Stack.

SUBROUTINE

1) A short program segment which performs a specific function and is available for general use by other programs and routines.

2) A subprogram (group of instructions) reached from more than one place in a main program. The process of passing control from the main program to a subroutine is a subroutine call, and the mechanism is a

subroutine linkage. Often data or data addresses are made available by the main program to the subroutine. The process of returning control from subroutine to main program is subroutine return. The linkage automatically returns control to the original position in the main program or to another subroutine.

3) Programming technique that allows the same instruction sequence or subprogram to be given control and used repeatedly by other sections of the program.

SUBROUTINE CALL

Transferring program control to a Subroutine while saving the location of the instruction to be executed following the Subroutine in a Return Address Register.

TIMING PULSE

A short transition of a logic signal to its opposite state and back again which is used to trigger some event.

TWO'S COMPLEMENT

A system of binary notation which can represent both positive and negative quantities. In two's complement notation, the most significant bit represents the sign of the quantity (Ø for positive, 1 for negative) and the remaining bits represent its magnitude. To find the negative equivalent of a two's complement number invert all of its bits and add 1 to the result (complement and increment).

WORD

A set of binary bits handled by the computer as the primary unit of information. The length of a computer word is determined by the hardware design. Typically, each system memory location contains one word.

COMMENT SHEET

TITLE:

REVISION:

This form is not intended to be used as an order blank. Signetics Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments;

FROM NAME:____________________ POSITION:____________________

COMPANY NAME:____

ADDRESS:____

FOLD ON DOTTED LINES AND STAPLE

Tear off and return

STAPLE STAPLE

FOLD FOLD

FIRST CLASS
PERMIT NO. 166
SUNNYVALE, CALIF.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

signetics

a subsidiary of **U.S. Philips Corporation**

Signetics Corporation
P.O. Box 9052
811 East Arques Avenue
Sunnyvale, California 94086

Bin No. 038 MOS Microprocessor Division

FOLD FOLD

STAPLE STAPLE

signetics

a subsidiary of **U.S. Philips Corporation**

Signetics Corporation
P.O. Box 9052
811 East Arques Avenue
Sunnyvale, California 94086
Telephone 408/739-7700

Printed in USA Dec 1978