Designing With Microcomputers

An Introductory Text
on
Microcomputer Fundamentals
for
Electronic Circuit and System Designers and Managers

January 1976

signetics

TABLE OF CONTENTS

PREFACE:
WHAT THIS BOOK IS ABOUT

This text is short because it has a singular objective: to teach you what a microcomputer is and how you (or your staff) can design with it. We've put a bibliography in the back if you want to know more about how and why microcomputers are revolutionizing electronics, what new applications have opened up, how to compare microcomputers, etc.

The approach of the book is very simple. A real-life design problem (currently in volume production) is posed and solved with a real-life microprocessor, the Signetics 2650.

To help you **learn** from the text, we've incorporated several features: key words are identified in both the text and separate glossary. There are two quizzes and, finally, the text is extensively illustrated.

The book can be read in about four hours — an easy investment for learning about what many say is the most important technological innovation of this decade.

I. INTRODUCTION

With the introduction of a class of electronic components called microprocessors, the hardware implementation of physical systems, governing a wide range of applications has undergone a radical change. The objective of this book is to assist electronic system engineers, managers and other creative individuals to reorient their system implementation methodology to take advantage of the exciting possibilities offered by this novel component. This process of reorientation is accomplished by taking the reader through the main steps of a specific electronic design problem; namely, the design of an intelligent typewriter system (ITS), using a microcomputer. This particular design example was selected because: (1) the system hardware configuration is usable in a number of other applications with similar serial input/output requirements, and (2) the hardware components, mounted on a PC card, are available for evaulation and demonstration.

Before we begin, note that we are using two words, **microprocessor** and **microcomputer.**

MICROPROCESSOR

The **microprocessor** is a device which performs arithmetic, control, and logical operations.

MICROCOMPUTER

The **microcomputer,** in turn, is a collection of devices that includes a microprocessor, memory, and associated interface circuits to communicate with the "outside world." This essential distinction will become clearer as we progress.

1.1 System Development Procedure

Using the microprocessor as a key system component, the system designer can significantly reduce the hardware component count and, therefore, production costs. But during the prototype development phase, he needs to carefully design the microcomputer software, and the hardware interface between the microcomputer and the "outside world."

The fundamental trade-off that must be foremost in the mind of the designer is: *How can I configure the system so as to minimize the component count and hardware complexity by performing more functions within the microcomputer, without any significant degradation in overall system performance (or response)?*

The sequence of procedural steps to be followed in the development of a hardware prototype system are familiar to most electronic system designers and managers. For the sake of completeness, this familiar sequence is presented in Figure 1.1 for a microcomputer-based prototype system. The first block requires the designer or the manager to write a detailed description of the functions the system is to perform; Section 1.2 will document the functional specification for the aforementioned ITS.

On the basis of this specification, a suitable system hardware configuration must be defined to (1) meet the interface requirements between the microcomputer and the "outside world" and (2) provide adequate capability within the microcomputer to meet the functional specification. In general, for a particular microprocessor, some ingenuity is required to accomplish these requirements economically. (These new components, therefore, do not supercede the need for clever engineers.) The definition of a suitable microcomputer hardware configuration for the ITS System is elaborated in Chapter III.

PROGRAM

The next step is to design the microcomputer **program.** By program is meant the "customized" sequencing of logical, arithmetic and control operations of the microprocessor to meet the desired functional specification. The system designer begins by breaking down the functions required into a set of elementary procedural steps arranged in a systematic and clearly defined manner by a suitable program description; additional details concerning this facet of system design are described in Appendix B.

The microcomputer program designed in the previous step is then implemented and tested in the two following blocks of Figure 1.1. The ease with which the program is implemented and tested, largely depends on the usage of proper structuring techniques during the program design process in the previous step. Programming methodologies that result in "well-structured" programs are presented in Appendix B. Then, a microcomputer-based hardware prototype system is implemented, incorporating the previously tested microcomputer program.

Successful testing of this prototype system completes the prototype development.

This book is organized as follows: In the following section, the intelligent typewriter system (ITS)'s design problem is specified. Since the design involves usage of a microcomputer, basic computer concepts are reviewed in chapter II, this material can be skipped by the computer specialists. Chapter III describes the design of the ITS, using the Signetics 2650 microprocessor. Additional microcomputer concepts and features, not required by the ITS but useful in other applications, are also described in chapter III. Figure 1.1 relates the discussions in the various sections to the typical development process.

The main text is followed by a selected bibliography of microcomputer literature and a glossary of commonly occurring terms. Appendix A presents the Signetics 2650 Microprocessor instruction set and electrical specifications. Appendix B describes alternative methodologies for microcomputer programming. Appendix C presents the ITS program listing.

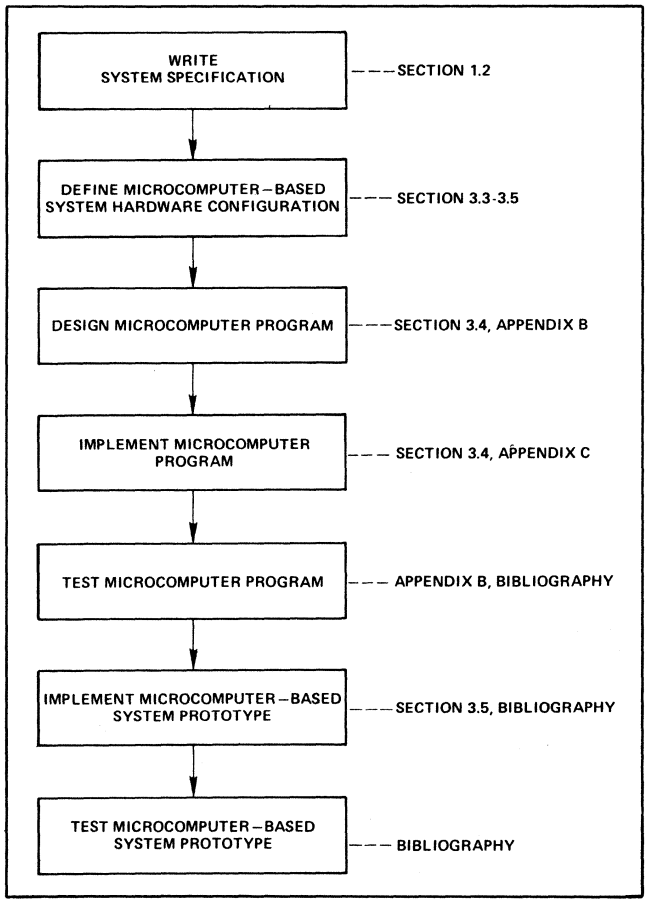


Figure 1.1 Prototype Development Procedure

1.2 The Design Problem: An Intelligent Typewriter System (ITS)

The overall design problem is to implement an intelligent typewriter system (i.e., text generating system) which outputs a "previously specified" text, with certain blank spaces that can be filled in by the user, to "customize" the text (e.g., a form letter with the name, age, and social security number of the individual to whom it is to be sent). The input medium for the "previously specified" text is to be the familiar typewriter keyboard. The output medium is to be the typewriter printing mechanism. Moreover, control characters need to be implemented into the system to allow insertion of unique characters at locations identified during text generation. Additional control characters will be required to provide an edit (i.e., erasure of the previous character entered) and system reset capability.

The above functional specification of the intelligent typewriter system (ITS) expressed in commonly used English language is reworded in more precise technical terminology in chapter III. In this section certain hardware constraints are imposed and the functional usage of the various control characters is defined. Then, in chapter III (after a review of microcomputer fundamentals in chapter II), the hardware and software configuration details, as outlined in Figure 1.1 are generated. A listing of the software portion of the ITS is included in Appendix C.

For the typewriter mechanism, we will employ a teletype (TTY) terminal. We will use this device for two reasons. First, a microcomputer must always employ an input/output (I/O) device or devices. The TTY can perform all the I/O functions for our application. Second, as a microcomputer system designer, you will ultimately have to employ a TTY or similar terminal in developing the microcomputer system itself. An understanding of the ITS/Teletype interface gives you a headstart in understanding these terminals and their operation.

Operation of the TTY is very similar to operation of a typewriter with the exception that the TTY has some additional keys. Figure 1.2 shows the TTY keyboard. The keyboard includes the familiar alphanumeric keys found on a conventional typewriter. In addition to these, there are several control keys. These are described in terms of operation of the ITS as follows:

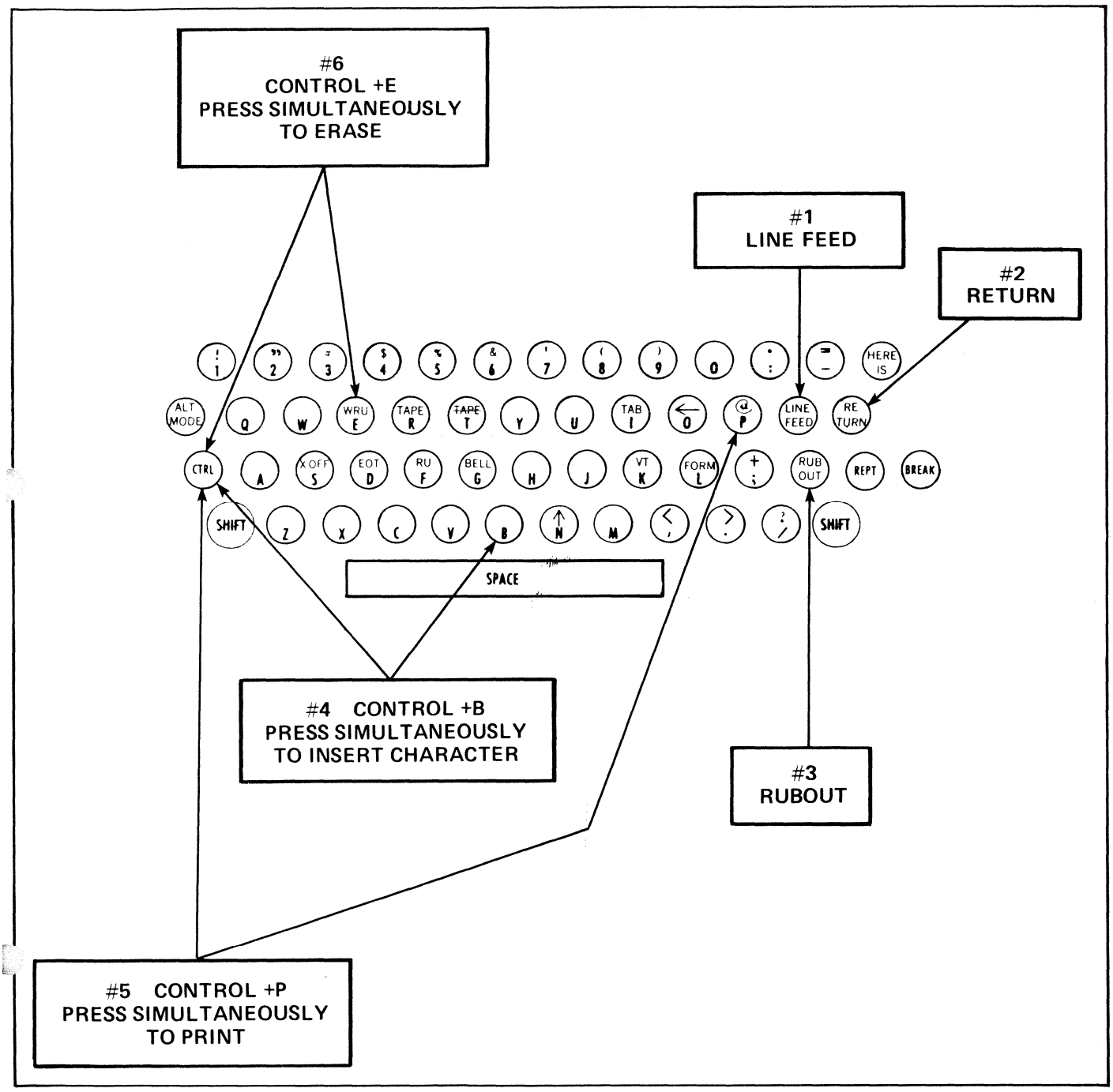


Figure 1.2 TTY Keyboard With ITS Commands

1. The LINE FEED key advances the paper, on which the TTY is printing, by one line.

2. The TTY printing mechanism moves from left to right while printing. The RETURN key moves the printing mechanism to the left hand margin.

3. Recall that the user will be typing into the microcomputer memory. (This will include letters, numerals, line feeds, and returns.) The RUBOUT key will be used to delete from memory the last typed character or control key. Additional preceding characters can be deleted by continuing to press the RUBOUT key. RUBOUT will affect the editing function of the ITS.

4. The IDS feature for producing form letters will be acheived using the following set of controls.

When the user reaches a point in the letter where unique information is to be inserted, he will simultaneously depress the CTRL key and character B. (We will refer to this combination as CONTROL + B.) This will cause the material written back from memory to stop so that the unique information may be typed in by the user. After the user has typed in the unique information, he can resume the typing from memory by depressing CONTROL + C.

5. We will provide the user with a command CONTROL + P which will initiate printout from microcomputer memory.

6. Finally, the entire storage of the ITS can be erased by depressing CONTROL + E.

The TTY terminal has a bell which can be operated on command from the microcomputer. We will ring the bell when the user:

1. Attempts to store more data characters than the ITS storage will permit. Let us limit this to 250 characters.

2. Tries to read an empty memory.

3. Attempts to delete more characters than exist in storage.

4. Attempts to continue printing after the contents of memory have been printed out.

The ITS system is now specified. Before describing the actual microcomputer design, we must first review some computer concepts. This is the subject of the next chapter.

II. MICROCOMPUTER BASICS

2.1 Introduction

This chapter develops the fundamental concepts one needs to understand and use microcomputers. The basic approach of the chapter is to develop the structure (i.e., architecture) and operation of a practical microprocessor. We begin by describing a very simple device, a hand calculator, that adds binary numbers. In the next section, the device is redesigned to operate automatically—becoming, in fact, a very primitive microcomputer. In subsequent sections, additional refinements are added until the complete microcomputer system is defined.

2.2 A Binary Hand Calculator

Everyone has operated a (decimal) hand calculator. Numbers are entered on a keyboard, operations are performed (+, ÷, =, etc.), and results are displayed. As it turns out, the operations one goes through in operating a hand calculator match very closely with what happens in a microcomputer. To show this correspondence, let's first design a very simple hand calculator, one that works with binary numbers. In fact, let's restrict operation of the device to the following: we can enter two numbers (in binary) and output the sum, i.e., out calculator will be a binary parallel adder. (We can, of course, do this with one chip, but our purpose here is to ultimately evolve to a microcomputer.)

In a real calculator design, one must first ask, how many digits should be used? The selection comes by trading off desired precision with circuit complexity (i.e., the more digits, the more complex the calculator).

BIT

We will use 8 binary digits **(bits)** since we will be ultimately describing an 8-bit microcomputer. Hence each number we wish to add will be represented by 8 bits as will their sum.

BYTE

We will refer to this group of 8 bits as a **byte**. The following examples show decimal addition along with the corresponding addition in binary.

Table 3.1 Addition Example

Decimal	Binary Equivalent
11 + 6 17	00001011 +00000110 00010001
130 + 15 145	10000010 +00001111 10010001

Note that we use 8 bits or one byte to represent each number in binary.

We can now move on to describing the design of the binary calculator. In doing so, it is necessary to introduce the reader to a logic device which will be the heart not only of the calculator at hand, but also of the microcomputer.

ALU

This device is called an arithmetic-logic unit **(ALU)** and is shown in Figure 2.1. As the figure suggests, the ALU takes inputs A and B and performs functions (add, subtract, compare, etc.) based on a function select input and outputs a result. (We will be discussing the status output later.)

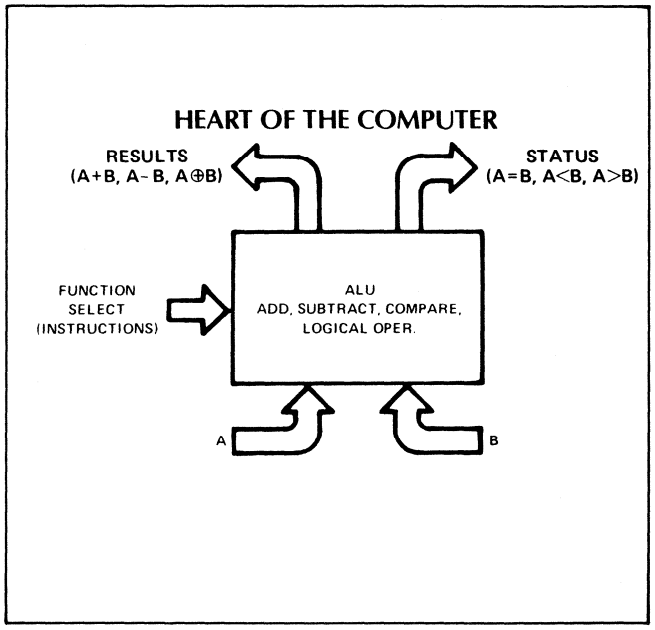


Figure 2.1 Arithmetic Logic Unit (ALU)

For our binary calculator, we will be concerned only with the addition feature of the ALU. Inputs A and B will represent the two one-byte numbers to be added and the result (A + B) will also be one byte (i.e., the inputs A + B and the output in the figure actually consists of eight parallel lines.) We will assume that the function select in the figure has whatever is required to effect the add function of the ALU.

OPERANDS

Finally, since the ALU will be performing an (add) **operation** on A and B, we will call A and B **operands.**

We could, at this point, complete the design of the binary calculator using 16 switches (8 Single Pole-Double Throw switches connected to 0 and 5 volts for each input byte) and 8 lamps driven from the buffered ALU output. However, we ultimately intend to convert this calculator into a microcomputer. For this reason, we will complicate the calculator design by supplementing the ALU with an 8-bit register R_0 as shown in Figure 2.2.

CPU

What results is called a central processing unit (**CPU**) and functions as follows:

INPUT

Looking outside the CPU, a byte can be an **input** (i.e., fed into) the CPU or

OUTPUT

output (i.e., driven out) depending upon an operation select which can have one of several states.

That is, input and output share the same set of lines, since the data flows in both directions—in and out. Looking inside the CPU, we see R_0 connected to both the input/output as well as the ALU. These lines are one byte wide.

I/O

It should be noted that the interconnections as shown are not actual electrical connections but rather a composite of all possible signal paths among R_0, the ALU, and the INPUT/OUTPUT (**I/O**) terminal of the CPU. We will shortly discuss these signal paths.

Inputs to the ALU are the operand inputs (e.g., A and B in Figure 2.1) to the ALU itself is not shown for it is driven by logic whose input is the operation

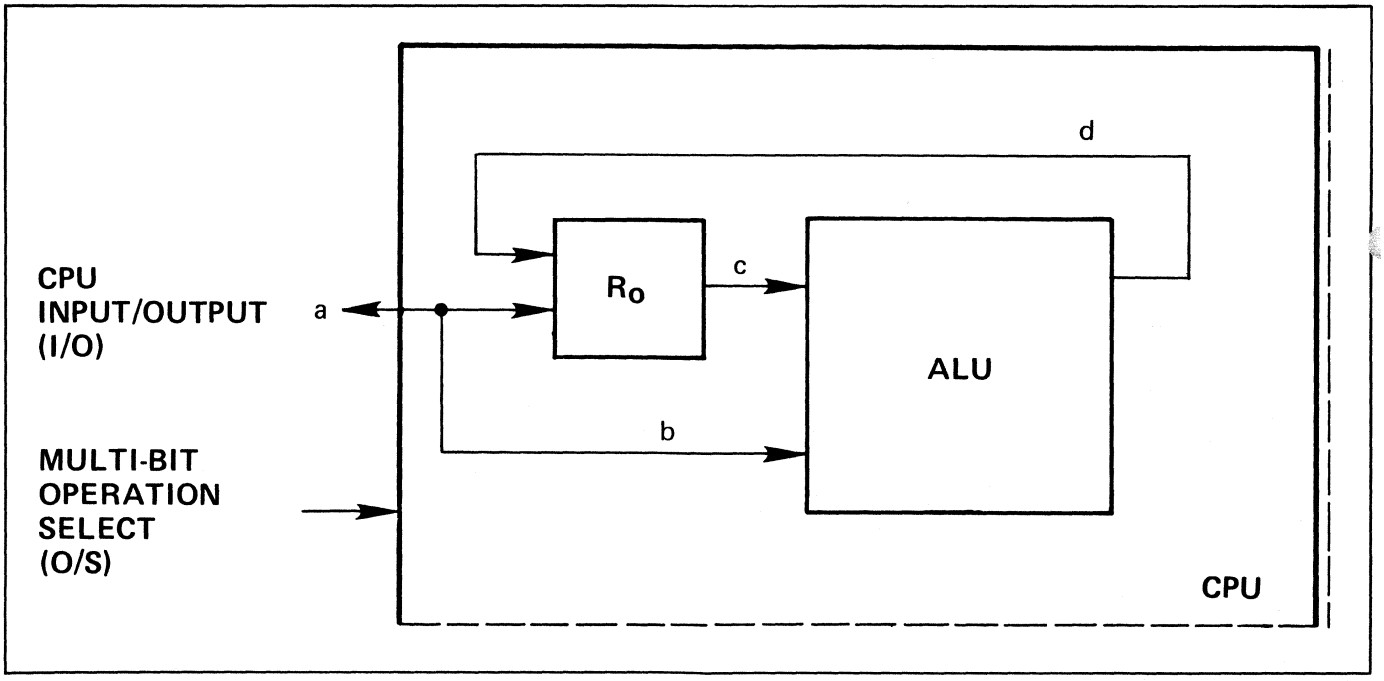


Figure 2.2 Central Processing Unit (CPU)

select input to the CPU. As indicated above, the CPU interconnections shown in Figure 2.2 represents a composite of signal paths. The actual paths themselves are governed by the state of the operation select input. These individual paths can be visualized by considering the operation of the CPU. For our calculator application, the CPU has three operating modes:

MODE 1: A byte can be **input** directly into R_0. That is, the 8 bits in R_0 will be set to whatever 8 bits are on the input. (a). In this mode paths to and from the ALU (b, c, and d) are inactive. (Logic gates enabled and disabled by the signals on the OPERATION SELECT input perform this function.)

MODE 2: The ALU will take the byte on the input a/b and the byte in R_0 (c) and sequentially perform the binary sum placing the result in R_0 (through path d). In this case, the path between input and R_0 is inactive. Note also that the prior contents of R_0 are destroyed.

MODE 3: The contents of R_0 are fed to the **output** (a). Here, all ALU paths are inactive. Also, the contents of R_0 are unaltered.

It is now a simple matter to construct a binary calculator. Figure 2.3 shows a possible implementation using switch banks and lamps. Referring to the figure, we will be sequentially putting the numbers to be added into the operand switch bank. The

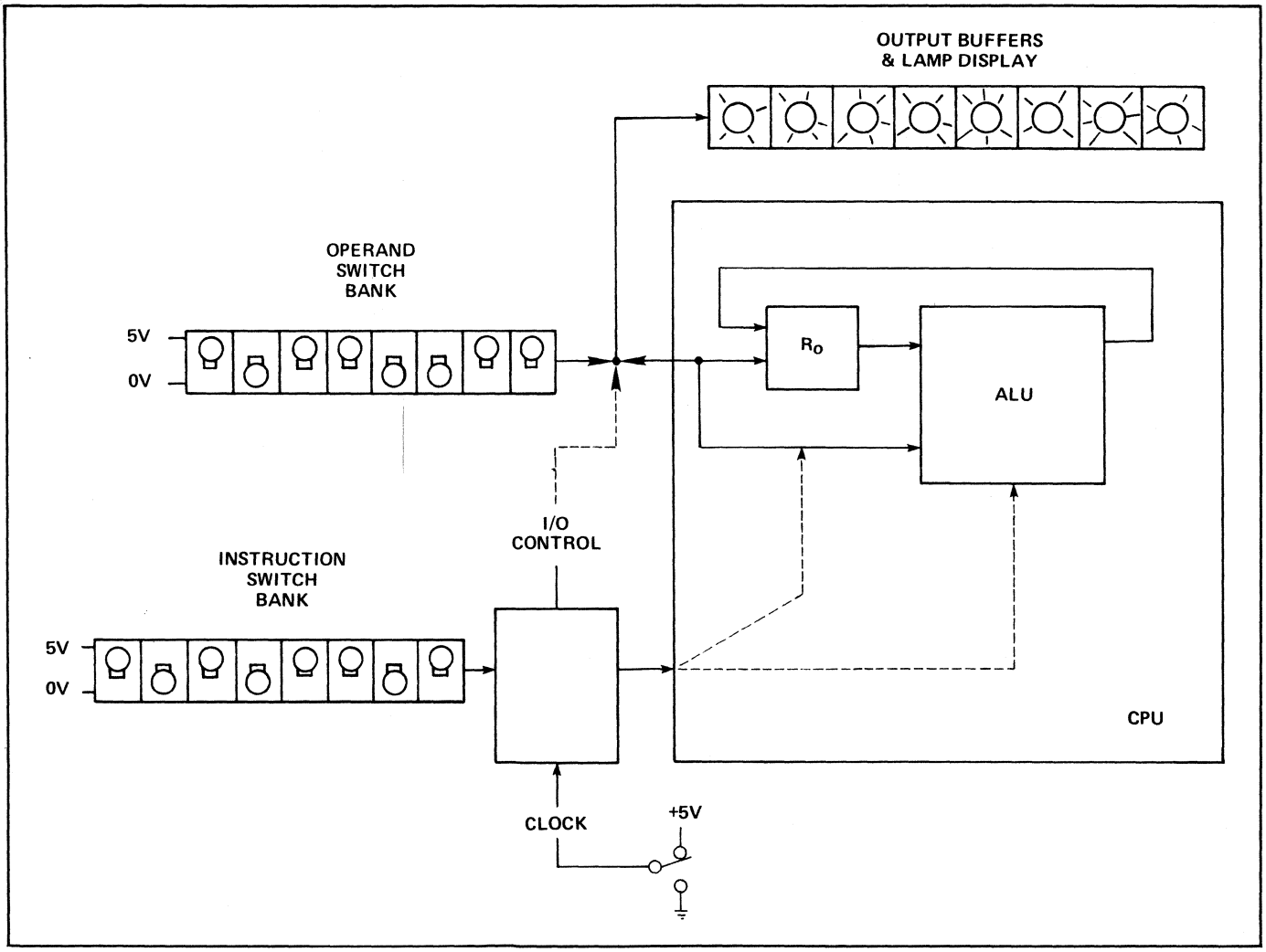


Figure 2.3 Binary Calculator

resultant output (A + B) will be displayed on the output lamp display. (Note again that input and output paths are shown as a composite; the actual path employed will be controlled by additional logic not shown.) We can select I/O paths and select the CPU operating mode using an 8 bit* (i.e., one byte long) instruction switch bank.

INSTRUCTION

Here we use the word **instruction** in the sense that the setting of the eight switches will "instruct" the CPU and other logic what to do.

CLOCK

Since we will be sequentially setting switches and operating the CPU, it will be necessary to provide the system with **clock** pulses as shown. The binary calculator is now designed.

We can turn now to operation of the system. Let's suppose that we want the binary sum of operands A and B (i.e., we want A + B). To be orderly, let's put both the operands A and B and the result (when we get it) on a scratch pad as in Figure 2.4a.

DATA

We will refer to A, B, and A + B as **data**, i.e., binary numbers that are the object of the calculation.

To be orderly, let's itemize the data using numbers 101, 102, 103 as shown. In similar fashion, we can list the instructions that have to be performed on another piece of paper as shown in Figure 2.4b. Each instruction as shown is in a shorthand notation and represents one or more unique 8-bit bytes which will be placed in the instruction switch bank and clocked into the system. These binary instructions will be discussed in a later section; the shorthand designation will be discussed shortly. Note

DATA (ON A SCRATCHPAD)

ITEM NO.	DATA
101	A
102	B
103	A+B

Figure 2.4a

INSTRUCTIONS

ITEM NO.	INSTRUCTION (SHORTHAND)
000	LOAD RO, 101
001	ADD RO, 102
002	STORE RO, 103

Figure 2.4b

Figure 2.4 Data and Instructions

*At this point, the use of an 8 bit "instruction" was somewhat arbitrary. To perform the add operation, fewer bits are actually required. However, as we will see later, there is a relationship between the length of the data word A, B, etc. and the instruction in a real microcomputer.

that like the data, the instructions are itemized with decimal numbers 000, 001, 002. Note also that data and instructions have different item numbers.

The calculator can now be operated by writing data A and B next to item numbers 101 and 102 and executing the instructions sequentially starting with item 000 as follows:

1. **Instruction Item 000:** Put the data in data item 101 in the operand switch bank and place (i.e., load) it into register R_0. A now resides in R_0.

2. **Instruction Item 001:** Put the data in data item 102 into the operand switch bank. Add this value to the contents of R_0 (A) and place the result in R_0. A + B now resides in R_0.

3. **Instruction Item 002:** Output the contents of R_0 (A + B) to the lamp display and place (i.e., store) the value at data item 103.

Again, note that in actuality each instruction consists of one or more eight-bit bytes set into the instruction switch bank. (As mentioned, these will be discussed in greater detail later on.) For the time bing, we will refer to these instructions by the English equivalents shown in Figure 2.4b.

LOAD

LOAD: puts CPU input into R_0.

ADD

ADD: adds CPU input to contents of R_0 and places the result into R_0.

STORE

STORE: places the contents of R_0 on the data scratch pad.

Let's move on now to automating the calculator and in the process, develop a basic minicomputer.

2.3 An Automatic Calculator

In this section, we will modify the calculator of Figure 2.3 such that it will operate automatically. In doing so, it is clear that we must mechanize the process of getting both the data and instruction lists of Figure 2.4 in and out of the hardware. The key to this is a new hardware element: **Memory.**

MEMORY

For our purposes, we will describe **memory** as a device which contains 8-bit bytes. In particular, these 8-bit bytes comprise the data and instruction bytes of Figure 2.4.

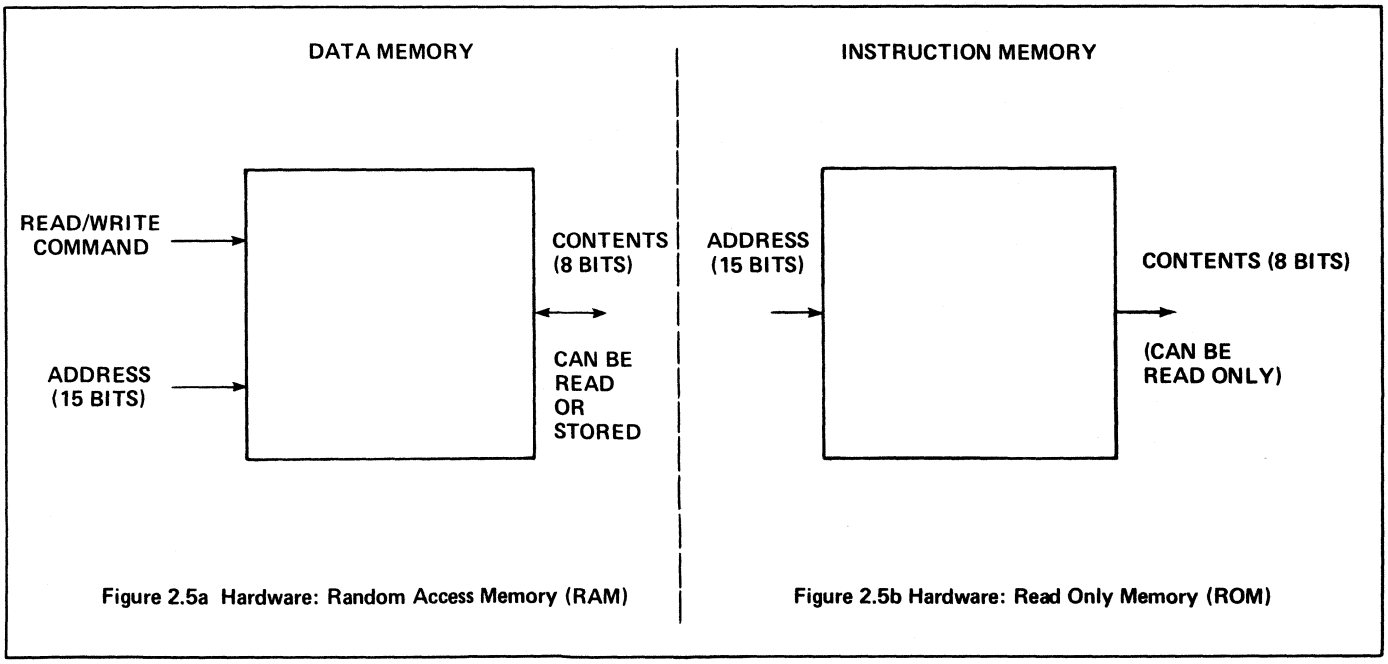


Figure 2.5a Hardware: Random Access Memory (RAM)

Figure 2.5b Hardware: Read Only Memory (ROM)

Figure 2.5 Memory

CONTENTS

We will refer to these bytes as memory **contents**. Since these bytes are stored in hardware, the question naturally arises—how does the remaining hardware know which byte is which? and where in memory they are located.

ADDRESS

This question is resolved by giving each byte an **address.**

The address here corresponds exactly with the data item and instruction item numbers shown in the lists in Figure 2.4 (e.g., 101, 102, 001, 002, etc.). For the microcomputer system we are developing, we will use a 15-bit address. (That is, in our system, we could use a memory having as many as 32,768 unique contents.)

Finally, just as we had a data list and instruction list in Figure 2.4, we will employ a **data memory** and **instruction memory**. These are depicted conceptually in Figure 2.5.

DATA MEMORY

For the **data memory**, we can either output (i.e., read) the contents of each binary address or input (i.e., store) a byte which will become contents at that address.

We can read or store depending upon whether the read/$\overline{\text{write}}$ command (Figure 2.5a) is high or low. It should be noted here that when data are read, the contents are undisturbed; however, when data are stored, the original contents are lost. Figure 2.5b shows the instruction memory.

INSTRUCTION MEMORY

Instruction memory will only be read. (Recall in the lists of Figure 2.4 that we **read** A and B off the data list and **stored** the sum A + B; we only **read** the instruction list.)

At this point, let's now interface the data and instruction memories with the calculator hardware of Figure 2.3. We will be able to read data memory contents directly into R_0. Hence, we can eliminate the operand switch bank. In a similar fashion, let's replace the instruction switch bank with an 8-bit register into which we can place the contents of the instruction memory.

INSTRUCTION REGISTER

We will call this register an **instruction register** (IR). It will serve the same purpose in the automatic calculator that the instruction switch bank served in the hand calculator.

At this point, we now have data memory contents feeding R_0 and the instruction memory contents feeding an instruction register. We must finally "address" data memory and instruction memory in order to determine which data goes into R_0 and which instructions go into IR.

OPERAND ADDRESS REGISTER

We will add two new registers for this purpose: an **operand address register** (OAR) for addressing data (or operand) memory

INSTRUCTION ADDRESS REGISTER

and an **instruction address register** (IAR) for addressing instruction memory. These registers will have 15 bits to match the number of address bits.

Memory and foregoing registers are shown interconnected in Figure 2.6. Note first that the output lamps have been removed since we can now store R_0 directly into data memory. Studying the figure, we see that the automatic calculator has three basic blocks: (1) CPU, (2) Memory, and (3) Control.

MICROPROCESSOR

It should be noted here that CPU and Control Sections comprise a **microprocessor.**

MICROCOMPUTER

A **microcomputer** on the other hand consists of a microprocessor, memory, and I/O. The CPU is unchanged from previous examples.

The memory block depicts both instruction memory contents (upper half of memory block) and data memory contents (lower half). Associated with each content is a unique address. Note that addresses and contents correspond exactly with the data and instruction lists we made for the hand calculator (Figure 2.4).

BUS

Addresses are selected by activating an address line or addres **bus** which as noted earlier is 15 bits wide.

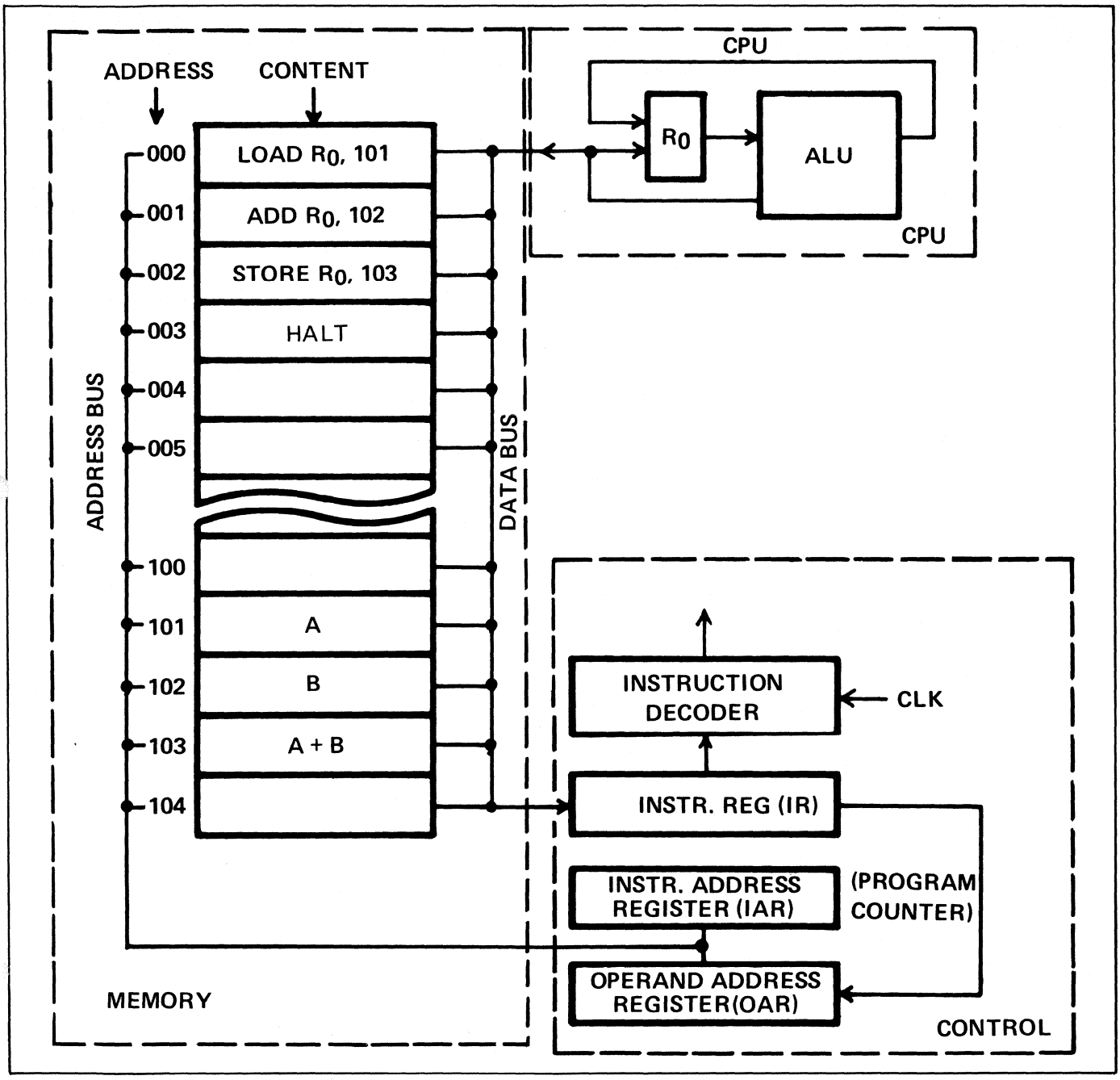


Figure 2.6 Automatic Calculator

Memory contents can be individually placed on an 8-bit bidirectional data line

or **data bus**; in the case of the data memory, data can be both placed onto and from the **data bus.**

Note that the data bus as shown is a composite. Only one content will be on the data bus at a time.

The control unit contains the 8-bit instruction register (IR) and the 15-bit instruction address register (IAR) and 15-bit operand address register (OAR). Since the instructions are in numerical order, the instruction address register is incremented by one after each addressing of instruction memory. For this reason, the IAR is sometimes called a program counter. The final item in the control block is the instruction decoder. The decoder uses the contents of the IR as input to control other parts of the

system (CPU operation select, memory read/write, signal path select, etc.). Note in the figure that both the IAR and OAR are on the address bus. In actuality, only one register at a time is on the bus. Finally, note the signal path between the IR and OAR. This path is shown since in the actual microcomputer the contents of the OAR are governed by the IR. We will discuss this point in more detail in Section 2.10.

At this point, let us now discuss the operation of the automatic calculator. Refer to Figure 2.6. We will assume that the instructions are in instruction memory as shown. A and B are in data memory. The system will compute A + B and store the result in data memory as follows:

1. The instruction address register will be initially set to a value corresponding to memory location 000. This will cause contents of location 000 (Load R_0, 101) to be placed on the data bus and into the instruction register. The instruction itself specifies the loading of R_0 with data stored at address 101; hence, the operand address register will take on a value corresponding to address 101. (This is accomplished by transferring a portion of the instruction—namely, "101" to the OAR. This is accomplished automatically when the Instruction Decode circuitry decodes the instruction.) The instruction decoder will then cause the system to place the contents of OAR on the Address Bus which, in turn, puts A on the data bus and finally into R_0. The instruction address register will automatically increase by one to 001. A is now in R_0.

2. With the instruction address register at value 001, the instruction ADD R_0, 102 will be put on the data bus and placed into the instruction register. The operand address register will take on value 102 causing B to appear on the data bus where it will be input to the (lower) ALU input. The ALU will add what is in R_0 and B and return the result to R_0. This again is caused by signals from the instruction decoder which is looking at the ADD instruction. The instruction address register will again increment by one to address 102. The sum A + B is now in R_0.

3. The instruction address register will put instruction STORE R_0, 103 on the data bus and into the instruction register. The operand address register will take on value 103 and the data in R_0 (A + B) will be stored in data memory location 103.

HALT

Note the addition of a **HALT** instruction at memory address 003.

HALT stops excution of instructions. (Without a HALT, the instruction register would be loaded with unknown contents that could cause the system to behave unpredictably, e.g., storing something other than A + B at data address 103.) The system will accordingly have behaved as an automatic calculator. The system is, in fact, a microcomputer. One element is lacking, however: communication with the outside world. This aspect is covered by an example in the following section.

2.4 A More Practical Example: Mixing Vat

Let us, at this point, leave the calculator problem behind and look at a somewhat more practical problem.

Figure 2.7 depicts a mixing vat having two pipes placing material into the top of the vat and two pipes extracting material from the bottom of the vat. We will monitor flow rates in all pipes (A and B at the top, C and D at the bottom as shown). We will use the microcomputer system to calculate total flowrate into the vat (A + B) and total flowrate out (C + D). If, at any time, flowrate into the vat exceeds flowrate out [i.e., (A + B) > (C + D)] we will cause a bell to ring.

Looking at the above requirements, the microcomputer system must have instructions that will read in the flowrates, perform the appropriate sums, and make a comparison. A convenient technique for obtaining these instructions is to first diagram the sequence of individual steps in a flowchart as shown in Figure 2.8.

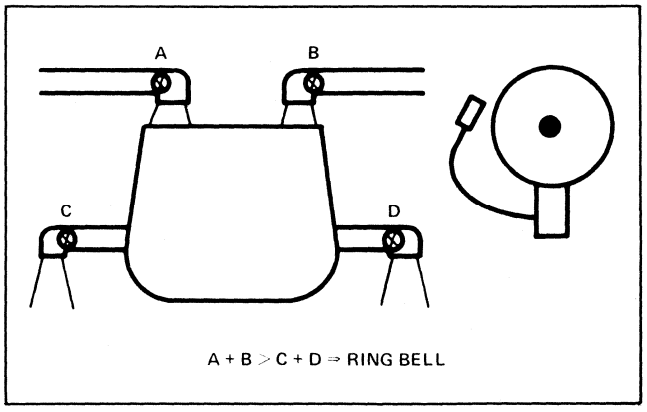


Figure 2.7 Mixing Vat Example

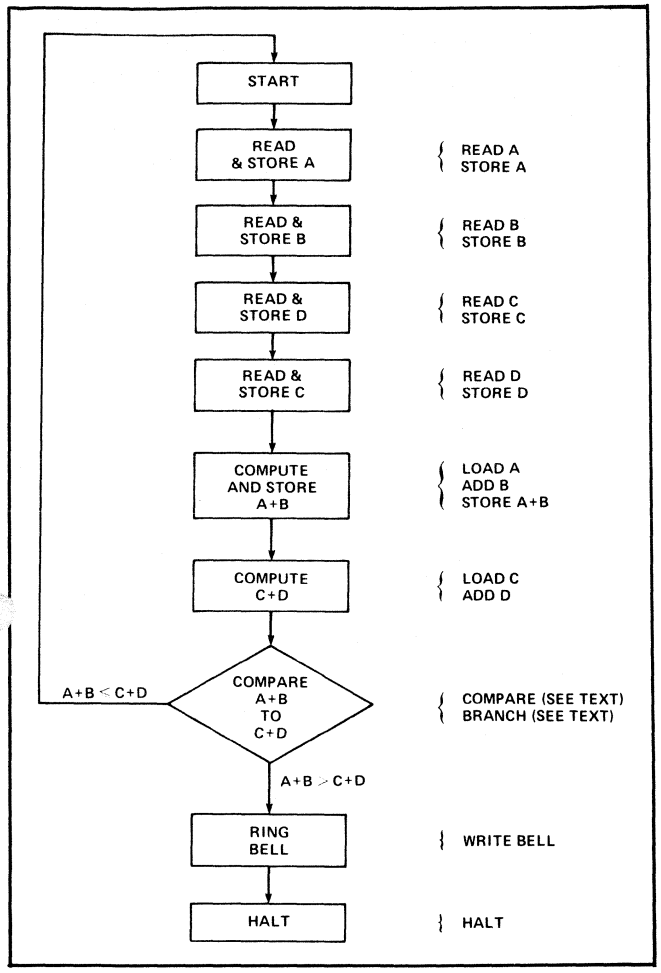


Figure 2.8 Flowchart for Mixing Vat Example

FLOWCHART

Instructions can then be written from the **flowchart** as shown.

Our microcomputer system can perform this task as shown in Figure 2.9. The CPU, Control, and Memory Sections are the same. Two things, however, have changed:

1. The data bus and address bus go out to an input/output (I/O) block containing the flowrate meters and the bell.

2. We have more instructions than in the previous example.

The I/O block comprises external logic (I/O port) connected to the four flowmeters and the bell. This logic is designed to selectively place A, B, C, D, or the bell on the data bus as a function of inputs from the address bus. Note here that the address bus is being used in conjunction with a control line "C" (discussed in more detail in Section 2.10) to not only select data and instruction memory contents but to also select devices outside of the system. For this reason, we must select "I/O addresses" distinct from instruction and data addresses. The I/O addresses for the figure are tabulated as follows:

Table 2.2 I/O Addresses

I/O Address	Item on Data Bus
017	Bell
200	A
201	B
202	C
203	D

Look now at the instructions. From the foregoing, it should be apparent that all the READ instructions are accessing the flowmeter data and not memory. The reader is already familiar with READ, STORE, and LOAD instructions and should at this point be able to trace through the microcomputer operation through instruction address 012. (Try it!) After execution of instruction 012, we have (A + B) at data memory address 105 and register R_0 contains (C + D).

These instructions at 013 (COMPARE R_0, 105) and 014 (BRANCH 000) are new. Moreover, they are interrelated (in a manner that will be explained in Section 2.9). Basically, these instructions work as follows.

BRANCH

The **BRANCH** 000 causes the instruction address register to be reset to 000 (i.e., BRANCH XYZ resets the IAR to XYZ).

Instructions beginning at 000 are then repeated. The BRANCH instructions, however, is executed as a result of what happens with the COMPARE instruction as follows.

COMPARE

The **COMPARE** R_0, 105 instruction compares the magnitude of the contents of R_0 (C + D) with the contents of memory address 105 (A + B).

If the contents of R_0 are greater than (i.e., flow out is greater than flow in) the contents at 105 the BRANCH instruction is executed. Otherwise, the BRANCH instruction is ignored and the next instruction (WRITE) is executed. The WRITE BELL, 017 instruction is self-explanatory.

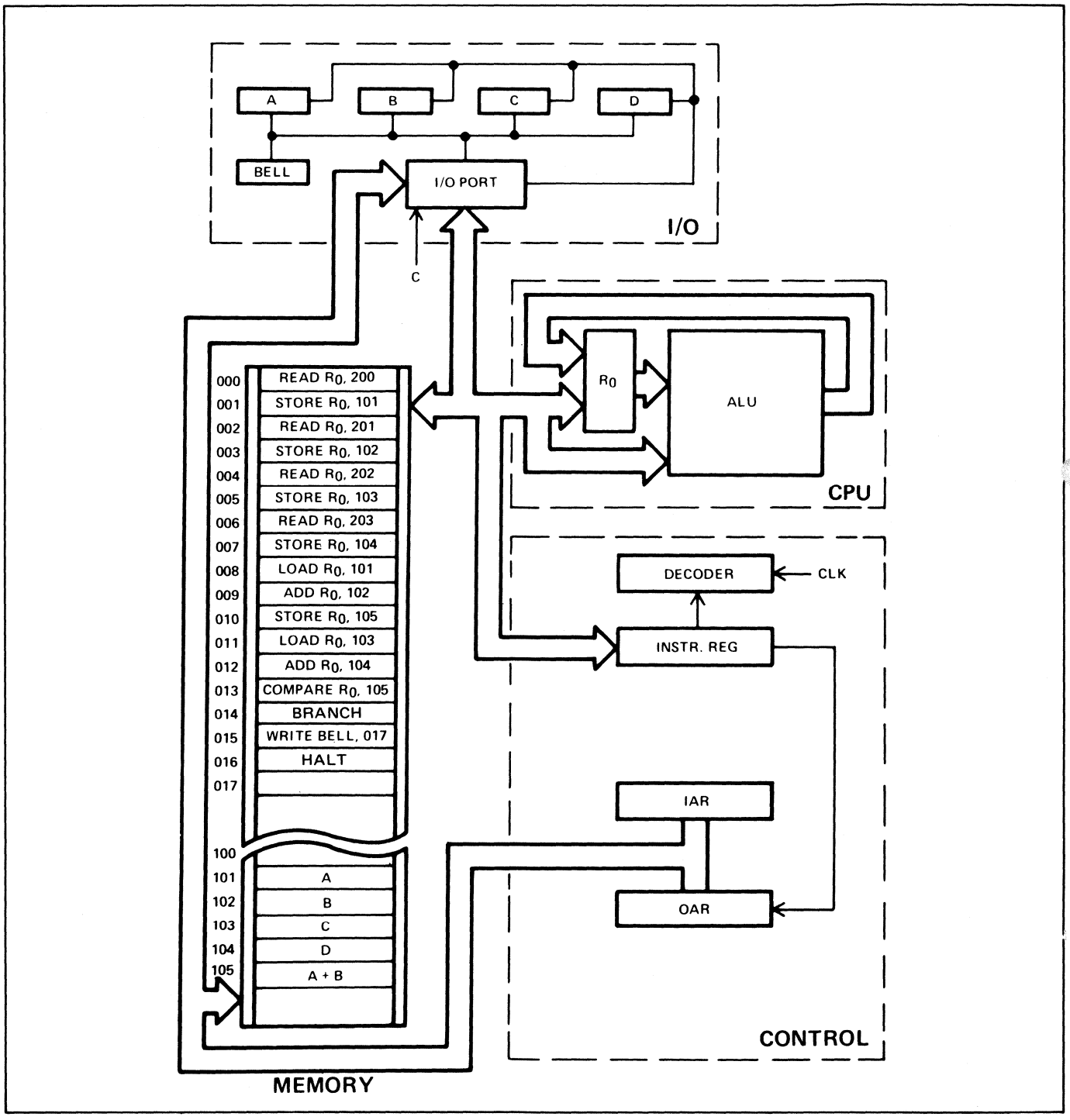


Figure 2.9 Mixing Vat Program Sequence

The reader should now see that the system of Figure 2.8 will be continuously computing (A + B) and (C + D) and comparing magnitudes until (A + B) > (C + D), at which point, the bell will ring.

This example introduces two new things that can happen in microcomputer systems:

1. The system can make *decisions* (compare then BRANCH or don't BRANCH).
2. The sequence of instructions can be changed (e.g., due to BRANCH) and repeated.

Moreover, the example begins to illustrate the most important point in this book: EXCEPTING FOR THE I/O, THE MICROCOMPUTER HARDWARE IS FIXED; THE ACTUAL SYSTEM DESIGN LIES MAINLY IN DESIGNING THE SEQUENCE OF INSTRUCTIONS.

This point becomes more obvious as we consider some further instructions in the next section.

2.5 A More Powerful Microcomputer

PROGRAMMING

As stated in the previous section, design with microcomputers will principally comprise designing a sequence of instructions or, to define a new term, **programming.**

Consequently, we can make our microcomputer in Figure 2.9 a more powerful device by increasing the number and diversity of instructions.

We can at this point very easily introduce several new instructions. Recall that the microcomputer CPU was developed using an arithmetic logic unit ALU. So far, we have discussed use of the ALU only for addition. Referring back to Figure 2.1, it should be obvious that other operations can be performed. For example, we can **SUBTRACT** operands (A – B). Additionally, logical operations can be performed.

AND

For example, we can **AND** operands A and B.

Recall that operands A and B are 8-bit bytes by "and-ing" corresponding bits giving an 8-bit byte as a result (A • B) as in the following example:

Table 2.3 AND Operation Example

ALU Input		ALU Output
A	B	A • B
10101100	11001010	10001000

Note also that the 8-bit operands need not be binary numbers. In fact, the operands can represent data characters (e.g., as in the intelligent typewriter system to be discussed later), codes, logic states, etc.

INCLUSIVE OR

EXCLUSIVE OR

We can, in similar fashion to the above, develop other logical instructions such as **"INCLUSIVE OR"** and **"EXCLUSIVE OR"**:

Table 2.4 OR Function Example

FUNCTION	ALU INPUT		ALU OUTPUT
	A	B	A + B
INCLUSIVE OR	10101100	11001010	11101110
EXCLUSIVE OR	10101100	11001010	01100110

These instructions are shown in Table 2.6 which includes instructions discussed in previous sections.

ROTATE

Also included are **ROTATE** instructions (ROTATE LEFT and ROTATE RIGHT).

These instructions move the bits in register R_0 one bit to the left or right as described in the table and as indicated in the following.

Table 2.5 ROTATE Instruction Example

	BEFORE ROTATE	AFTER ROTATE
ROTATE RIGHT	11010101	11101010
ROTATE LEFT	11010101	10101011

It should be clear that, as a group, the instructions shown in Table 2.6 make the basic microcomputer hardware we've developed very flexible in terms of what can be performed on the outside world. By now, the reader has probably been wondering. "How fast are these instructions performed?" As it turns out, time to execute each instruction varies from instruction to instruction. On the average, however, instructions are performed in about 7 microseconds. In the previous examples, it would, therefore, take about 21 microseconds for the automatic calculator to sum and store results; each mixing vat computation would be done in about 110 microseconds.

Instructions have been discussed so far by their English equivalents (READ, ADD, STORE, etc.). As we know, the instructions, themselves, are made up of one or more bytes that are loaded into the instruction register. We discuss this in the next section.

Table 2.6 Microcomputer Instruction List Example

INSTRUCTION	FUNCTION
LOAD R_0	Loads R_0
STORE	Places the contents of R_0 into memory
ADD	Adds to R_0 what is on data bus and puts result in R_0
SUBTRACT	Subtracts what is on data bus from R_0 and puts result in R_0
AND	"Ands" what is on data bus with R_0 and puts result in R_0
INCLUSIVE OR	Performs "Inclusive OR" between data bus contents and R_0 putting result in R_0
EXCLUSIVE OR	Performs "Exclusive OR" between data bus contents and R_0 putting result in R_0
COMPARE	Compares data memory content with R_0 as prerequisite to branch
ROTATE RIGHT	Shifts bits in R_0 one bit to right. Least significant bit moves to most significant bit position
ROTATE LEFT	Shifts bits in R_0 one bit to left. Most significant bit moves to least significant bit position
BRANCH	Causes instructions to begin execution at another instruction address
WRITE	Places contents of R_0 onto data bus
READ	Places contents of memory (or I/O) into R_0
HALT	Stops instruction execution

2.6 Binary Instructions

As we know, instructions are made up of one or more 8-bit bytes fed into the instruction register. While a detailed treatment of binary instructions is beyond the scope of this book (the reader is referred to more detailed literature such as the Signetics 2650 MICROPROCESSOR manual), we endeavor in this section to at least give a general flavor of how these instructions are actually structured. (Actual instruction formats are in Appendix A.)

Instructions can be made up of one or more 8-bit bytes which are put (one-at-a-time, of course) into the instruction register. The first byte into the IR basically tells the microcomputer (a) the operation to perform and (b) the number of bytes in the instruction. The second byte in a two byte instruction generally consists of a data value to be operated on.

Example

The instruction **LOAD** into register R_0 binary number 10110010 would be written as a two byte binary instruction.

00000100←FIRST BYTE: Tells Microcomputer

a. What to do (LOAD)

b. Instruction has two bytes (number to be stored is in second byte)

c. Where to load (R_0)

10110010←SECOND BYTE:

Binary Number to be placed in R_0

There are also single byte instructions:

Example

We can ROTATE contents of register R_0 one bit to the right with binary instruction:

01010000←SINGLE BYTE

a. Performs ROTATE operation

b. Indicates single byte instruction

A three byte instruction usually uses the second and third bytes to derive a memory address. Since an address has 15 bits, the third instruction byte

can be used for the least significant bits of the address; the remaining bits of the address are taken from 7 least significant bits in the second byte.

Example

The instruction STORE the contents of R_0 into data memory address 010010101110110 would be written as a three byte binary instruction:

11001100 FIRST BYTE:

Instructs store operation and tells computer the following two bytes are a data address.

00100101 SECOND BYTE:

Last 7 bits are upper part of address.

01110110 THIRD BYTE:

Lower part of address.

Note that the second and third bytes are passed to the operand address register (see Figures 2.6 and 2.8).

MACHINE INSTRUCTIONS

These binary instructions are often called **machine instructions.**

Example

Machine instructions for the binary calculator of Figure 2.6 would appear as seen in Table 2.7.

Table 2.7 Example of Instructions

Machine Instructions	English Instructions (Figure 2.6)
00001100 00000000 01100101	LOAD R_0, 101
10001100 00000000 01100110	ADD R_0, 102
01010000	ROTATE RIGHT R_0
11001100 00000000 01100111	STORE R_0, 103

Now the bulk of our design activity with microcomputers is going to be devoted to the writing of instructions. It is rather obvious looking at Table 2.7 that even though the machine instructions are what we finally want in instruction memory, it would be an extremely tedious proposition to write error free instructions.* It would be certainly easier to somehow write the instructions in English as in the right hand side of the table. We can, in fact, write English-like instructions using the approach discussed in the next section.

2.7 Assembler Instructions

Before discussing how we can write more English-like instructions, it is first useful to define a few terms. We have seen that we can write machine instructions.

ASSEMBLER INSTRUCTIONS

We will shortly demonstrate that it is possible to write these in English-like **assembler instructions** which can then be converted to binary instructions.

In microconverter jargon, we have two "languages" we can use: **machine language** and **assembly language.**

PROGRAM

The instructions themselves will constitute a **program.**

ASSEMBLY LANGUAGE

What we will do is the following; we will "write" our program in **assembly language**

MACHINE LANGUAGE

and use another computer (not necessarily the microcomputer) to convert the assembly language to **machine language.**

ASSEMBLER

The computer that makes this conversion will do so using another program called an **assembler.**

This conversion process is depicted in Figure 2.10.

SOURCE PROGRAM

Here the assembly language program is often referred to as a **source program.**

OBJECT PROGRAM

The machine instructions (or code) are referred to as the **object program.**

* The problem can be somewhat alleviated by using an abbreviated notation like hexadecimal. The basic problem of generating error-free code remains.

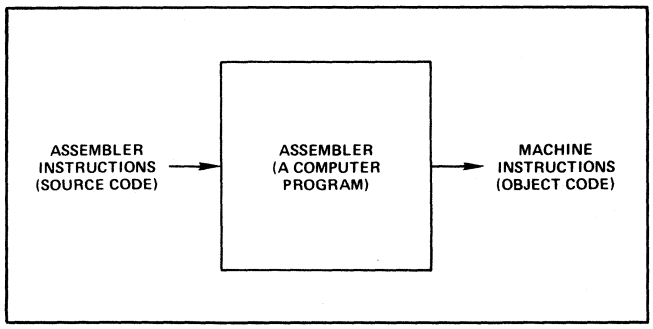


Figure 2.10 Conversion of Assembler Instructions to Machine Instructions

The assembler converts the source program into binary object program which, in turn, can be placed into the microcomputer instruction memory.

It is beyond the scope of this book to show in detail how to write the basic instructions shown in Table 2.1 in assembly language. (See instead the Signetics 2650 Microprocessor Manual. Also, assembly language instructions are summarized in Appendix A.) We can, however, give the reader insight into the general structure of assembly language by showing the assembly language instructions corresponding to the machine language instructions shown in Table 2.7. We do this in Table 2.8 as follows:

Table 2.8 Assembly Language Instructions for Binary Calculator

English Instructions (Figure 2.6)	Assembler Instructions* (Source Code)	Machine Instructions (Object Code)
LOAD R_0, 101	LODZ 101	00001100 00000000 01100101
ADD R_0, 102	ADDZ 102	10001100 00000000 01100110
STORE R_0, 103	STRZ 103	11001100 00000000 01100111

The assembler converts these into these.

*In the actual application, memory addresses will be written in hexadecimal.

Note in the table the similarity between the English instructions and the assembler instructions.

To summarize this section, we have demonstrated that the tedium and possibilities for error in writing machine instructions can be eased by writing instructions in assembly language and converting these to binary. Since the bulk of the effort in design with microcomputers comprises programming, the assembler becomes a significant cost/effective design tool.

2.8 Saving Instruction Memory—The Subroutine

From the previous discussion, we have seen that our microcomputer can address 32,768 bytes. In actual design, however, we will employ only as much memory as is required (i.e., we will attempt to minimize the number of memory chips).

SUBROUTINE

As it turns out, it is almost always possible to reduce the size of instruction memory by employing a programming technique called the **subroutine.**

This technique can be illustrated by referring to Figure 2.11. Look at the first column in the figure. This column represents a segment of instruction memory with distinct binary instructions which we have labeled A, B, C, D, E. Notice that the boxed instruction sequence D, A, C, B, E appears 3 times, i.e., we are using 15 memory locations to store the same sequence of 5 instructions.

We can reduce the total number of instructions by using a branch command as seen in the second column of the figure. Here, we write the repeated sequence D, A, C, B, E beginning at instruction address X. Each time that the repeated instruction sequence must be executed, we BRANCH to address X and perform the sequence.

RETURN

The instruction following the sequence is a **return** instruction which tells the system to resume executing instructions with the instruction following the last executed BRANCH instruction.

SUBROUTINE

The sequence D, A, C, B, E that starts at location X is called a **subroutine.**

Compare the first and second columns: we have made the repeated sequence a subroutine and eliminated six instruction memory locations. These

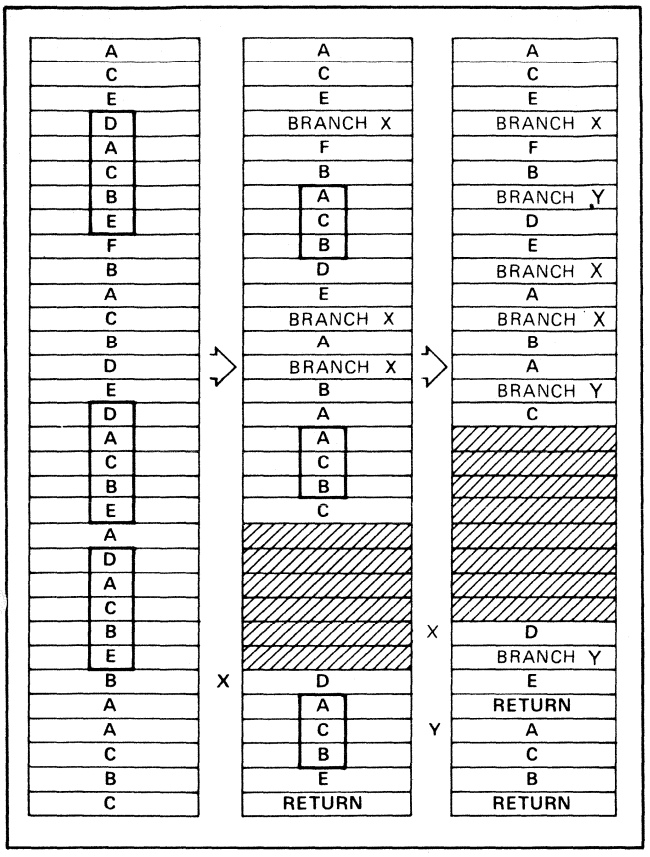


Figure 2.11 Subroutines

saved locations are shaded in the figure. We can make further economies by noting that in the second column, the instruction sequence A, C, B is also repeated. We can further reduce memory by letting A, C, B be a subroutine at location Y as shown in the third column. Note again that the RETURN instruction takes us back to the address immediately following the **last** BRANCH instruction.

NESTING

Notice in the third column that we have located a subroutine within a subroutine, a process called subroutine **nesting.**

Observe also that in the third column, we have reduced instruction memory to 75 percent of that used in the first column.

We should point out here that the use of subroutines in this manner is not merely a "frill" but in actuality a technique which can save instruction memory in virtually every microcomputer application.

Let's now turn to an important question: How do we build this subroutine capability into the hardware we've already developed? The key to this question is the fact that everytime we BRANCH to a subroutine location (X or Y in Figure 2.11) we must somehow save the instruction address that follows the BRANCH instruction (so that we can return to normal operation later).

RETURN ADDRESS STACK

We will do this by incorporating a register bank in the system called a **return address stack** (RAS).

We can conceptualize the return address stack by comparing it to a cafeteria tray holder where clean trays are loaded and extracted from the top of the tray holder. In terms of subroutine operation, we will write the return address on a cafeteria tray and put it on the stack of trays already there. When it is time to RETURN, we will go to the tray holder top and get the return address. If we are nesting subroutines, we will successively place return addresses on trays and push them down on the stack of trays. Each time a RETURN address is executed we will pull the top tray and use its address for the next instruction. It should be clear to the reader that we can nest as many subroutines as we have trays.

The hardware implementation of the return address stack is shown in Figure 2.12. The stack itself consists of a register bank fed by the instruction address register.

STACK POINTER

The specific register employed is governed by a counter called a **stack pointer** which operates (as a rotating counter) to effect the cafeteria tray analogy.

The hardware of Figure 2.12 shows the mechanism by which subroutines are implemented. We can employ this hardware without concern for its operation the instructions noted in the following table.

Table 2.9 Subroutine Instructions

Instruction	Function
BRANCH to SUBROUTINE X	Causes the program to begin execution of the subroutine beginning at instruction X.
RETURN	Placed at the end of the subroutine. Causes return of the program to the instruction address immediately following the last BRANCH to SUBROUTINE instruction.

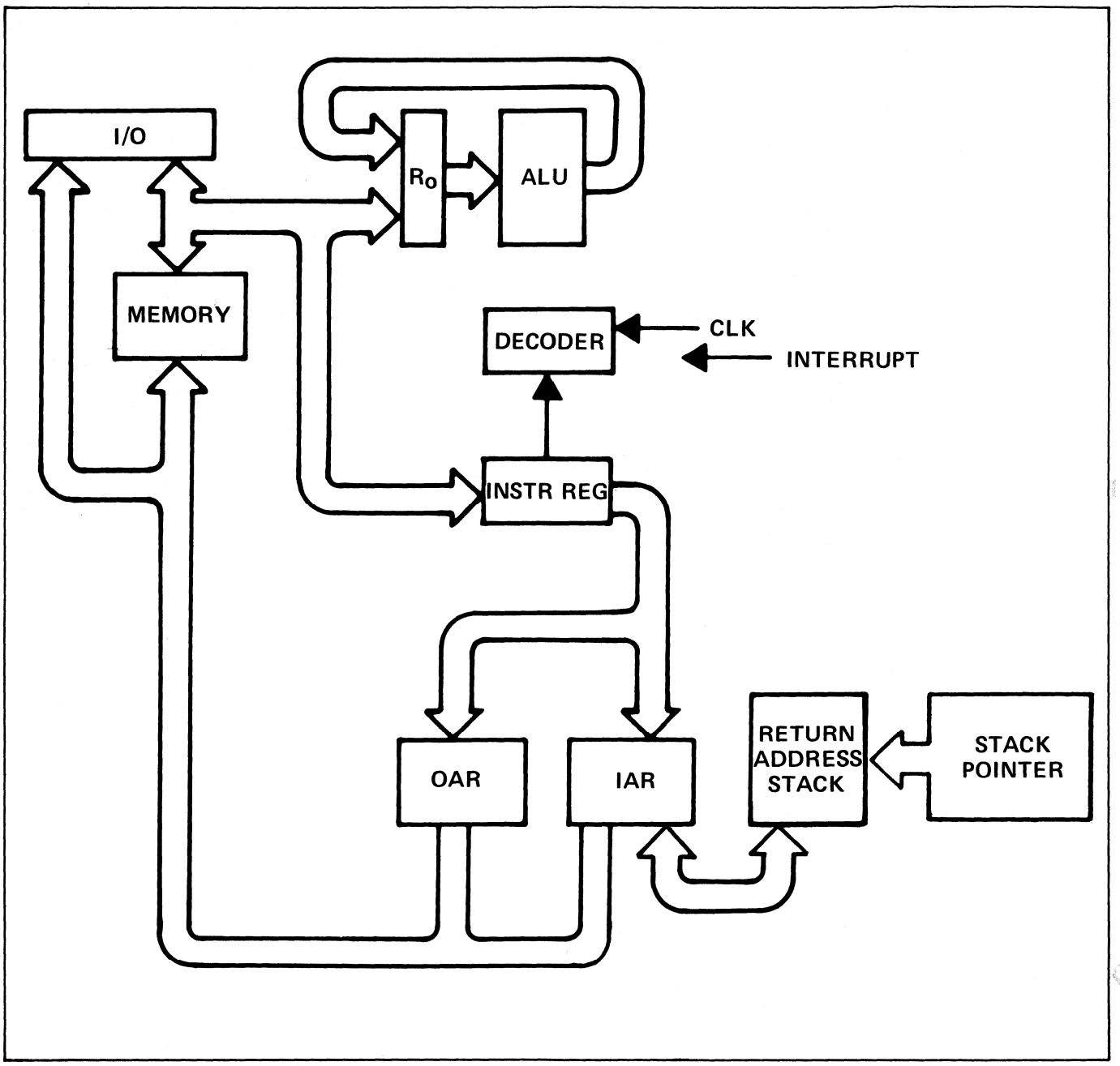


Figure 2.12 Implementation of Subroutine Hardware

The use of the subroutine instruction is illustrated in the intelligent typewriter system design discussed in the next chapter.

In the following section, we complete our description of the microcomputer hardware by describing a powerful adjunct to the system, program status.

2.9 Program Status Word

PROGRAM STATUS WORD

As a final element in our microcomputer system, we will add a special purpose register which we will call a **program status word** (PSW).

WORD

(A **word** is a collection of bits.)

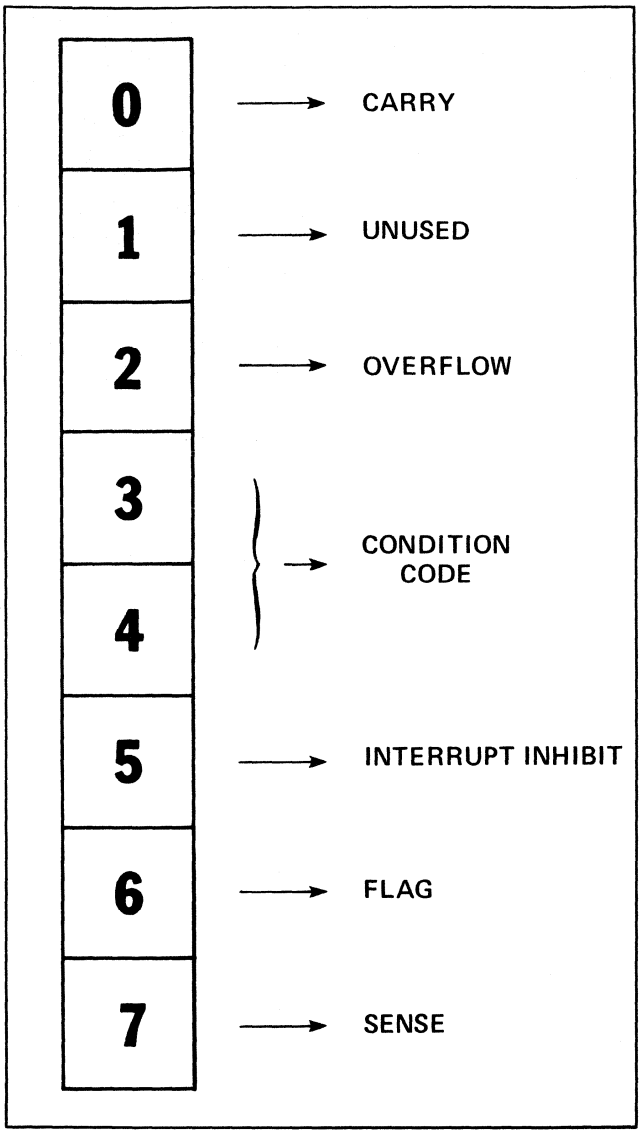


Figure 2.13 Typical Program Status Word

The bits in the PSW can serve a variety of purposes as will be discussed shortly. What is important to first note is that:

1. We will be able to put PSW bits into R_0 and vice versa. Hence, computations can be made based on what is in the program status word.
2. We will develop instructions that will permit us to test and alter bits in the PSW.

A sample program status word is shown in Figure 2.13 and is 8 bits long (since it will be put in R_0). The functions of each bit are indicated in the figure and described as follows:

Carry (C) — Set by execution of any add or subtract instruction resulting in a carry or borrow out of the high order bit of the ALU.

Compare Control (COM) — Used with COMPARE command: 0 = arithmetic compare; 1 = logical compare.

Overflow (OVF) — Employed for signed arithmetic operations; is set when a result exceeds a range of operands.

Condition Code (CC) — Used to interlink COMPARE and BRANCH instructions (See Section 2.4).

Interrupt Inhibit (II) — As shall be discussed in Chapter III, we can use an external signal to change (i.e., interrupt) the mode of the microcomputer operation. When the interrupt inhibit is set, the system will not respond to an external interrupt signal.

Flag — The flag bit is a latch driving output to a pin on the microprocessor chip. Use of the flag is illustrated in the next chapter.

Sense — This bit is connected directly to a pin on the microprocessor chip.

Note that all bits in the PSW are not used in this microprocessor.

As discussed earlier, we can transfer information between register R_0 and the PSW. This can be done through the following instructions.

Table 2.10 Program Status Instructions

Instruction	Function
LOAD PS	Causes current contents of PSW to be replaced with contents of R_0.
STORE PS	Causes contents of PSW to be transferred into R_0.

At this point, we have developed a basic microprocessor system. This system is summarized in the next section.

2.10 Summary—The Microprocessor

The microprocessor we have developed thus far is shown in Figure 2.14 within the dashed lines. This entire system (i.e., within the dashed lines) is usually made on **one** chip.

The figure includes some additional components not previously discussed. First of these are the holding register and data bus register. These registers are used to hold data and addresses continued in multi-byte instructions and are a means of transferring data and address from instruction memory onto the data bus and into the OAR, respectively. Second, observe that the figure includes blocks R1, R2, and R3 connected to R_0 and the ALU input. These registers actually comprise two bands of three registers each. They are used as supplements to R_0 (e.g., can be used as source or destination for arithmetic operations, I/O transfer, etc.).

From what has been developed so far, it should be clear that we will design with this microprocessor by:

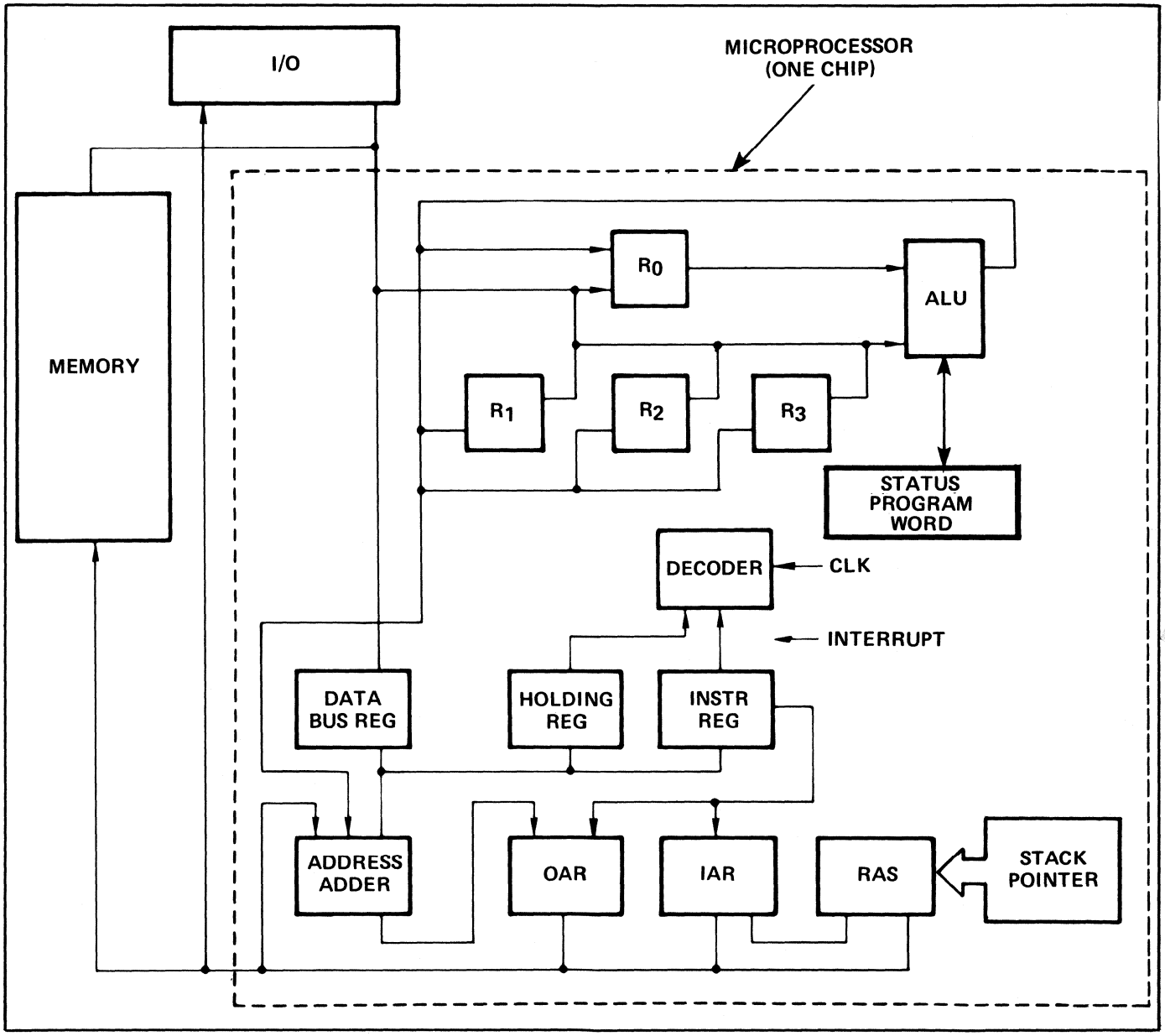


Figure 2.14 Basic Microprocessor Diagram

1. Designing the sequence of instructions, i.e., programming, and
2. Designing the electrical interface between the microprocessor and the memory and I/O.

For the first requirement, we really need only the conceptual picture of the unit shown in Figure 2.15 and the instructions. The dashed block in the upper left of the figure contains the program status, discussed earlier. The instructions we have described in the text are summarized in Table 2.11.

The electrical interface design will be made on the basis of the electrical pinouts for the microprocessor chip. This aspect is covered in more detail in the next chapter.

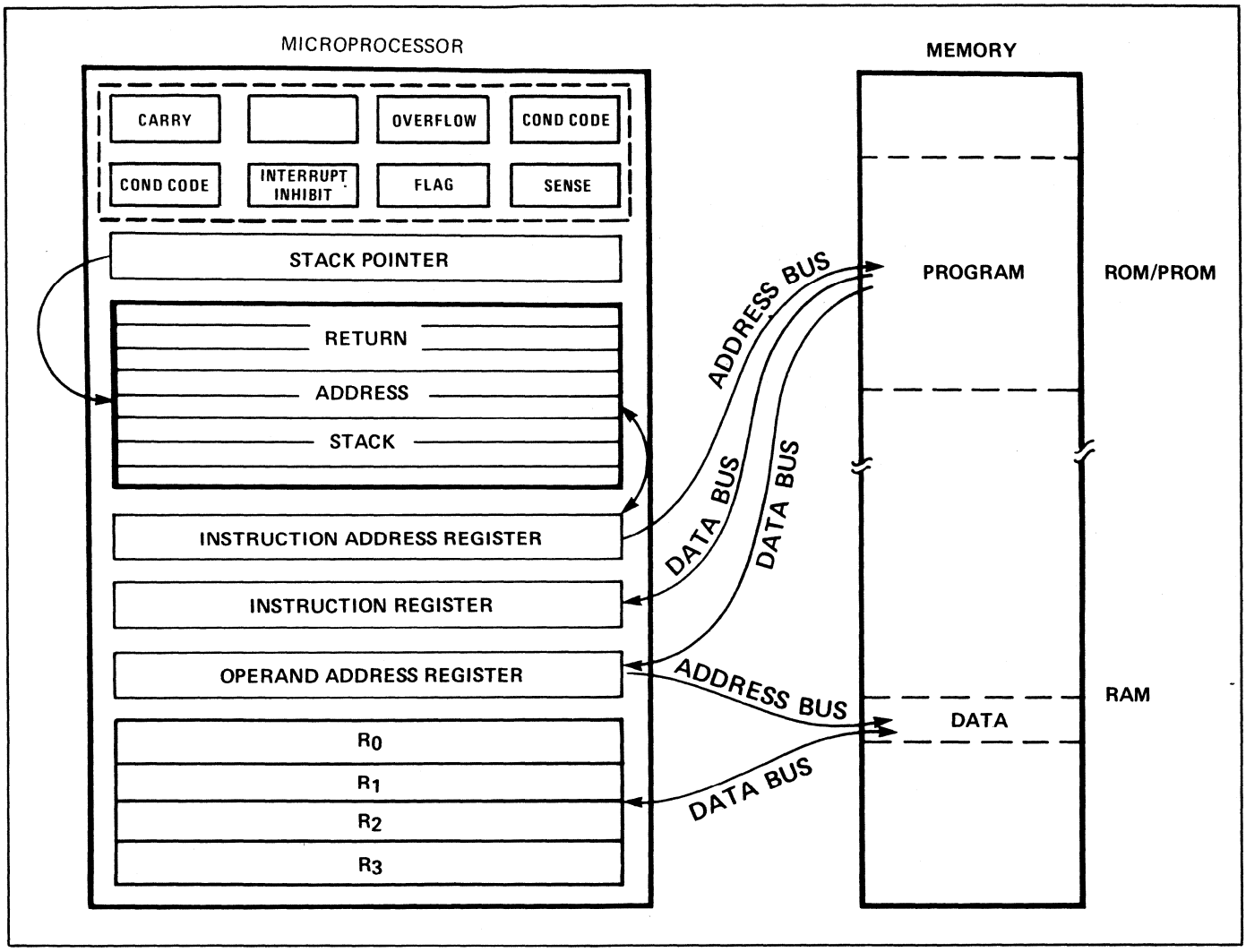


Figure 2.15 A Programmer's Conceptual Model

Table 2.11 Microprocessor Instructions—Summary

Instruction	Function
LOAD	Loads R_0
STORE	Places the contents of R_0 into memory
ADD	Adds to R_0 what is on data bus and puts result in R_0
SUBTRACT	Subtracts what is on data bus from R_0 and puts result in R_0
AND	"Ands" what is on data bus with R_0 and puts result in R_0
INCLUSIVE OR	Performs "Inclusive or" between data bus contents and R_0, putting result in R_0
EXCLUSIVE OR	Performs "Exclusive or" between data bus contents and R_0, putting result in R_0
COMPARE	Compares data memory content with R_0 as prerequisite to branch
ROTATE RIGHT	Shifts bits in R_0 one bit to right—least significant bit moves to most significant bit position
ROTATE LEFT	Shifts bits in R_0 one bit to left—most significant bit moves to least significant bit position
BRANCH	Causes instructions to begin execution at another instruction address
WRITE	Places contents of R_0 onto data bus
READ	Places contents of memory (or I/O) into R_0
BRANCH TO SUBROUTINE X	Causes the program to begin execution of the subroutine beginning at instruction X
RETURN	Placed at the end of the subroutine—causes return of the program to the instruction address immediately following the last BRANCH TO SUBROUTINE instruction
LOAD PS	Causes current contents of PSW to be replaced with contents of R_0
STORE PS	Causes contents of PSW to be transferred into R_0
HALT	Stops instruction execution

QUIZ FOR CHAPTER II – MICROCOMPUTER BASICS

(Answers on Following Page)

1. What is a byte?
2. What is an ALU?
3. What is a CPU?
4. What is I/O?
5. What is an instruction?
6. What is the function of the LOAD instruction?
7. What are MEMORY CONTENTS?
8. What is a MEMORY ADDRESS?
9. What is AN INSTRUCTION REGISTER?
10. What is the distinction between an INSTRUCTION REGISTER (IR) and an INSTRUCTION ADDRESS REGISTER (IAR)?
11. What is the function of the OPERAND ADDRESS REGISTER (OAR)?
12. What is the difference between a MICROPROCESSOR and a MICROCOMPUTER?
13. What is a bus?
14. What does a BRANCH XYZ command do?
15. What is meant by PROGRAMMING?
16. What two basic functions does the ALU perform?
17. What are MACHINE INSTRUCTIONS?
18. What are ASSEMBLER INSTRUCTIONS?
19. What is an ASSEMBLER?
20. What is a SUBROUTINE?
21. What is the main advantage of the subroutine?
22. What is SUBROUTINE NESTING?
23. What is a RETURN ADDRESS STACK (RAS)?
24. What is the PROGRAM STATUS WORK (PSW)?
25. What is DIRECT MEMORY ACCESS (DMA)?
26. What is an INTERRUPT?

ANSWERS TO QUIZ ON CHAPTER II

1. A group of binary digits.
2. **A**rithmetic **L**ogic **U**nit.
3. A **C**entral **P**rocessing **U**nit, consisting of an ALU and holding registers.
4. **I**nput/**O**utput.
5. A group of bits which decoded direct operation of the CPU and other logic.
6. Puts data in R_0.
7. The stored bits.
8. A group of unique bits that define a specific memory content.
9. A register which contains instruction bytes.
10. The IR contains an instruction byte; the IAR contains the **address** of an instruction byte.
11. The OAR contains the address of data.
12. A **MICROPROCESSOR** consists of CPU and associated control circuitry; a **MICROCOMPUTER** consists of a microprocessor, memory, and I/O.
13. Parallel lines over which multiple bits can be transmitted (e.g., 8-bit **data** bus; 15-bit **address** bus).
14. Sets the IAR to XYZ such that the instruction sequence continues beginning with the instruction at address XYZ.
15. Designing the sequence of instructions.
16. Arithmetic operations and logical operations.
17. Binary instructions.
18. English-like statements which can be converted to machine instructions.
19. A computer program that converts ASSEMBLER INSTRUCTIONS into MACHINE INSTRUCTIONS.
20. A subroutine is a subprogram comprising a sequence of instructions that is usually executed more than once during microcomputer operation.
21. To save instruction memory.
22. The placing of subroutines within subroutines.
23. A RAS is a register bank used to store return address during subroutine operation.
24. The PSW is a register containing bits corresponding to numerous microprocessor functions.
25. DMA is a microcomputer operating mode which permits direct interfacing between memory and I/O.
26. A signal to the microprocessor to suspend the current computation and to execute a more urgently needed computation.

III. DESIGN AND IMPLEMENTATION OF THE INTELLIGENT TYPEWRITER SYSTEM (ITS)

In the previous chapter, the basic features and capabilities of a typical microprocessor, namely the Signetics 2650, were explained. Now, we can proceed with the design problem posed in Section 1.2 of Chapter I. The relationship between the steps in the microcomputer system development process and the material in this chapter was noted in Figure 1.1. We will begin by considering the interface requirements for the teletype keyboard and typing mechanism; this requirement must be met by both the conventional design using standard circuitry (i.e., LSI, MSI, SSI) as well as that using the microprocessor as a system component. Then, we will consider a system level block diagram of the conventional design and make an estimate of the IC packages required.

At this point, we will begin considering the incorporation of a microcomputer to implement the system specification. The first step will be to select a suitable microprocessor, based on the guidelines that we shall develop. Then, we will describe pertinent features of the selected microprocessor, namely, the Signetics 2650.

Following this, we will consider two possible hardware configurations using this microprocessor. This will be followed by the software program design and implementation details. Finally, we will conclude the chapter by reviewing additional features useful in other classes of applications.

3.1 System Overview

Based on the specification in Chapter II, we can depict the system hardware block diagram, as in Figure 3.1. Essentially, the system consists of a teletype (to enter and type the text), control circuitry (to implement the desired functions) and memory (to store the text).

The Teletype (TTY) is a standard device which encodes each of the keyboard character keys into a unique bit pattern which is seven bits long, together with a parity bit (see glossary) for error control. Similarly, when the teletype receives characters encoded in this manner, the typewriter mechanism is activated to print the appropriate symbol.

SERIAL I/O

This standardized **serial data input/output** procedure is graphically depicted in Figure 3.2.

Referring to Figure 3.1, we note that, when the operator pushes a key, a unique serial bit pattern is sent to the control circuitry.

The control circuitry must wait until the entire bit pattern is received and then send it over the same serial channel to the typewriter print mechanism so that the user can visually verify that the correct character was received by the control circuitry.

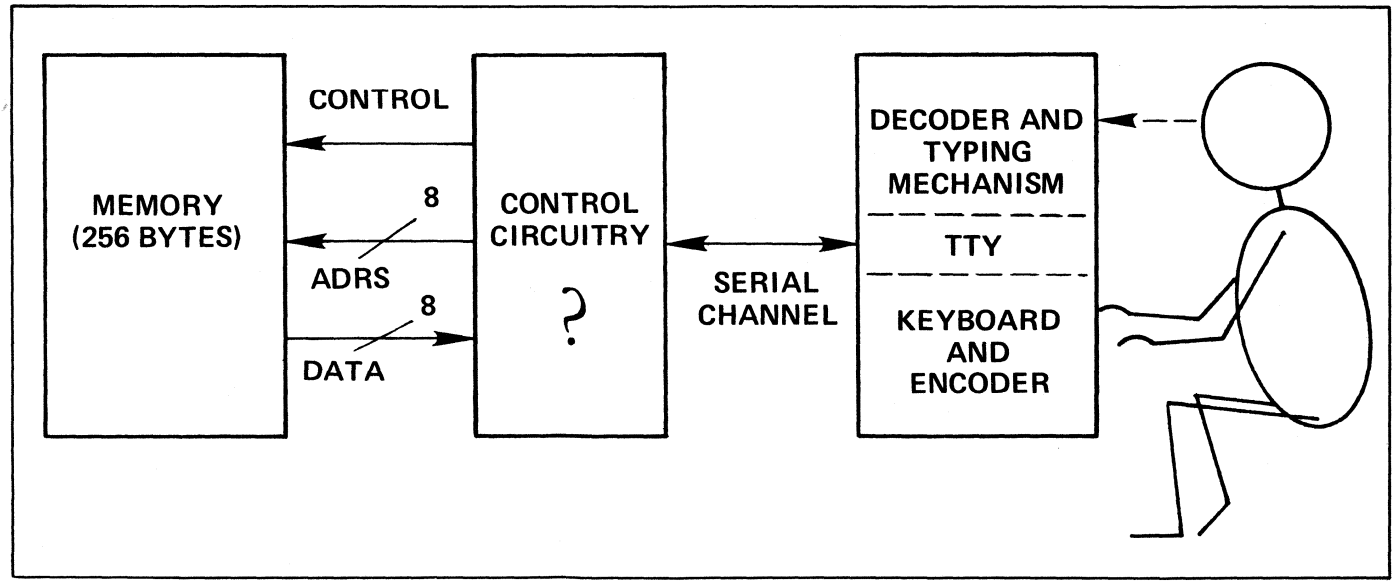


Figure 3.1 ITS Block Diagram

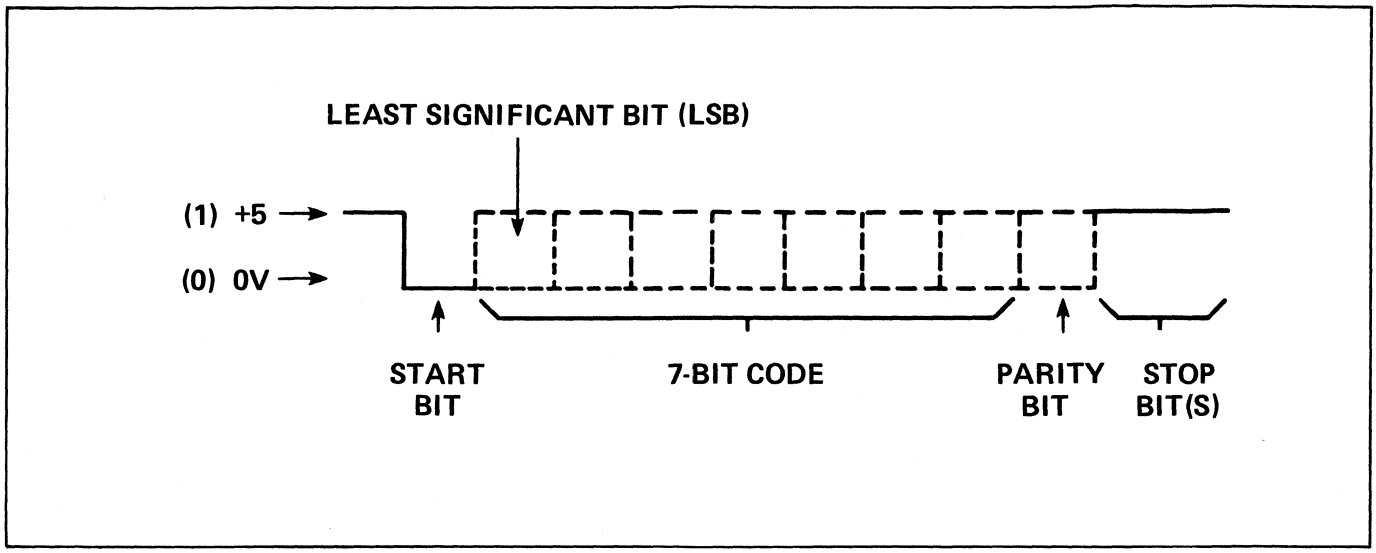


Figure 3.2 Teletype Serial Data I/O Transfers

ECHO

This process of retransmitting the received data is called **echoing.**

For convenience, a summary of the ITS commands, discussed in Chapter I, is tabulated in Table 3.1. This command specification, together with the teletype serial input/output process described above, gives us adequate information concerning the user and the teletype interface. Referring to Figure 3.1, we note that whatever the hardware implementation of the control circuitry, at least 256 bytes of memory will be required to store the text and the corresponding commands, and then send them back to the teletype print mechanism at the request of the user.

The above description completes the discussion of the common parts of the system implementation. In the following sections, three possible hardware implementations with the associated software details will be considered. The first implementation uses conventional hardware circuitry and, thus, no software is required. Then, after selecting a particular microprocessor, we will describe two microprocessor-based implementations. The first of these implementations, using the Signetics 2650 Microprocessor as a system component, implements a number of functions previously performed by hardware by the microprocessor software program. Then the latter implementation described takes advantage of some of the unique capabilities of the Signetics Microprocessor to significantly reduce the hardware complexity.

Table 3.1 ITS Command Summary

KEY	FUNCTION
Rubout (delete)	Erase last character in memory and echo the erased character. Additional preceding characters can be erased by continuing to depress the delete key.
Control and E	Erase entire memory.
Control and B	Used to indicate beginning of inserted message. Is not printed, but stored in memory. Stops print out when read from memory. Required **once** for each unique information entry point.
Control and C	Continues print out of memory after entry of unique information.
Control and P	Prints out contents of text memory.
Control and R	Software reset. Clears text buffer and restarts program.

Note: Bell will ring if any of the following are true.
1. Entering more than 250 characters in memory.
2. Requesting print out of an empty buffer.
3. Attempting to delete more characters than there are in memory.

To keep this text at a reasonable length, we cannot discuss these designs in minute detail; but the material in this chapter, together with that in the appendices, is sufficient to complete the design. Additional material pertinent to this application, including the hardware itself, is available from Signetics Corporation. For the latter microprocessor-based design, we will specifically consider the programming aspects of the serial input/output interface; this will give the reader a flavor for the nature of the software programming task.

3.2 ITS Random Logic Implementation

RANDOM LOGIC

Random logic is made up of: (1) SSI circuits such as inverters, gates, and flip-flops,

(2) MSI circuits such as decoders, multiplexers, registers and counters, and

UART

(3) LSI circuits such as memories and **universal asynchronous transmitters and receivers (UART)**.

The random logic implementation of the Intelligent Typewriter System requires, first of all, a serial/parallel converter. This is an LSI integrated circuit which converts from the serial transmission mode (one bit of information at a time) of the teletype to the parallel mode (several bits at a time) of the memory and vice versa.

One possible serial/parallel converter that could be used is the TR1602 asynchronous receiver transmitter. The TR1602 has 40 pins of data lines, control lines, and power supply lines. Dual power supplies of +5 and - 12 volts are required. Control lines are for receiving and transmitting data, error indications, clock, reset, and data format control.

As noted in Section 3.1, for all implementations, each memory word is required to be eight bits wide. A suitable memory component is the Signetics 2606 static RAM. Its organization is 256 words of 4 bits each. So two packages will provide the necessary 256 bytes (8 bits wide) of storage space for the text.

The largest and most complicated portion of the ITS is the control. It can be designed from TTL, SSI, and MSI integrated circuits. Figure 3.3 shows the hardware block diagram for thie ITS using a conventional logic approach. Remember, each block contains many integrated circuit packages.

First of all, it is necesary to control the TR1602 Asynchronous Receiver Transmitter. The 37 lines of data and control are controlled by three functional blocks: (1) Receive Data Control, (2) Transmit Data Control, and (3) Miscellaneous Control, each controlling its respective function.

A clock is required to drive the TR1602 and possibly the rest of the system. The clock block performs this function.

Memory Control controls the 2606 memory. Addressing the memory, data flow control, read or write operation select, and chip enable are the functions this block provides.

The Character Storage Control block controls storage of characters received from the TTY into memory. These are the characters that will make up the printed page when the print command is issued later.

Control-Character Storage Control controls storage of control-characters received from the TTY into memory. This type of character will not be printed when printout is requested, however. Control characters control page format and provide Stop control (insertion of special user information into the letter after a stop). Control Characters are stored in memory.

The Control-Character Control is a major functional requirement of the ITS. It provides the control functions of character delete, memory erase, continue (after Stop), and printout.

Error control performs the error indication tasks of memory overfill attempt, empty print attempt, and erroneous delete attempt.

The coordinating control block is another major functional block in the ITS. It performs the coordination of all the functional blocks in the system.

In summary, the conventionally designed ITS consists of a TTY, TR1602 serial/parallel interface, a memory, and a large control section. The control section must be large and complex to handle the functions of the ITS. And it must be designed from scratch out of a large array of SSI and MSI circuits such as inverters, gates, flip-flops, multiplexors, decoders, counters, registers, etc. Seventy-five IC

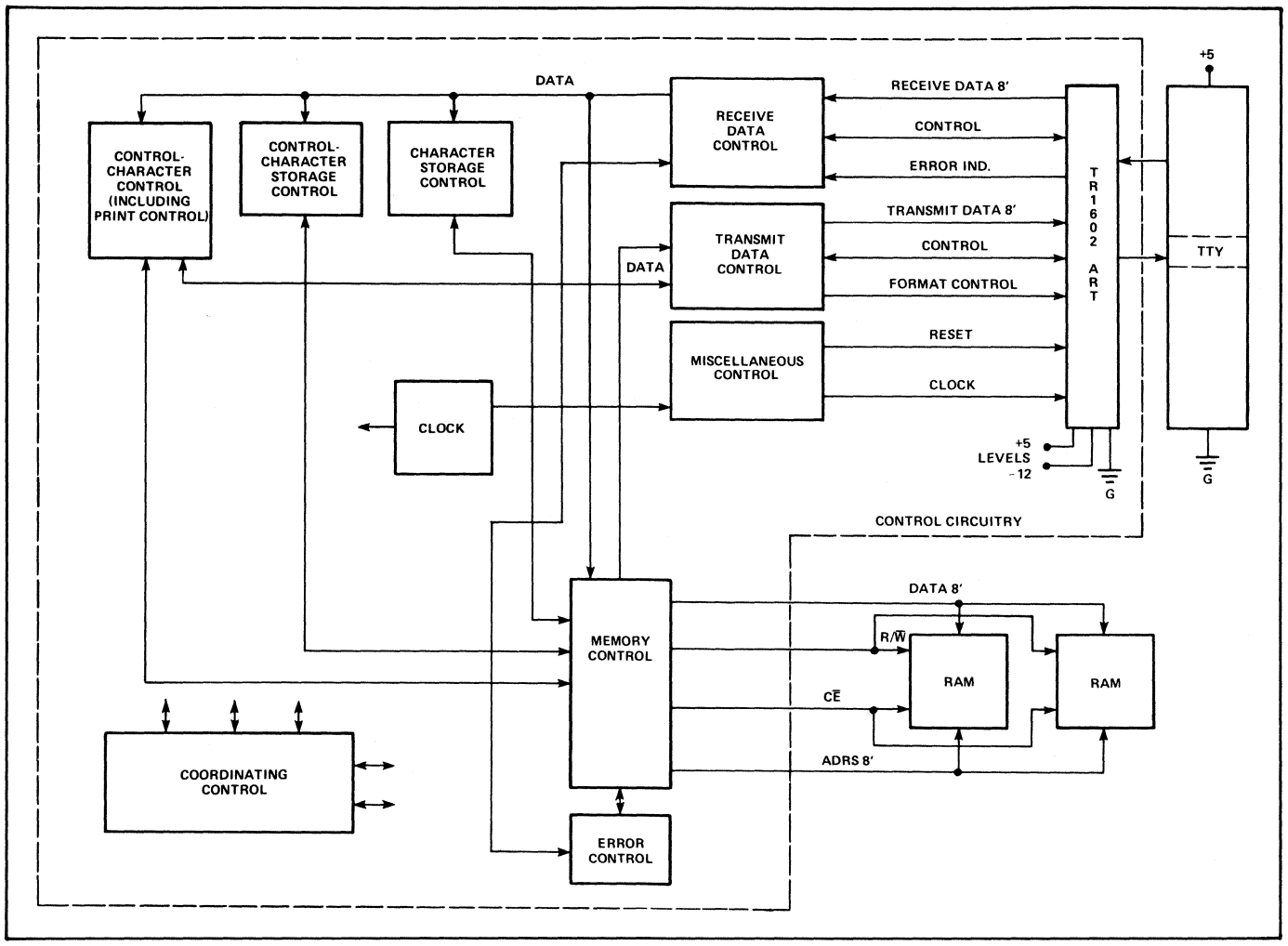


Figure 3.3 Block Diagram—Conventional Implementation for the Intelligent Typewriter System (ITS)

packages are required to implement this random logic version of the ITS.

3.3 Selection of a Microprocessor

The process of microprocessor selection involves a rather complex tradeoff between a number of key factors that include (1) the overhead electronics (e.g., input/output interface, clock, power supply), (2) CPU capability (e.g., functional speed, instruction, set, software development tools) and (3) availability (e.g., delivery schedule, second sourcing, cost/volume).

It is generally recognized that one particular microprocessor is not suitable for all possible applications. Thus, we will discuss this process of selection, placing particular attention to the application at hand, namely, ITS. Moreover, during this selection process, the designer should keep the overall system specification uppermost in his mind. In other words, the successive narrowing down of the list of available microprocessors must be dictated by performance assessment at a system level rather than any individual feature. In the interest of brevity, we will discuss the abovementioned factors and then suggest one suitable microprocessor.

In Figure 3.1, we have identified the portion of the overall system that is to be replaced by a microprocessor. The first factor to be assessed is the overhead electronics to incorporate the microprocessor.

All microprocessors require a clock which may be single or multiphase; from a timing generation viewpoint, a single phase clock is more desirable. Similarly, microprocessors that use a single power supply level—that is, TTL compatible—are more desirable. The two main types of data transfers in and out of a microprocessor are serial and parallel. The microprocessor hardware should be such as to allow the implementation of a serial or a parallel interface with little or no interface circuitry (e.g., latches, line drivers, multiplexers).

Having devised means of getting data into and out of microprocessor, we are in a position to consider the second factor, CPU capability. Since we are replacing hardware components by a software program within the microprocessor, it is essential that the instruction set of the microprocessor be sufficiently "powerful" to perform the job. The "power" of the instruction set is reflected by the available addressing modes for executing the data transfer, control transfer, arithmetic and logic instructions. Ultimately, for a particular program, this can be translated into usage of memory (RAM/ROM) and the speed with which the important functional blocks of the application can be implemented.

One important point to be kept in mind is that, provided the microprocessor can execute the required function fast enough so that the overall system meets the desired performance specifications, then it is uneconomical to incorporate a faster but more expensive microprocessor. Another facet of CPU capability is the ease with which the necessary software program can be developed. An essential software development tool is the assembler, discussed in Chapter II.

LOADER

Other software tools include means of (1) entering the program into the microcomputer memory (i.e., **loader**),

EDITOR

(2) deleting or inserting instructions into a program (i.e., **editor**), and

SIMULATOR

(3) duplicating the functioning of the program from EDITOR a software point of view (i.e., **simulator**).

Additional details regarding these development tools can be obtained from manufacturer microcomputer manuals (e.g., Signetics 2650 manual).

The third main factor in microprocessor selection is availability. To assure himself of delivery schedules, the system designer should ensure that there are multiple sources for a selected microprocessor. Moreover, the manufacturer must be capable of delivering in reasonably high volume at a competitive price.

As noted earlier, the selection of a microprocessor is a long and time consuming process. In the remaining part of this section, we will describe the Signetics 2650, which meets all of the above-mentioned requirements handsomely. The pinouts of the 2650 are functionally arranged in Figure 3.4; these are described in the following:

Power Supply The microprocessor operates on +5 VDC supply. In fact, **all** inputs/outputs for the 2650 are **TTL compatible.**

Clock A single phase clock (using normal TTL voltage swing) is employed which can run from DC to 1.25 MHz. (Note: the processor can be single-stepped for debugging.)

Reset Starts processing from a known state (location zero).

Flag Is output from a latch driven by one of the program status word (PSW) bits. Use is programmer's choice.

Sense Is input directly to another PSW bit. Use is programmer's choice.

Address 15 bit address bus for program, data memory and I/O.

Data 8 bit, bidirectional data bus for program, data memory and I/O.

M/$\overline{IO}$ Indicates whether operation is memory (M) or I/O. Used to gate read or write signals between memory or I/O devices. High state corresponds to memory operation; low state to I/O.

$\overline{R}$/W Determines direction of data bus in reading or writing. High state corresponds to write operation; low state read.

WRP Timing signal from the processor that provides a positive going pulse

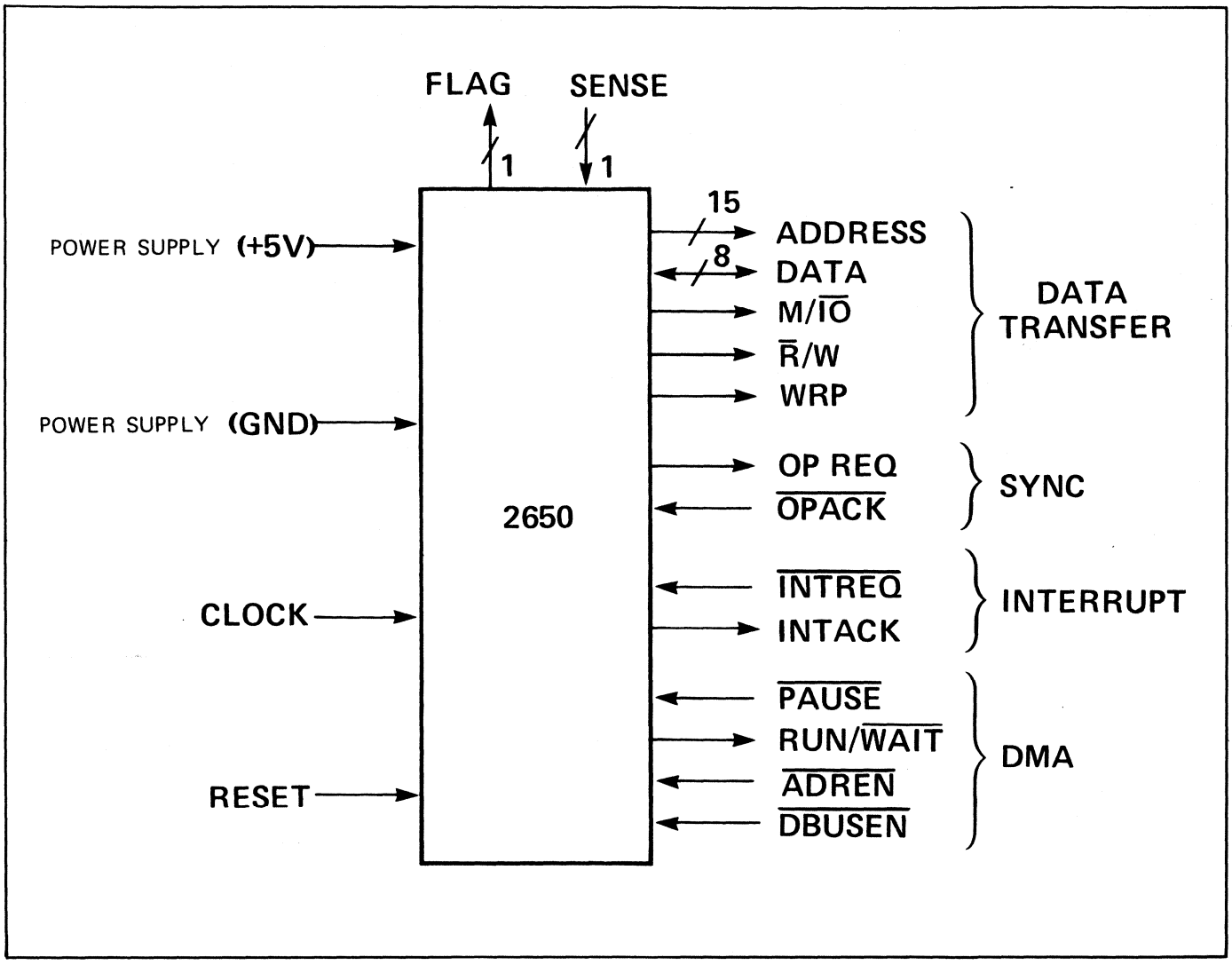


Figure 3.4 2650 Interface Signals

in the middle of each requested write operation (memory or I/O) and a high level during read operations. Designed for use with Signetics 2606 R/W memory circuits to provide a timed chip enable signal.

OP REQ Coordinating signal for all operations.

$\overline{\text{OPACK}}$ Response to OP REQ from external device.

$\overline{\text{INTREQ}}$ External interrupt.

INTACK Response to INTREQ from 2650.

$\overline{\text{PAUSE}}$ Request to temporarily stop operation of the 2650.

RUN/$\overline{\text{WAIT}}$ Indication of the operating or temporarily stopped state of the 2650.

$\overline{\text{ADREN}}$ Removes 2650 Address lines from the tri-state bus.

$\overline{\text{DBUSEN}}$ Removes the 2650 Data lines from the tri-state bus.

The above description of the pinouts is terse; for more details the reader is referred to Appendix A and the Signetics 2650 manuals.

Taking advantage of the available pins on the 2650, the system designer minimizes the external hardware circuitry to (1) interface the 2650 as simply as possible and (2) perform as many functions in software as possible. Thus, the task of the system designer is now oriented toward software program

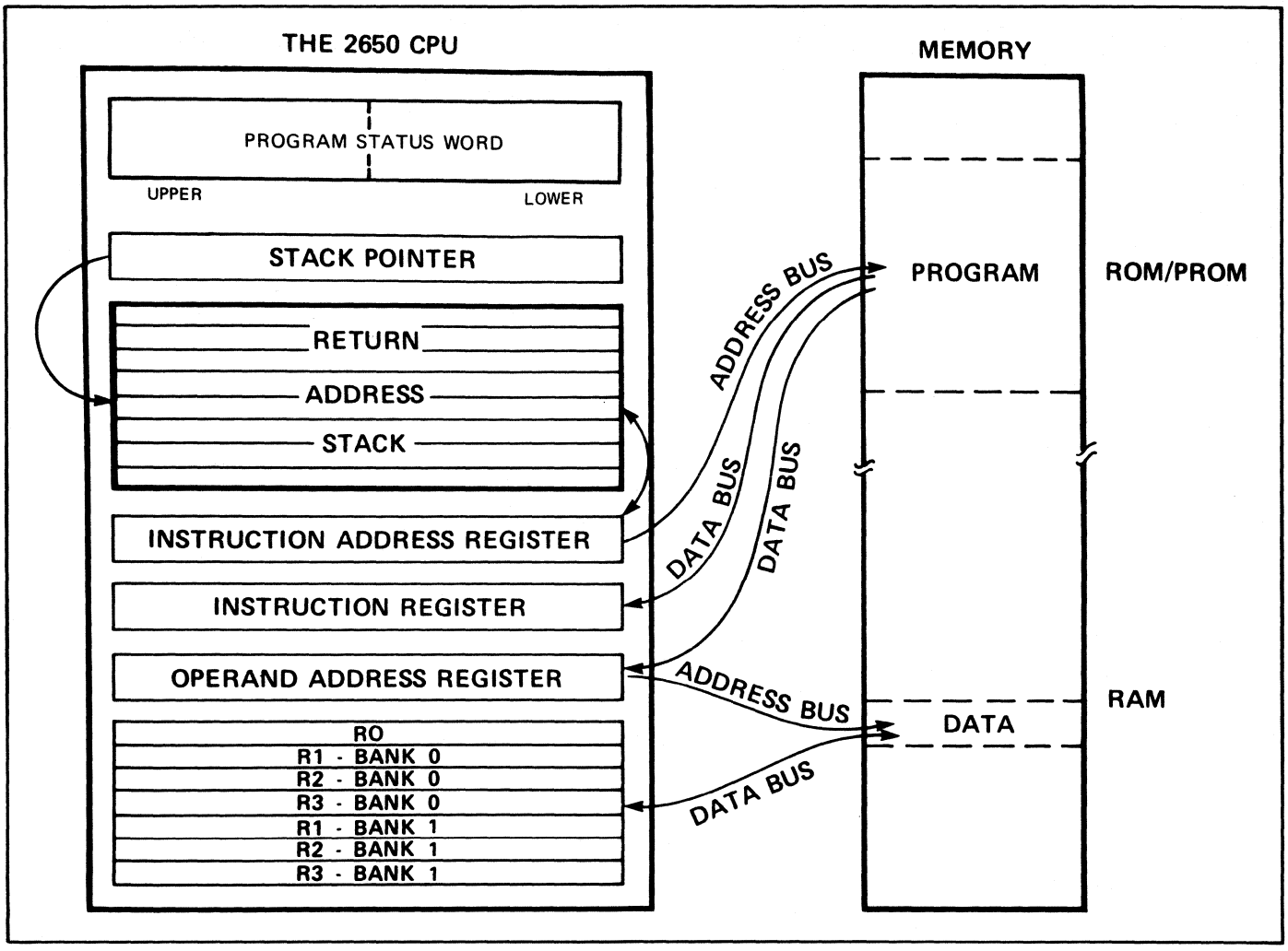


Figure 3.5 A Programmer's Conceptual Model

design, rather than conventional random logic design. To facilitate the programming task of the system designer, we present a conceptual model of the 2650 CPU in Figure 3.5. While designing, coding and debugging programs, the system designer should interpret the operation of the microprocessor during the execution of each instruction in terms of these registers. The reader is urged to go over the 2650 instruction set, documented in Appendix A, by seeing what happens to the contents of these registers after the execution of each instruction.

In the following sections, we will propose two designs for the ITS. We motivate the reader to consider these designs by summarizing the main software and hardware features of each design. As can be seen from Table 3.2 the last design proposed is significantly superior to the other two. We note that (1) the hardware electronics parts count is reduced by a factor of 10:1, (2) support components are significantly reduced, (3) prototype development is more methodical and, therefore, less expensive, and (4) production costs are significantly reduced. We will return to this table while considering the proposed designs.

3.4 Microprocessor-Based ITS Using a UART

By designing a general purpose serial I/O interface between the Signetics 2650 microprocessor and the teletype, we can transfer the burden of designing hardware control circuitry to implement the necessary functions, as in the random logic based design, to that of designing a software program within the microprocessor.

The basic design approach is to use a UART, as in the previous design of Section 3.2, to convert from

Table 3.2 Software/Hardware Comparison of the Designs

DESIGN	HARDWARE		PROTOTYPE DEVELOPMENT	PRODUCTION COST ESTIMATE (%)**
	IC PARTS COUNT	SUPPORT* COMPONENTS		
Conventional	75	Substantial	Significant Hardware Debugging	100%
Microprocessor Based (1)	18	Same	Some Hardware Debugging Software Debugging (350 bytes)***	40%
Microprocessor Based (2)	6	Negligible	Software Debugging (250 Bytes)***	10%

*Support components—PC board, connectors, cables, power supplies, cooling, packaging, etc.
**Quantities of 100 units; amortized development costs.
***Excludes comment cards.

serial teletype I/O to the more convenient parallel I/O. Then, the parallel input/output data bus of the microprocessor is connected to the parallel port of the UART. The additional control circuitry required to accomplish this is presented in Figure 3.6. The signals lines on the left hand side of the page are the Signetics 2650 pins. The number of IC packages to implement this version is 18, and the length of the software program is less than 350 bytes.

The flow chart for this program is documented in

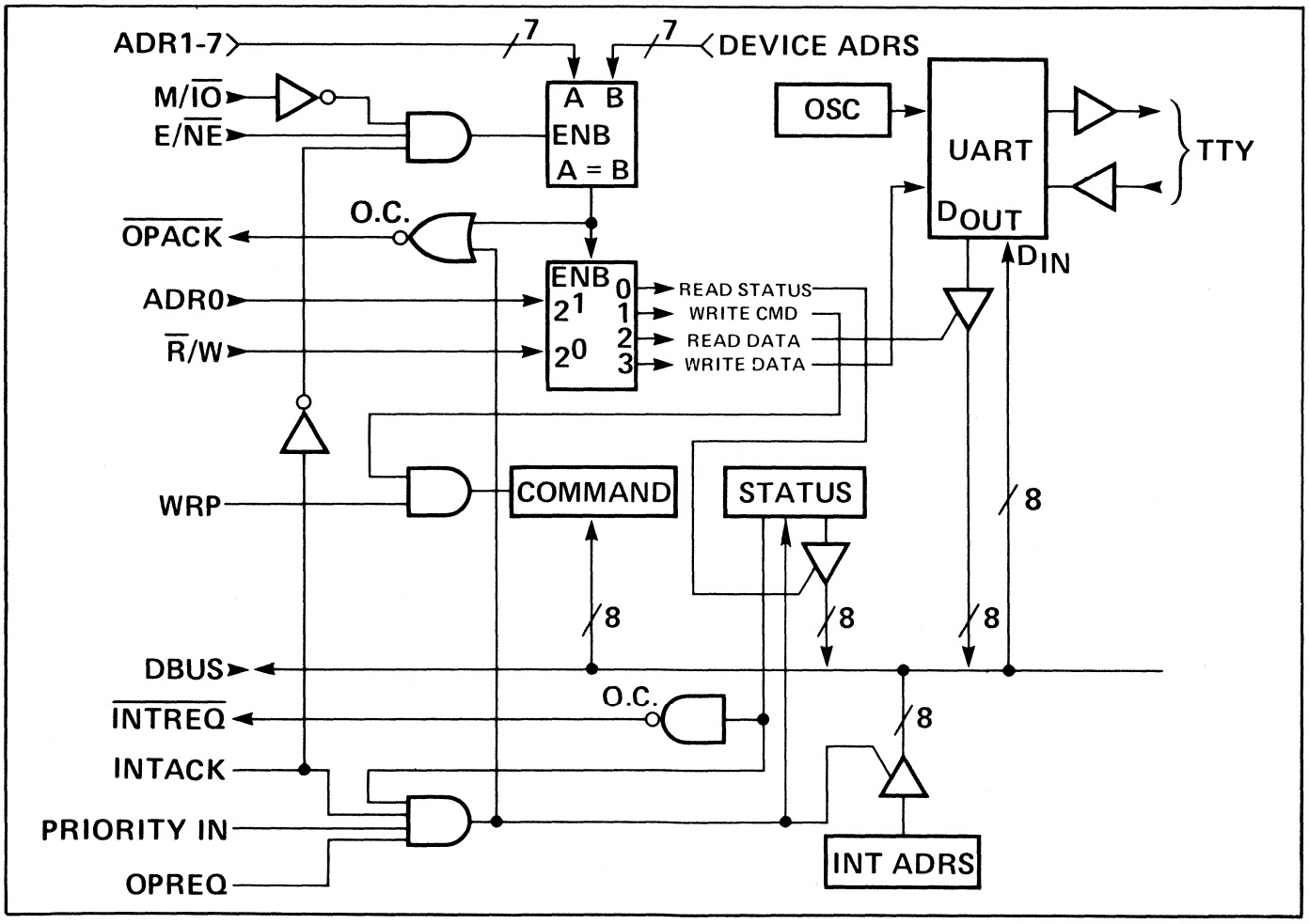


Figure 3.6 General Purpose Serial I/O Interface (ITS)

Appendix B and the corresponding assembly language program listing is presented in Appendix C. The main ITS software program flow chart is depicted in Figure 3.7, describing the process of text insertion, including the main subroutines. Referring to Figure 3.7, we begin by initializing the ITS in the subroutine labeled INIT; this entails clearing the typewriter control mechanism, the keyboard buffers and the memory in which the text is stored. Then, subroutine "IN" gets a character from the keyboard buffer. Since the hardware interface is parallel, the 7-bit character pattern is

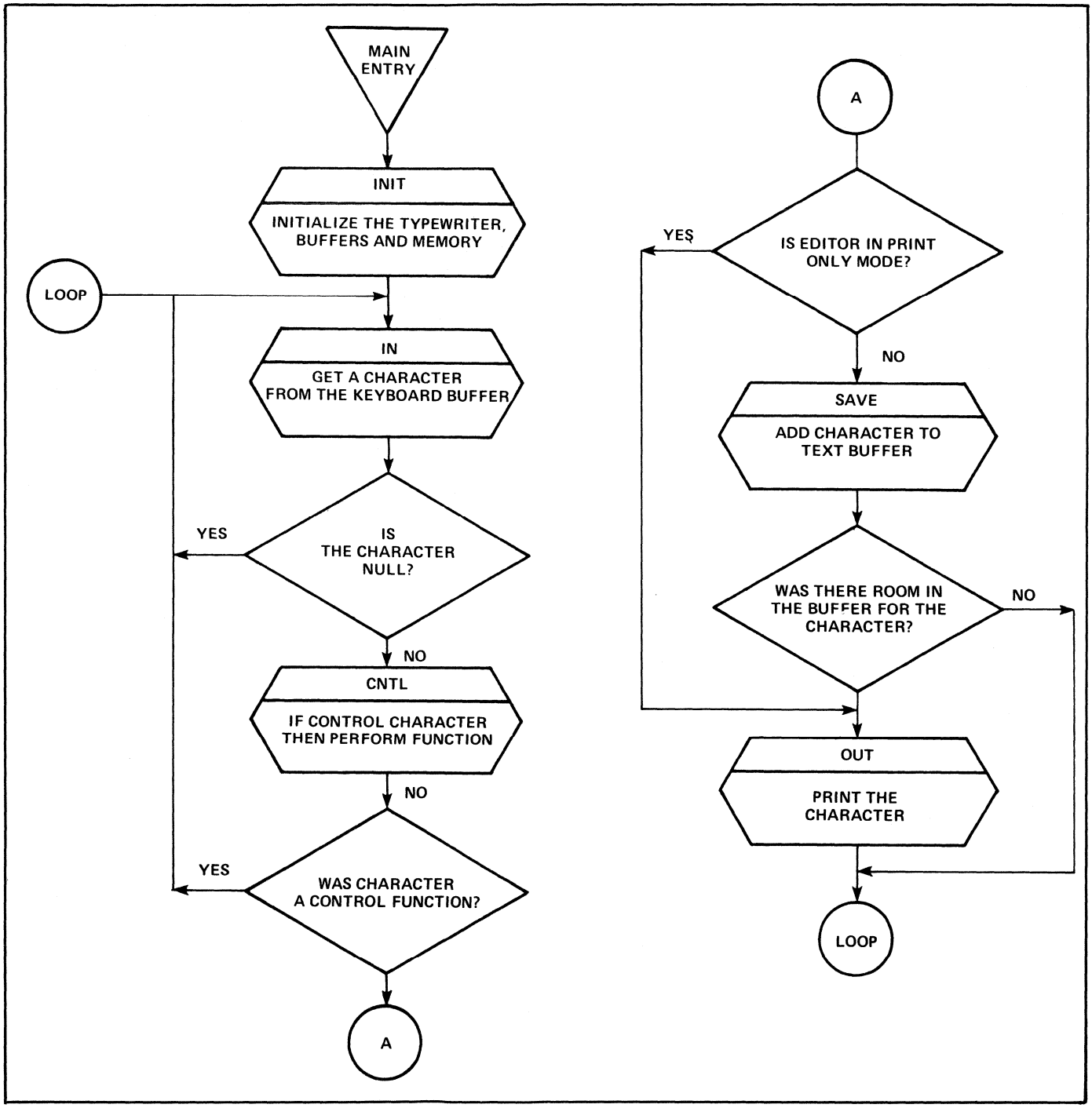


Figure 3.7 ITS Program Flowchart

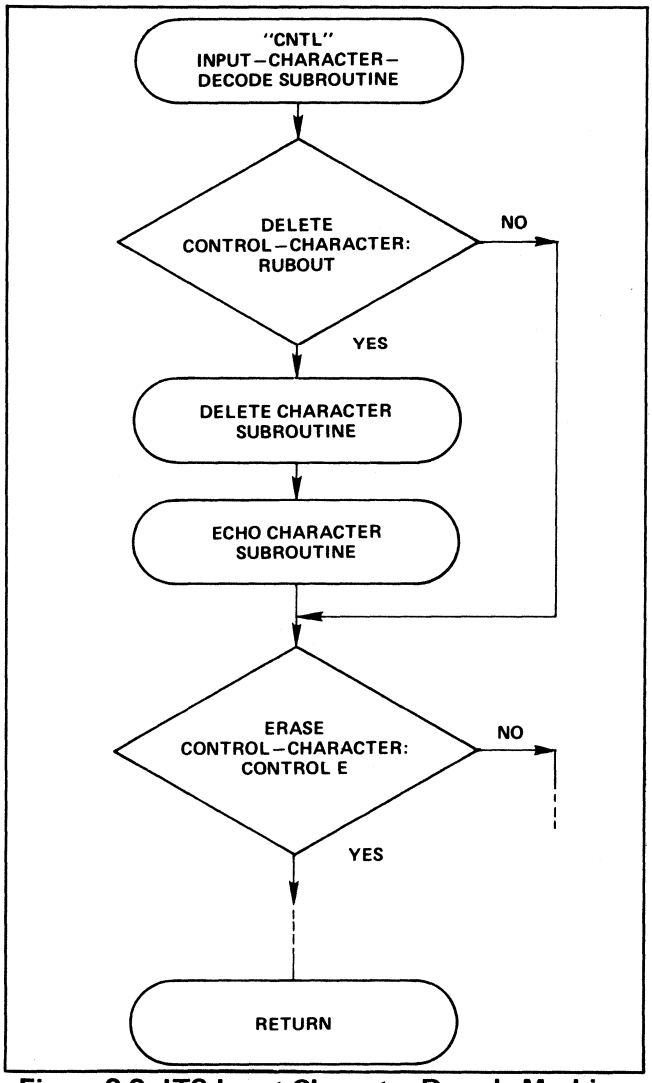


Figure 3.8 ITS Input Character Decode Machine (CNTL) Flowchart

transferred to the 2650 by a simple five instruction routine. We will discuss this routine at length in this section. (See Figure 3.10.) Note that the line from the teletype input is high (+5V or a logical 1) when no character is being transmitted, as in Figure 3.2. In this hardware configuration, the UART handles the task of determining whether or not a character is being sent. In the next section, we will propose a configuration where this function is performed by software. (See Figures 3.11 and 3.13.)

The next operation in the basic ITS flow chart (see Figure 3.7) depicted by subroutine "CNTL" is the determination of the type of character just received:

1. Character for memory storage.
2. TTY control character for memory storage.
3. Control character for text control purposes.

The sequence of operations that takes place within this routine is further expanded in Figure 3.8. The character just received is compared by the 2650 against known values of control characters. If a match is found, like the RUBOUT control character (Figure 3.9), from the TTY, the control function is executed. In this example, the RUBOUT character causes delete of the last character in memory. The delete-character subroutine is called by an instruction to execute the delete task. Next, the deleted character is "echoed" to the TTY so the user can verify what he deleted.

Proceeding to the next level of detail, let's look at what happens inside the delete character routine, documented in Figure 3.9. Referring to this figure, we note that the main operation in this routine is the replacement of the given character with a null character. In the 2650, a null character is NULL represented by an eight-bit byte containing all zeros. This byte is readily generated by the logical function instruction called "EXCLUSIVE OR" discussed in Chapter II. All we have to do is "EXCLUSIVE OR" the contents of R_0 with itself. This is accomplished by the instruction:

EXCLUSIVE OR, R_0

Note that it is implicit in this instruction that the other register to be EXCLUSIVE OR'ed is R_0. We will consider a version of the echo character subroutine in the next section.

In the foregoing discussion, we began with the main ITS program of Figure 3.7. Then, we looked at the

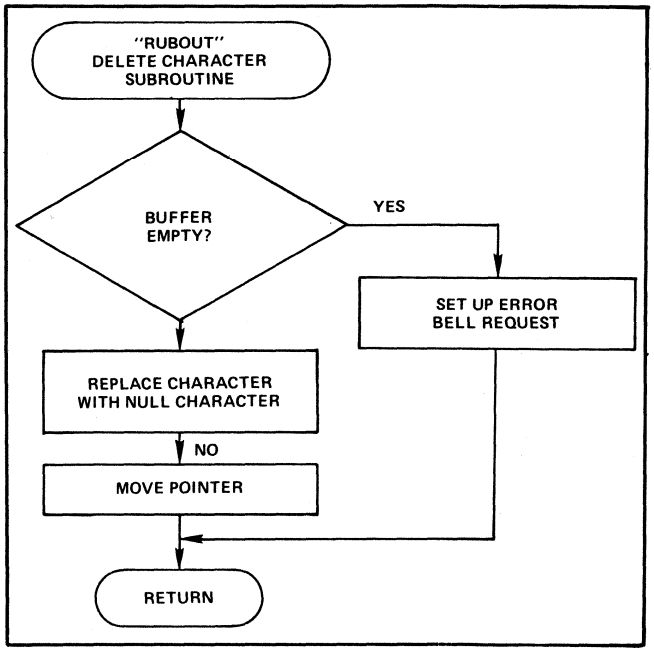


Figure 3.9 Delete Character Subroutine Flowchart

flow chart of a specific routine "CNTL" in Figure 3.8. Subsequently, we looked at a specific routine "RUBOUT" in Figure 3.9 that was called in "CNTL" of Figure 3.8.

Finally, we elaborated on the way in which the 2650 generated the null character by the EXCLUSIVE OR operation discussed in Chapter II.

TOP DOWN DESIGN

This process of sequentially proceeding to the next level of detail until the task to be performed can be described by the microprocessor instructions themselves is called **top-down design.**

Starting with a system specification, the job of the microprocessor-based system designer is to plan the functioning of the entire system by this logical top-down programming process. Thus, the emphasis in developing a good design in a timely manner is to design well-structured, easy to debug/modify/understand programs. Additional details regarding this are presented in Appendix B.

Going back to Figure 3.7, we see that the next task, after performing the functions in routine "CNTL" is to check the editor status. If the editor is not in the print mode, then it implies that we are inputting the text; consequently, we add a character to the text buffer memory in routine "SAVE." Of course, if the character was a control operation as described in the last paragraphs, it is not stored in memory. But, if it is one of the following, it is stored in memory:

1. Character for Memory Storage
2. TTY Control Character for Memory Storage (like typewriter carriage, return, line feed, or advance paper and stop)

After ensuring that there was, indeed, room left to store this new character, we send the character back to the teletype printing mechanism (ECHO), so that the user can verify what he typed in. This whole process is repeated in an endless loop until an appropriate command is decoded to indicate the completion of the text insertion task.

Going back to Figure 3.7, let us see what the sequence of operations must be, to get a character from the keyboard buffer into one of the 2650 registers, say R1. The flow chart of this routine is shown in Figure 3.10. Since we have a parallel I/O channel (see Figure 3.6), we can directly load the character from the keyboard buffer into R1.

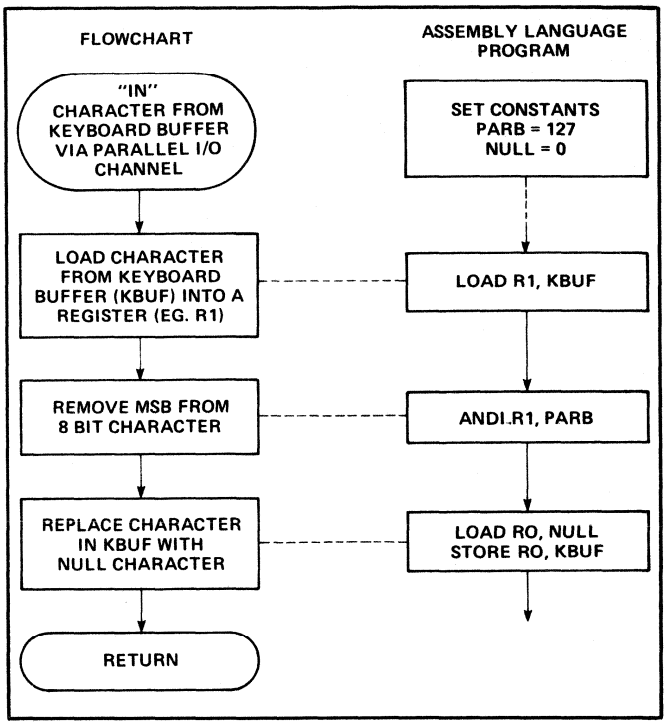


Figure 3.10 Input Character Routine Flowchart and Corresponding Assembly Language Instructions for a Parallel I/O Channel

Then, we replace the most significant bit (MSB) of the contents of R1 by zero to get the 7 bit character code. Finally, the contents of the keyboard buffer are zeroed out by loading in the prestored null character. On the right hand side of the flow chart, Figure 3.10 shows the corresponding assembly language instructions to implement this program. The operation of these LOAD, STORE, and AND instructions was described in Chapter II. The reader is invited to refer to Appendix A for the complete set of 2650 instructions. The listing of this particular program in 2650 assembly language can be found in Appendix C for the subroutine 'KEYIN,' whose listing begins on page 4, line 173.

We will return to the flow chart in Figure 3.10 to compare it to that required when the I/O channel is serial, as in the design proposed in the next section.

As noted in Table 3.2, the implementation of this microprocessor-based ITS required 18 IC packages containing a program that is less than 350 bytes long; a listing of this program is given in Appendix C. We are now in a position to discuss the main reasons why microcomputers have a significant advantage over random logic:

1. Reduces system complexity
2. Ease of development
3. Flexibility (ease of system function modification)
4. Reliability
5. Ease of support
6. Cost

Comparing the two implementations, we can see that the system complexity is significantly reduced; this will be even more evident in the design proposed in the next section. Since the hardware complexity is reduced in terms of parts count, it is much easier to lay out the printed circuit boards; cross talk, and other interference problems are reduced; connections, cabling cooling and packaging requirements are reduced. Most significantly, the 2650 requires only one +5V supply.

Other reasons for the ease of development are that a software program is often much easier to understand than an equally complex piece of hardware. Debugging software is much more systematic and, therefore, often less time consuming than hardware troubleshooting. For example, problems such as electronic circuit malfunction, interfacing, timing pulse alignment, radio frequency interference are practically eliminated. Debugging the 2650 is particularly easy because its internal circuitry is static rather than dynamic; consequently, the clock can be stopped to look at its pins without losing data or status. The microcomputer-based system is more flexible and easier to support because of the fact that software can be readily modified and is readily documentable. Reliability is greatly enhanced, again due to reduced parts count.

All the above factors can finally be translated into cost savings to the manufacturer. Software development is a one-time cost that can be spread across the production run. Field support is easier with fewer spares required in stock. Finally, the product can be continually upgraded without altering the hardware packaging leading to market competitiveness in terms of the introduction of newer products. Additional comments related to the cost-effectiveness of microprocessor-based solutions to electronic system design problems are made in the next section.

3.5 Microprocessor-Based ITS Using Serial I/O

We noted in Section 3.1 that the teletype was a serial I/O device. In the microprocessor-based design of Section 3.4, it was necessary to use a UART to convert the serial I/O teletype channel to a parallel channel so that the characters could be input to the 2650 via the parallel data bus shown in Figure 3.4. But, for an application involving a relatively low speed device such as a teletype, there is no real need to use the high speed parallel data transfer paths of the 2650.

PSU

Recall that the "sense" bit in the 16-bit program status word (PSW) is located in the most significant bit location, i.e., bit 7 of the upper half of the PSW designated as **PSU**; and bit 6 is the flag bit in the PSU.

Referring to Figure 3.4, these bits are directly accessible on the 2650 pins. These two pins (the sense and flag pins) can be used to implement a serial I/O channel in the following manner.

For inputting TTL compatible serial input data, we can use the sense line. The sense bit is normally a 1 (+5V) between data transfers (see Figure 3.2). The line drops to zero volts (0) to indicate a start bit. Then 8 bits are serially transferred. After this, the line goes back to a 1 (+5V) for one or two stop times, depending on the data transfer rate.

STORE PSU

This line can be sampled inside the 2650, under software control, by executing a **STORE PSU** instruction which stores the contents of the PSU into R_0 and sets the condition code bit (CC) of the PSW. For outputting TTL compatible serial data, we can use the flag line.

STOP BIT

To transmit a start bit back to the teletype, we set the flag bit of the PSU to a 0; to transmit a **stop bit**, we set the flag to a 1.

SET PSU

Moreover, to transmit data bits, the flag bit is set the same as the corresponding data bit. This process is accomplished under software control by executing the **SET PSU** instruction.

Thus, we realize that, in the case of this dedicated microprocessor application (namely ITS) there is really no need for the generalized serial I/O interface proposed in Figure 3.6. Instead, we can directly use the sense/flag pins on the 2650 for serial I/O. The resulting hardware configuration for this dedicated ITS application is shown in Figure 3.11.

Three control signals from the Signetics 2650 control the ITS memory, not including the address bus:

OPREQ

OPREQ is a coordinating signal signaling that an external operation is taking place.

$\overline{\text{OPACK}}$

$\overline{\text{OPACK}}$ is grounded and unused since the 2606 and 2608 respond in less than 1 μsec to a 2650 request.

$\overline{\text{R/W}}$

$\overline{\text{R}}$/W selects a read or write operation on the 2606 RAM memory, and WRP provides a timing pulse for the same.

ADR10

The 10th address bit, **ADR10**, acts as a chip select. It places the 2608 in address space 0 to 1023, and the 2606 is in the address space 1024 to 2047.

And ADR0-ADR9 select one location in those address spaces. Notice that we have a total of 6 IC packages and only one +5V supply drawing about 500 milliamps! The hardware for this system is available from Signetics on a 2-inch by 3-inch printed circuit card!!

Now let us look at the software program. Functionally the software program becomes simpler! We no longer have to generate the UART control signals. The only significantly new software program is one that converts the serial input from

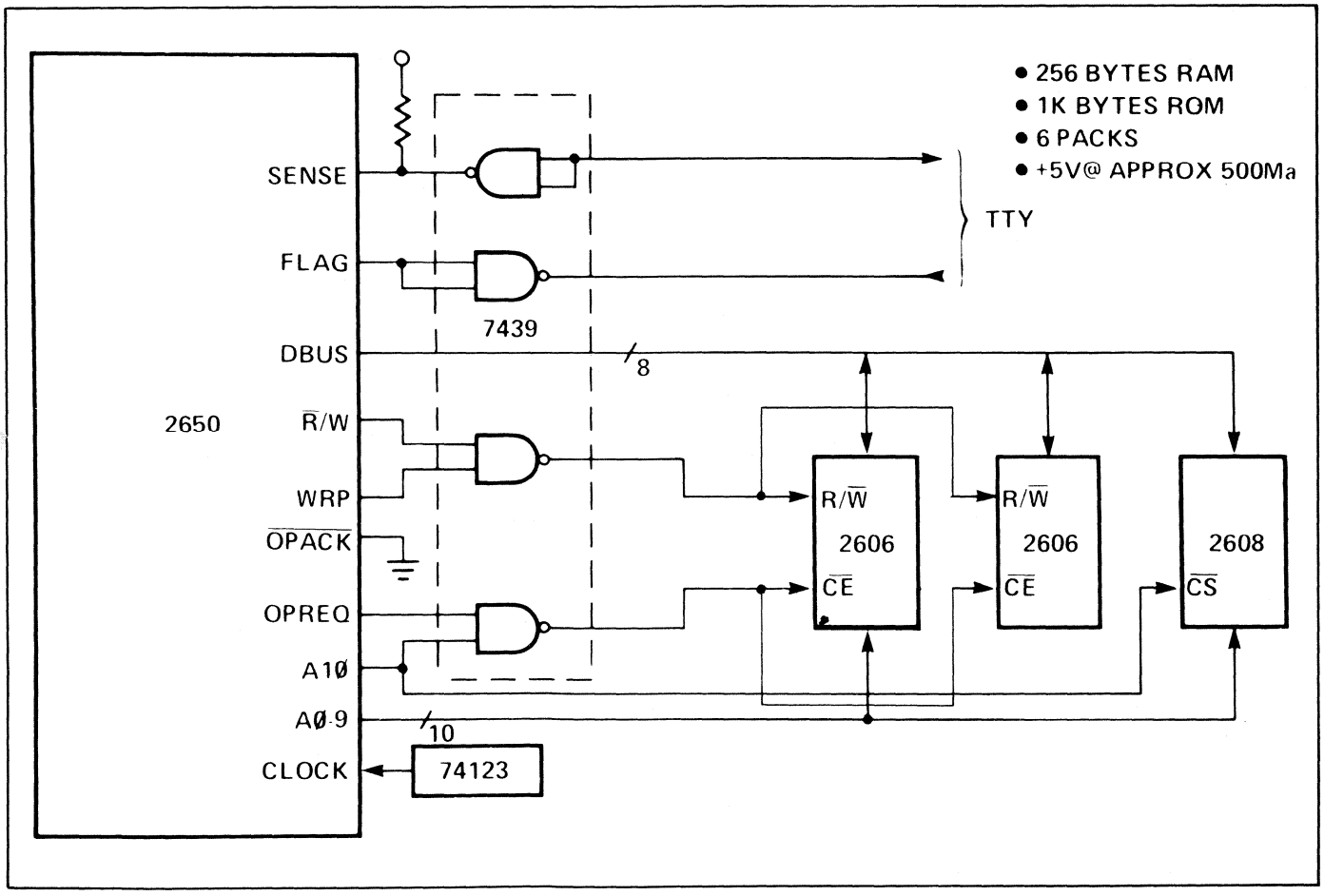


Figure 3.11 Optimum 2650 Solution

the sense line to parallel byte format for further processing and the logic required to set the flag line to echo or print the proper character on the teletype printer. We will look at this program in more detail in the following.

Referring to Figure 3.7, we note that keyboard processing is done in subroutine "IN." Let us discuss the detailed flow chart and the corresponding program for this subroutine, using the simplified instruction set developed in Chapters II and III. The flow chart for this conversion is shown in Figure 3.12. The first job is to continually sample the sense line until a start bit is detected. Then, we introduce a delay of half the bit time to test the sense line again to ensure that it was not a noise spike. After ensuring that it was indeed the start bit, we then introduce a delay of one bit time to test the sense line for the first bit of the 7-bit character. This process is repeated until all 7 bits are received and put into the proper parallel byte format.

The delay for any timed operation is a simple matter in the Signetics 2650 Microprocessor. Any register, like R_0, is loaded with a number.

LOOP

The register is decremented by one each time through a program loop (a **loop** is a sequence of instructions which transfers execution from finish to start and is usually executed more than once).

When the register is tested (each time through the loop) and found equal to zero, the timed delay is complete. The timing is provided by three things:

1. 2650 input clock frequency 1 MHz in the case of the ITS. The 2650 clock frequency is variable up to a maximum of 1.25 MHz.
2. Instruction execution time. The time to execute an instruction is a fixed value which depends on the type of instruction and the clock frequency. The total of the execution times of every instruction in the loop gives the loop delay time.
3. Number loaded into the register being used in the program loop. This is the number of times the loop is executed, and, therefore, the number of loop delay times.

Example

Clock frequency	= 1.25 MHz
Loop contains instruction A, B, and C	
Instruction execution times	A = 4.8 μsec
	B = 4.8 μsec
	C = 7.2 μsec
Loop execution time	= 16.8 μsec
Number of times through loop	= 100
Total delay time	= 1.680 msec

Once a valid start bit has been detected, a delay of one bit time (~ 9.1 msec) is made until the middle of the first data bit. The middle of the first data bit was reached in the following manner: the leading edge of the Start bit was detected because the 2650 program was continuously looking for it in a tight loop. The program loop is very fast compared to the frequency of the sense signal (several microseconds compared to 9.1 milliseconds); so when the start bit was detected, it can be assumed the leading edge was detected and not the middle. The middle of the start bit was located due to the ½ bit time delay during the noise check. Finally, the middle of the first data bit was detected due to the one bit time delay from the middle of the start bit.

The first data bit is sampled on the sense line as "1" or "0" (high or low) and saved. When 7 bits have been received in this manner (a count is kept in R_2), an entire character has been received.

The serial to parallel conversion for each character is accomplished by transferring a data bit from the sense bit into R_0 with the "STORE PSU" instruction. The data bit alone is left in R_0 after execution of the "AND" instruction. The last data bit sampled is assembled together with the data bits previously received in R_0 by the "INCLUSIVE OR" instruction. The "STORE" instruction puts the contents of R_0 into R_1. Finally, the "ROTATE RIGHT" instruction gets the contents of R_1 ready for the next bit of the character.

The way in which the verbal flow charts of Figure 3.10 and 3.12 can be implemented using the set of basic instructions developed in Chapter II and III (e.g., see Table 2.11) has been shown. The interested reader is encouraged to follow through this translation process, from the flow chart to the assembly language instruction program. We call the reader's attention to the increased complexity of the flow chart in Figures 3.12 and 3.13 to that described in Figure 3.10. But this is a small one-time software development cost leading to a signi-

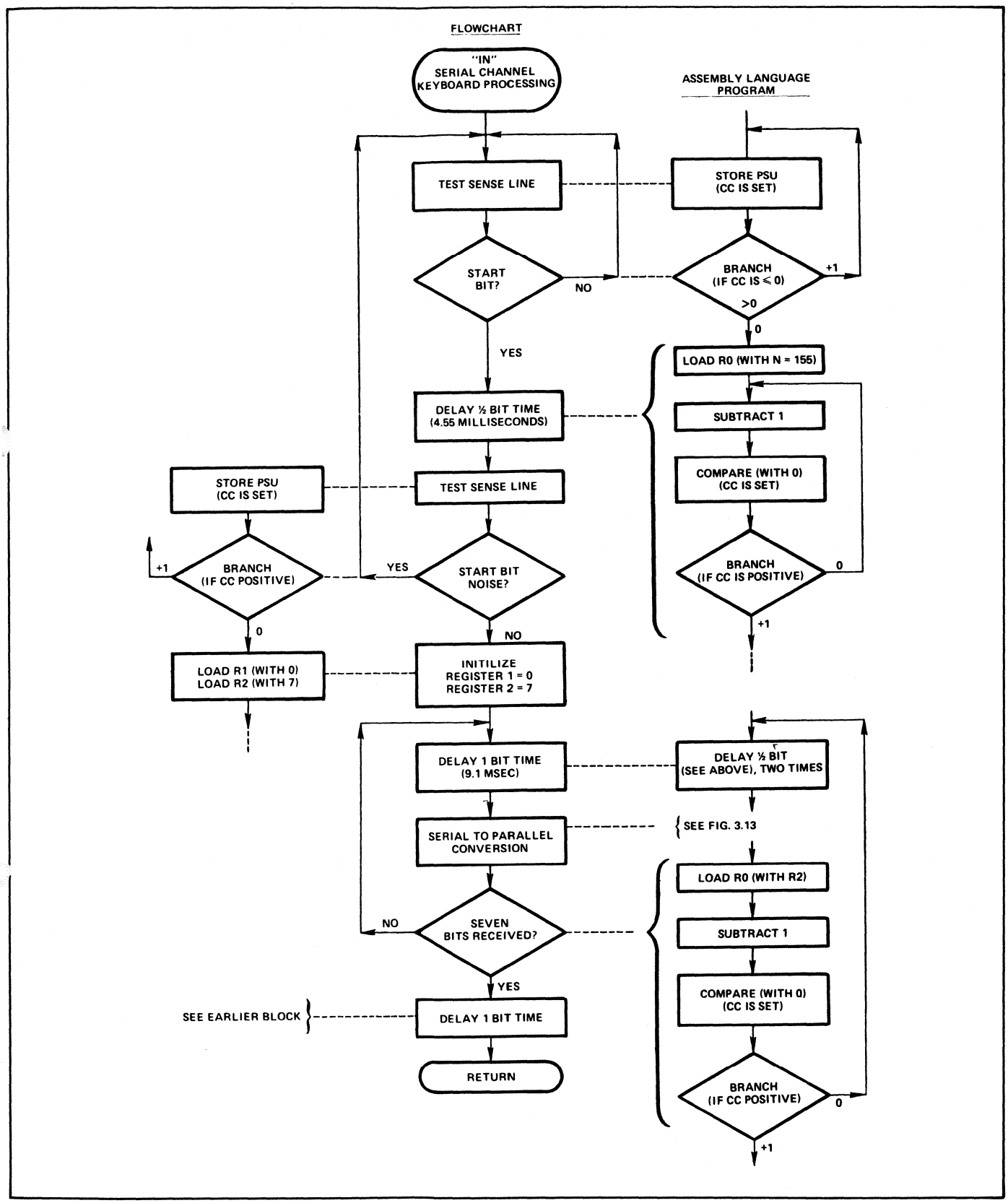


Figure 3.12 Input Character Routine Flowchart and Corresponding Assembly Language Instructions for a Serial I/O Channel

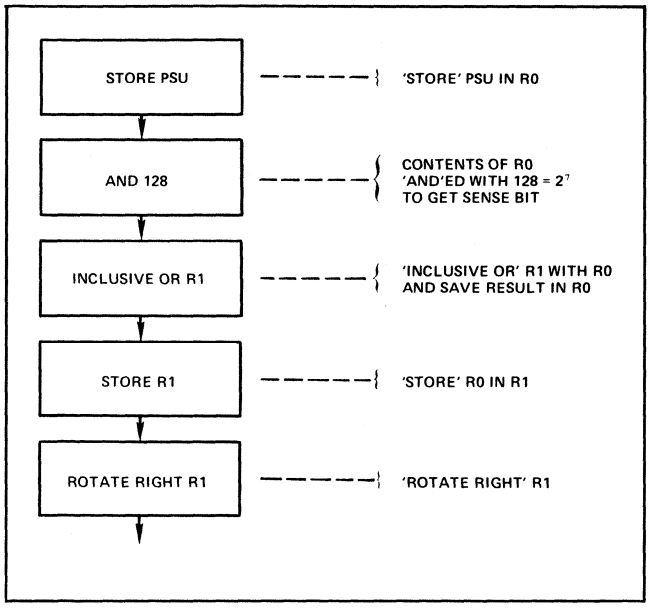


Figure 3.13 Serial to Parallel Conversion Flowchart

ficant reduction in hardware complexity and cost as can be seen by comparing Figure 3.6 and 3.11.

The "SET PSU" instruction is used to transmit data back to the teletype; this is done either to "ECHO" the character that has just been received from the teletype or it is done when characters are being read out from the text buffer, on command from the user. A preliminary flow chart for the echo character subroutine is presented in Figure 3.14. The reader is urged to translate this into a set of instructions.

The comparison between the three designs proposed was presented earlier in Table 3.2. We note that because of the specialized configuration used in the last design solution, the resulting software program is shorter than the parallel I/O interface microprocessor solution by almost thirty percent. We can, therefore, conclude that the more successful system designers will (1) attempt to perform as many functions in software as possible and (2) design well-structured software programs, to achieve cost-effective solutions. Technical details pertinent to developing well-structured programs are documented in Appendix B.

3.6 Other Features of the Signetics 2650

There are several other features of the Signetics 2650 that are important in various applications. We will note a few of them in the following. The advanced reader is referred to the 2650 Signetics manuals for information on features like DMA and vectored interrupt. A few of the features are reviewed below.

The seven general purpose registers are such a feature. This is a relatively large number of registers on-board the chip. This feature gives more flexible data manipulation capability including storage of intermediate variables during an involved computation. Another important feature is that the on-board chip RAS is 8 levels deep. Eight levels is generally enough to handle nesting subroutines, and having the RAS on-board makes access faster.

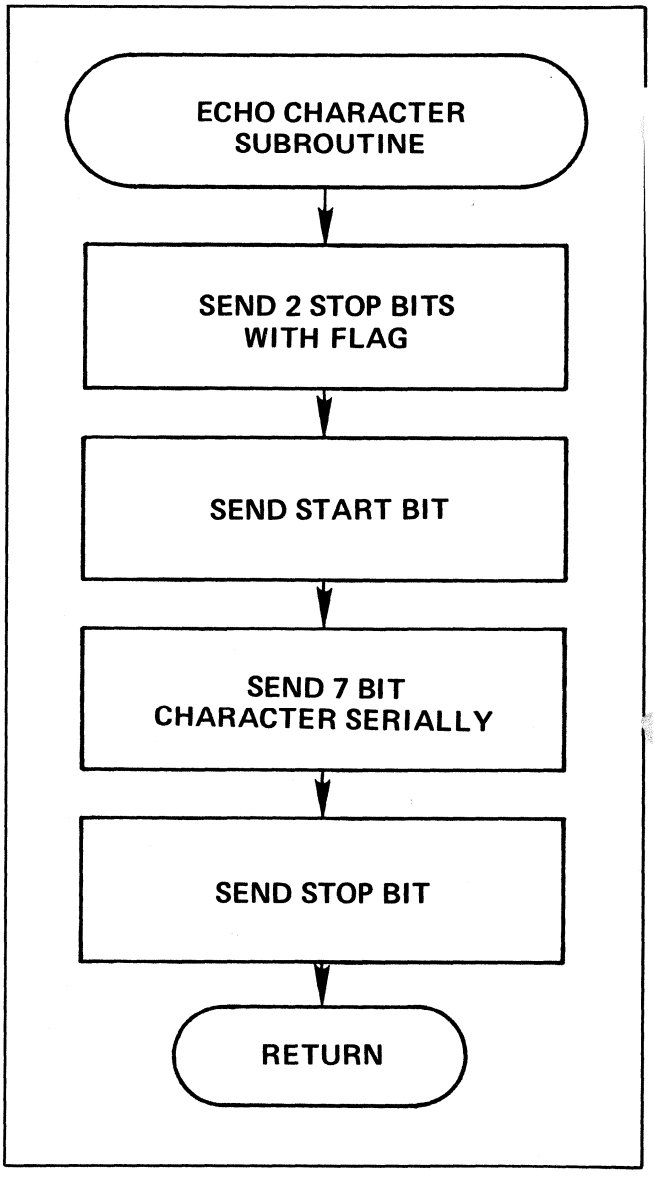


Figure 3.14 Echo Character Subroutine Flowchart

Another feature of importance is the instruction set. The 2650 has a powerful instruction set with 8 addressing modes. The modes are:

1. Register
2. Immediate
3. Relative
4. Relative Indirect
5. Absolute
6. Absolute Indirect
7. Absolute Indexed
8. Absolute Indirect Indexed

This is a very large number of addressing modes and it gives program flexibility.

Program flexibility allows a program to be written with fewer instructions. This means a savings in memory space and faster program execution. Faster program execution means that the Signetics 2650 can handle more tasks before increased computing power is required.

Microprocessors operate in two possible modes in a system, polled or interrupt, to service a number of external devices.

POLLING

For relatively slow processes or ones that can wait to be serviced, the microprocessor sequentially scans the internal devices; this is called **polling.**

When it finds a device that needs servicing, it performs the required function and the processor goes back to sensing.

INTERRUPT

In the **interrupt** mode of operation, generally used for real-time processing applications, the microprocessor is interrupted to do something special before executing the next instruction.

The microprocessor saves the contents of the program counter and branches to the interrupt routine to service the interrupting device, and upon completion, returns to the original program step.

VECTORED INTERRUPT

The Signetics 2650 processor incorporates the fastest mode of interrupt operation, namely, **vectored interrupt.** With this feature, the 2650 not only recognizes which of the many devices is requesting service but the interrupt also causes a direct branch to the servicing routine. An example of this is an anticipated power failure.

The contents of the critical registers would need to be preserved in non-volatile storage. A vectored interrupt would branch immediately to the program required to do this.

Another useful feature in microprocessors is the ability to access memory from an external device without having to pass the data through the 2650.

DMA

This process of direct memory access (**DMA**) to read or write blocks of data directly into memory without disturbing the processor, is particularly useful in real-time processing applications.

The Signetics 2650 allows DMA to be performed in three possible ways; the particular choice depends on the size of the data block to be transferred in or out of memory and the rate at which this transfer is to be accomplished. The advanced reader is referred to the Signetics 2650 manuals for a detailed exposition of these and other input/output capabilities of the processor.

In conclusion, the Signetics 2650 Microprocessor has a number of features (see Figure 3.15) which make it both easy to use and powerful. Features like TTL compatible I/O, single +5V power, static operation, and single phase clock make for ease of use. Features like seven general purpose registers, RAS on chip, vectored interrupt, serial I/O on chip, and 8 addressing modes make for processing power; processing power means less external logic which translates into less cost for every unit of a microprocessor-based system produced. The Signetics 2650 is a general purpose microprocessor which is applicable to a wide range of applications.

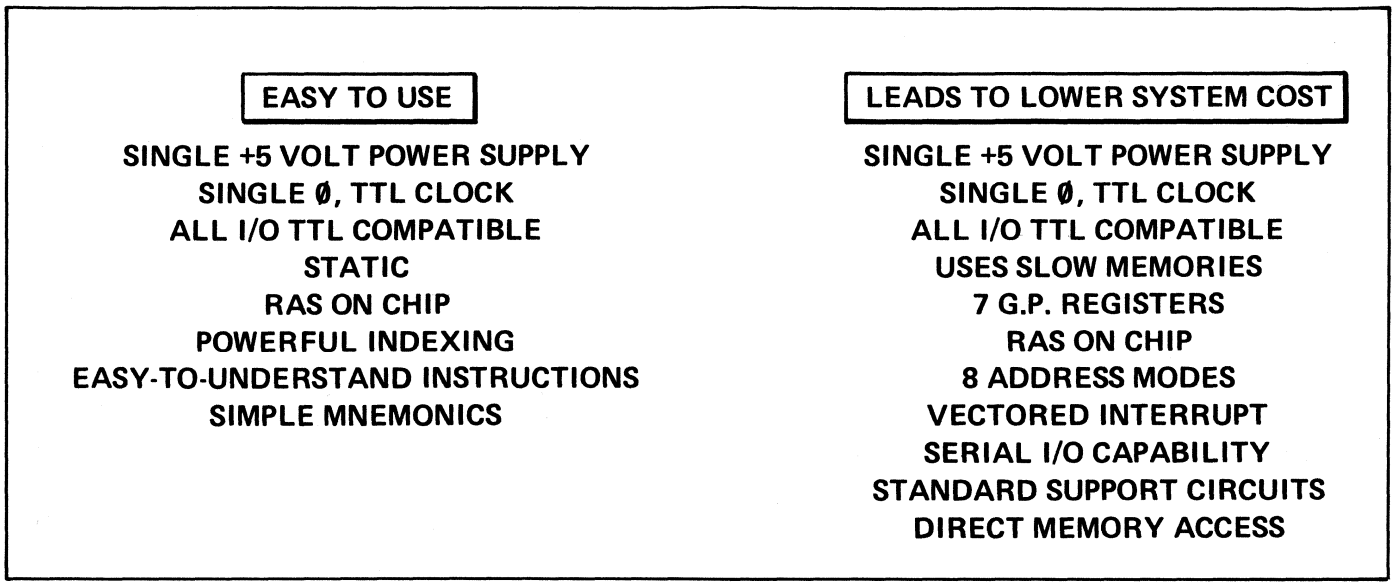


Figure 3.15 Signetics 2650 Features

QUIZ FOR CHAPTER III, INTELLIGENT TYPEWRITER SYSTEM

1. What is a use of stopping the clock in a system?

2. How does a microprocessor affect system hardware count when applied to a random logic system?

3. What is the meaning of serial/parallel conversion?

4. Why are programs written in assembly language?

5. What tool creates object code?

6. What is the use of a flow chart?

7. What can perform serial/parallel conversion besides hardware?

8. What is the relationship between programming costs and hardware costs?

9. What is a polled interrupt?

10. What are the two characteristics of polled interrupt operation?

11. What is a vectored interrupt?

12. Why use a vectored interrupt?

13. Why are general purpose registers on-board the microprocessor important?

14. Why use microprocessors?

15. What are the two main factors in microprocessor selection?

16. What are the two main objectives of the successful system designer?

ANSWERS TO QUIZ ON CHAPTER III

1. System debug; freeze system action to observe.
2. Reduces hardware count.
3. Converting a serial (one-at-a-time) bit stream into parallel form (bytes).
4. Assembly language is easily remembered and is a powerful tool in addressing and debugging.
5. Assembler.
6. In program development, to organize a program.
7. A program.
8. Programming cost occurs once. Hardware cost is multiplied by every unit produced.
9. A microcomputer polling subroutine scans I/O devices to determine the source of an interrupt.
10. Slower polling subroutine takes memory space.
11. The interrupting I/O device identifies itself to the microcomputer causing a branch to the proper service routine address.
12. Speed eliminates need for storage of polling routine.
13. Storage of intermediate results of a computation leading to higher processing speeds.
14. Cost reduction.
15. Overhead electronics and CPU capability.
16. Perform as many system functions in software as possible and design well-structured software programs.

BIBLIOGRAPHY

1. Altman, L., "Single-Chip Microprocessors Open Up a New World of Applications," *Electronics*, Vol. 47, No. 8 (April 17, 1974), pp. 81-87.
2. Andreiev, N., "Temperature Controllers Prepare to Tack as Microprocessor Wind Picks Up Strength," *Control Engineering* (October 1975), pp. 47-49.
3. Bailey, S. J., "Minicomputer Control: New LSI Chips Make it Practical," *Control Engineering* (December 1975), pp. 28-32.
4. ————, "Microprocessor: Candidate for Distributed Computing Control," *Control Engineering*, Vol. 32, No. 3 (March 1974), pp. 40-44.
5. Boehme, C. R., "Use of a Microprocessor as a Peripheral System Controller," *8th IEEE Computer Society International Conference Digest of Papers* (1974), "Computer Peripherals—Benefactor or Bottleneck?", pp. 181-184.
6. Brody, M. D., "Applying Microcomputers in the Laboratory," *Research/Development* (November 1975), pp. 28-36.
7. Cropper, L. C. and J. Whiting, "Evaluation of the Use of Microprocessors in CRT Display Terminals," *8th IEEE Computer Society International Conference Digest of Papers* (1974), "Computer Peripherals—Benefactor or Bottleneck?", pp. 185-187.
8. ————, "Microprocessors in CRT Terminals." *Computer* (August 1974).
9. Cushman, R. H., "Microprocessors are Rapidly Gaining on Minicomputers," *EDN*, Vol. 19, No. 10 (May 20, 1974), pp. 16-19.
10. ————, "Microprocessor Benchmarks: How Well Does the Up Move Data?", *EDN* (May 20, 1975).
11. ————, "Understand the 8-Bit Microprocessor: You'll See a Lot of It," *EDN,* Vol. 19, No. 2 (January 20, 1974), pp. 48-54.
12. ————, "What Can You Do with a Microprocessor?" *EDN*, Vol. 19, No. 6 (March 20, 1974), pp. 42-48.
13. Davis, S., "Fresh View of Mini- and Microcomputers," *Computer Design*, Vol, 13, No. 5 (May 1974), pp. 67-79.
14. Falk, H., "Microcomputer Software Makes its Debut," *IEEE Spectrum* (October 1974).
15. Gilder, J. H., "All About Microcomputers," *Computer Decisions* (December 1975), pp. 44-49.
16. Gott, W., "Microprocessors Make Intelligent Data Terminal Systems," *EE/Systems Engineering Today*, Vol. 33, No. 1 (January 1974), pp. 78-80. Part of "Microprocessors Course."
17. Howick, J. F., "Study of Message Routing Within a Microcomputer Network," AD 756573 (December 1972), 45 p., Available from National Technical Information Service, Springfield, Virginia 22151.
18. Jackson, R. E., "Designing a Microprocessor System," *EE/Systems Engineering Today*, Vol. 32, No. 12 (December 1973), pp. 62-67. Part of "Microprocessors Course."
19. Kerr, H., "Microprocessors in Graphic Display Systems," *EE/Systems Engineering Today*, Vol. 32, No. 12 (December 1973), pp. 70-71. Part of "Microprocessors Course."
20. Lewis, D. R. and W. R. Siena, "How to Build a Microcomputer," *Electronic Design*, Vol. 21, No. 18 (September 1, 1973), pp. 60-65.
21. ————, "Microprocessor or Random Logic?" *Electronic Design*, Vol. 21, No. 18 (September 1, 1973), pp. 106-110.
22. Metzger, J., "Forum: It's Go for Microprocessors," *Electronic Products,* Vol. 16, No. 6 (November 19, 1973), pp. 114-127.
23. "Microcomputer Air at a Huge New Market," *Business Week*, No. 2279 (May 12, 1973), pp. 180-182.
24. "Microprocessor-Driven CRT System Can be User-Customized," *Computerworld*, September 13, 1972. Abstract in *SIGMICRO Newsletter*, Vol. 3, No. 3 (October 1972), p. 74.
25. Monico, J. A. (Microsystems International, Ottawa, Canada), "Microprocessors—A Case History of an Integrated Approach," presented at IEEE International Convention (March 26-29, 1974).
26. Murphy, J. P. and others, "Enhancing an LSI Computer to Handle Decimal Data," *Electronics*, Vol. 46, No. 5 (March 1, 1973), pp. 77-82.
27. Ogdin, J. L., "Survey of Microprocessors Reveals Limitless Variety," *EDN*, Vol. 19, No. 8 (April 20, 1974), pp. 38-43.

28. Parasuraman, B., "Applications of LSI Processors," *Wescon Technical Papers* (1973), Paper No. 11/2.

29. Pittman, T., "Improve Interrupt Handling Capability of Microprocessors with a Few IC's," *Electronic Design*, Vol. 24 (November 22, 1974).

30. Reed, J. A., "Application of LSI Microprocessors to Display Terminals," *IEEE International Convention Technical Papers* (1973), Paper No. 21/2.

31. Reyling, G., Jr., "Extend LSI Processor Capabilities," *Electronic Design*, Vol. 22 (October 25, 1974).

32. ————, "LSI Building Blocks for Parallel Digital Processors," *IEEE International Convention Technical Papers* (1973), Paper No. 21/3.

33. ————, "Microprocessors: The Next Generation in Digital Design," *EE/Systems Engineering Today*, Vol. 32, No. 22 (November 1973), pp. 86-90. Part of "Microprocessors Course."

34. ————, "Performance and Control of Multiple Microprocessor Systems," *Computer Design*, Vol. 13, No. 3 (March 1974), pp. 81-86.

35. Schmid, H., "Monolithic Processors," *Computer Design* (October 1974).

36. Seabury, T., "Microprocessors Signal in Traffic Control," *EE/Systems Engineering Today*, Vol. 33, No. 1 (January 1973), pp. 76-78. Part of "Microprocessors Course."

37. Smith, H., "Anatomy of a Microcomputer," *EE/Systems Engineering Today*, Vol. 32, No. 11 (November 1973), pp. 91-94. Part of "Microprocessors Course."

38. ————, "Impact of Microcomputers on the Designer," *Wescon Technical Papers* (1973), Paper No. 11/1.

39. Threewitt, B., "The Microprocessor Rationale," *AFIPS*, Vol. 44 (1973), pp. 3-6.

40. Torrero, E. A., "Focus on Microprocessors," *Electronic Design*, Vol. 18 (September 1, 1974).

41. Theis, D. J., "Microprocessor and Microcomputer Survey," *Datamation* (December 1974).

42. Weisbecker, J., "Simplified Microcomputer Architecture," *Computer*, Vol. 7, No. 3 (March 1974), pp. 41-47.

43. Weiss, C. D., "Basic Microcomputer Software," *Electronic Design*, Vol. 22, No. 9 (April 26, 1974), pp. 142-146.

44. ————, "Microcomputer Programming I Basics," Notes for *NEC Professional Growth in Engineering Institute* (November 1973).

45. ————, "Microcomputer Programming II Producing Software," Notes for *NEC Professional Growth in Engineering Institute* (November 1973).

46. ————, "MOS/LSI Microcomputer Coding," *Electronic Design*, Vol. 22, No. 8 (April 12, 1974), pp. 66-71.

47. ————, "Software for MOS/LSI Microprocessors," *Electronic Design*, Vol. 22, No. 7 (April 1, 1974), pp. 50-57.

48. Weissburger, A. J., "Distributed Function Microprocessor Architecture," *Computer Design* (November 1974).

49. ————, "MOS/LSI Microprocessor Selection," *Electronic Design*, Vol. 12 (June 7, 1974).

50. Wolf, H., "Microcomputers—A New Revolution," *EE/Systems Engineering Today*, Vol. 33, No. 1 (January 1974). pp. 80-83. Part of "Microprocessors Course."

GLOSSARY OF MICROPROCESSOR TERMS

A

ABBREVIATED ADDRESSING:

A modification of the Direct Address mode which uses only part of the full address and provides a faster means of processing data because of the shortened code.

ACCUMULATOR:

One or more registers associated with the ALU which temporarily store sums and other arithmetical and logical results of the ALU.

ADAPTER:

A device used to effect operative capability between different parts of one or more systems or subsystems.

ADDRESSING MODES:

An address is a coded instruction designating the location of data or program segments in storage. The address may refer to storage in registers or memories or both. The address code itself may be stored so that a location may contain the address of data rather than the data itself. This form of addressing is common in microprocessors. Addressing modes vary considerably because of efforts to reduce program execution time.

ALU (ARITHMETIC AND LOGIC UNIT):

The ALU is one of the three essential components of a microprocessor, the other two being the registers and the control block. The ALU performs various forms of addition and subtraction; the logic mode performs such logic operations as ANDing the contents of two registers, or masking the contents of a register.

ARCHITECTURE:

Any design or orderly arrangement perceived by man; the architecture of the microprocessor. Since the extant microprocessors vary considerably in design, their architecture has become a bone of contention among specialists.

ASSEMBLER PROGRAM:

The Assembler Program translates man-readable source statements (mnemonics) into machine understandable object code.

ASSEMBLY LANGUAGE:

A machine oriented language. Normally the program is written as a series of source statements using mnemonic symbols that suggest the definition of the instruction and is then translated into machine language.

ASYNCHRONOUS:

Operation of a switching network by a free-running signal which signals successive instructions, the completion of one instruction triggering the next. There is no fixed time per cycle.

B

BAUD RATE:

A measure of data flow. The number of signal elements per second based on the duration of the shortest element. When each element carries one bit, the Baud rate is numerically equal to bits per second (bps). The Baud rates on UART data sheets are interchangeable with bps.

BCD (BINARY CODED DECIMAL):

Each decimal digit is binary coded into 4-bit words. The decimal number 11 would become 0001 0001 in BCD. Also known as the 8421 code.

BENCHMARK:

Originally a surveyor's mark used as a reference point in surveys. In connection with microprocessors, the benchmark is a frequently used routine or program selected for the purpose of comparing different makes of microprocessors. A flow chart in assembly language is written out for each microprocessor and the execution of the benchmark by each unit is evaluated on paper. It is not necessary to use hardware to measure capability by benchmark.

BIDIRECTIONAL:

A term applied to a port or bus line that can be used to transfer data in either direction.

BINARY:

A system of numbers using 2 as a base in contrast to the decimal system which uses 10 as a base. The binary system requires only two symbols, 0 and 1. Two is expressed in binary by the number 10 (read one, zero). Each digit after the initial 1 is multiplied by the base 2. Hence the following table expresses the first ten numbers in decimal and binary:

DECIMAL	BINARY
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

BRANCH:

Refers to the capability of a microprocessor to modify the function or program sequence. Such modification depends on the actual content of the data being processed at any given instant.

BREAKPOINT:

A program point indicated by a breakpoint flag which invites interruption to give the user the opportunity to check his program before continuing to its completion.

BUFFER:

A circuit inserted between other circuit elements to prevent interactions, to match impedances, to supply additional drive capability, or to delay rate of information flow. Buffers may be inverting or non-inverting.

BUS DRIVER:

An integrated circuit which is added to the data bus system to facilitate proper driver to the CPU when several memories are tied to the data bus line. These are necessary because of capacitive loading which slows down the data rate and prevents proper time sequencing of microprocessor operations.

BUS SYSTEM:

A network of paths inside the microprocessor which facilitate data flow. The important buses in a microprocessor are identified as Data Bus, Address Bus, and Control Bus.

BYTE:

Indicates a pre-determined number of consecutive bits treated as an entity. For example, 4-bit or 8-bit bytes. "Word" and "Byte" are used interchangeably.

C

CLOCK:

A generator of pulses which controls the timing of switching circuits in a microprocessor. Clock frequency is not the only criterion of data manipulation speed. Hardware architecture and programming skill are more important. Clocks are a requisite for most microprocessors and multiple phased clocks are common in MOS processors.

COMBINATIONAL LOGIC:

A circuit arrangement in which the output state is determined by the present state of the input. Also called Combinatorial Logic. (See also Sequential Logic.)

COMPILERS:

Compilers translate higher-level languages into machine codes.

CONDITION CODE:

Refers to a limited group of program conditions such as carry, borrow, overflow, etc.; which are pertinent to the execution of instructions. The codes are contained in a Condition Codes Register.

CONTROL BLOCK:

This is the circuitry which performs the control functions of the CPU. It is responsible for decoding microprogrammed instructions and then generating the internal control signals that perform the operations requested.

CONTROL BUS:

Conveys a mixture of signals which regulate system operation. These "traffic" signals are commands which may also originate in peripherals for transfer to the CPU or the reverse.

CONTROL PROGRAM:

The control Program is a sequence of instructions that will guide the CPU through the various operations it must perform. This program is stored permanently in ROM memory where it can be accessed by the CPU during operations.

CPU (CENTRAL PROCESSING UNIT):

The heart of any computer system. Basically the CPU is made up of storage elements called registers, computational circuits in the ALU, the Control Block, and I/O. As soon as LSI technology was able to build a CPU on an IC chip, the microprocessor became a reality. The one-chip microprocessors have limited storage space, so memory implementation is added in modular fashion. Most current microprocessors consist of a set of chips, one or two of which form the CPU.

CROM (CONTROL READ ONLY MEMORY):

This is a major component in the control block of some microprocessors. It is a ROM which has been microprogrammed to decode control logic.

CROSS-ASSEMBLER:

When the program is assembled by the microprocessor that it will run on, the program that performs the assembly is referred to simply as an assembler. If the program is assembled by some other microprocessor, the process is referred to as cross-assembly. Occasionally the phrase "native assembler" will be used to distinguish it from a cross-assembler.

D

DAISY CHAIN:

A bus line which is interconnected with units in such a way that the signal passes from one unit to the next in serial fashion. The architecture of the Fairchild F-8 provides an example of daisy-chained memory chips. Each chip connects to its neighbors to accomplish daisy-chaining of interrupt priorities beginning with the chip closest to the CPU.

DATA BUS:

The microprocessor communicates internally and externally by means of the data bus. It is bidirectional and can transfer data to and from the CPU, memory storage, and peripheral devices.

DATA COUNTER:

(See Program Counter)

DATA FIELD POINTER:

(See Stack Pointer)

DEBUG:

As used in connection with microprocessor software, debugging involves searching for and eliminating sources of error in programming routines. Finding a bug in software routine is said to be as difficult as finding a needle in the proverbial haystack. A single step tester is the suggested method, so that each instruction operation can be checked individually.

DECREMENT:

A programming instruction which decreases the contents of a storage location. (See also increment and decrement.)

DEDICATED:

To set apart for some special use. A dedicated microprocessor is one that has been specifically programmed for a single application such as weight measurement by scale, traffic light control, etc. ROMs by their very nature (Read-Only) are "dedicated" memories.

DIRECT ADDRESSING:

This is the standard addressing mode. It is characterized by an ability to reach any point in main storage directly. Direct addressing is sometimes restricted to the first 256 bits in main storage.

DMA (DIRECT MEMORY ACCESS):

A method of gaining direct access to main storage to achieve data transfer without involving the CPU. The manner in which CPU is disabled while DMA is in progress differs in different models and some use several methods to accomplish DMA.

E

EXECUTION TIME:

Usually expressed in clock cycles necessary to carry out an instruction. Since the clock frequency is known, the actual time can be calculated. Clock frequencies can be varied.

EXTENDED ADDRESSING:

Refers to an addressing mode that can reach any place in memory. (See also Direct Addressing.)

F

FETCH:

To go after and return with things. In a microprocessor, the "objects" fetched are instructions which are entered in the instruction register. The next, or a later step in the program, will cause the machine to execute what it was programmed to do with the fetched instructions. Often referred to as an "instruction fetch."

FIELDS:

A source statement is made up of a number of code fields, usually four, which are acceptable by the assembler. The four fields may connote Label, Operator, Operand, and Comment. Fields are also applicable to data storage. The eight bits stored in a memory location might contain two 4-bit fields, or eight 1-bit fields, etc.

FIRMWARE:

Software instructions which have been permanently frozen into a ROM are sometimes referred to as Firmware.

FLAG BIT:

An information bit which indicates some form of demarcation has been reached such as overflow or carry. Also an indicator of special conditions such as interrupts.

FLOW CHART OR FLOW DIAGRAM:

A sequence of operations charted with the aid of symbols, diagrams, or other representations to indicate an executive program. Flowcharts enable the designer to visualize the procedure necessary for each item on the program. A complete flowchart leads directly to the final code.

H

HANDSHAKING:

A colloquial term which describes the method used by a Modem to establish contact with another Modem at the other end of a telephone line. Often used interchangeably with buffering and interfacing, but with a fine line of difference in which handshaking implies a direct package to package connection regardless of functional circuitry.

HARDWARE:

The individual components of a circuit, both passive and active have long been characterized as hardware in the jargon of the engineer. Today, any piece of data processing equipment is informally called hardware.

HARD-WIRED LOGIC:

Random Logic design solutions require interconnection of numerous integrated circuits representing the logic elements. An example of hard-wired logic is the use of a hard-wired diode matrix instead of a ROM. These interconnections, whether done with soldering iron or by printed circuit board, are referred to as hard-wired logic in contrast to the software solutions achieved by a programmed ROM or Microprocessor.

HIGH LEVEL LANGUAGE:

This is a problem-oriented programming language as distinguished from a machine-oriented programming language. The former's instruction approach is closer to the needs of the problems to be handled than the language of the machine on which they are to be implemented.

HEXADECIMAL:

Whole numbers in positional notation using 16 as a base. (See Octal and Compare.) Since there are 16 hexadecimal digits (0 through 15) and there are only ten numerical digits (0 through 9) an additional six digits representing 10 through 15 must be introduced. Recourse is had to the alphabet to provide the extra digits. Hence, the least significant hexadecimal digits read: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. The decimal number 16 becomes the hexadecimal number 10. The decimal number 26 becomes the hexadecimal number 1A.

I

IMMEDIATE ADDRESSING:

In this mode of addressing, the operand contains the value to be operated on, and no address reference is required.

INCREMENT (AND DECREMENT):

These two words are software operations most often associated with the stack and stack pointer. Bytes of information are stored in the stack register at the addresses contained in the stack pointer. The stack pointer is decremented after each byte of information is entered into the stack; it is incremented after each byte is removed from the stack. The terms can also refer to any addressable register.

INDEX REGISTER:

The Index Register contains address information subject to modification by the Control Block without affecting the instruction in the memory. The IR information is available for loading onto the stack pointer when needed.

INDIRECT ADDRESSING:

Addressing a memory location which contains the address of data rather than the data itself.

INSTRUCTION SET:

Constitutes the total list of instructions which can be executed by a given microprocessor and is supplied to the user to provide the basic information necessary to assemble a program.

INTERFACE:

Indicates a common boundary between adjacent components, circuits, or systems enabling the devices to yield and/or acquire information from one another. In the face of common usage, one must regretfully add that the words Buffer, Handshake, and Adapter are interchangeable with Interface.

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 interrupt priorities; and, after completion of interrupt service, to resume the interrupted program from the point where it was interrupted.

INTERRUPT MASK BIT:

The Interrupt Mask Bit prevents the CPU from responding to further interrupt requests until cleared by execution of programmed instructions. It may also be manipulated by specific mask bit instructions.

I/O (INPUT/OUTPUT):

Package pins which are tied directly to the internal bus network to enable I/O to interface the microprocessor with the outside world.

J

JUMP:

The Jump operation, like the Branch operation, is used to control the transfer of operations from one point to a more distant point in the control program. Jumps differ from Branching in not using the Relative Addressing mode.

L

LABEL:

A label may correspond to a numerical value or a memory location in the programmable system. The specific absolute address is not necessary since the intent of the label is a general destination. Labels are a requisite for jump and branch instructions.

LIBRARY:

A collection of complete programs written for a particular computer, minicomputer, or microprocessor. For example, Second Order Differential Equation may be the name of a program in the Library of a particular computer; this program will contain all the subroutines necessary to perform the solution of second order differential equations written in machine language and using the instruction set of this machine.

LIFO:

Last-In-First-Out buffer. (See Push Down Stack.)

LOGIC:

A mathematical treatment of formal logic in which a system of symbols is used to represent quantities and relationships. The symbols or logical functions are called AND, OR, NOT, to mention a few examples. Each function can be translated into a switching circuit, more commonly referred to as a "gate." Since a switch (or gate) has only two states—open or closed—it makes possible the application of binary numbers for the solution of problems. The basic logic functions obtained from gate circuits are the foundation of complex computing machines.

LOOK AHEAD:

(1) A feature of the CPU which allows the machine to mask an interrupt request until the following instruction has been completed. (2) A feature of adder circuits and ALUs which allow these devices to look ahead to see that all carrys generated are available for addition.

LOOPING:

Repetition of instructions at delayed speeds until a final value is determined (as in a weight scale indication) is called looping. The looped repititions are usually frozen into a ROM memory location and then jumped to when needed. Looping also occurs when the CPU is in a wait condition.

LSI (LARGE SCALE INTEGRATION):

At the beginning of the LSI era a count of 100 gates qualified for LSI. Today an 8-bit CPU can be fabricated on a single chip.

M

MACHINE LANGUAGE:

The only language the microprocessor can understand is binary. All other programming languages must be translated into binary code before entering the processor and decoded back into the original language after leaving it.

MACRO COMMAND:

A program entity formed by a string of standard, but related, commands which are put into effect by means of a single macro command. Any group of frequently used commands can be combined into a macro command. The many become one.

MNEMONIC CODE:

These are designed to assist the human memory. The microprocessor language consists of binary words which are a series of 0's and 1's making it difficult for the programmer to remember the instructions corresponding to a given operation. To assist the human memory, the binary numbered codes are assigned groups of letters (or mnemonic symbols) that suggest the definition of the instruction. LDA for load accumulator, etc. Source statements can be written in this symbolic language and then translated into machine language.

MICROINSTRUCTION:

(See Microprogram)

MEMORY:

The part of a computer system into which information can be inserted and held for future use. Storage and Memory are interchangeable expressions. Memories accept and hold binary numbers only. Memory types are core, disk, drum, and semiconductor.

MOS (METAL OXIDE SEMICONDUCTOR):

The structure of a MOS Field Effect Transistor (FET) is metal over silicon oxide over silicon. The metal electrode is the gate; the silicon oxide is the insulator; and carrier doped regions in the silicon substrate become the drain and source. The result is a sandwich very much like a capacitor, which explains why MOS is slower than bipolar since the 'capacitor sandwich' must charge up before current can flow. The three great advantages of MOS are its process simplicity because of reduced fabrication stages; the savings in chip real estate resulting in functional density; and the ease of interconnection on chip. These qualities enable MOS to break the LSI barrier, something bipolar is just beginning to achieve. The hand-held calculator and the microprocessor are triumphs of MOS-LSI technology.

MICROPROCESSOR:

The microprocessor is a Central Processing Unit fabricated on one or two chips. While no standard design is visible in existing units, a number of well-delineated areas are present in all of them: Arithmetic & Logic Unit, Control Block, and Register Array. When joined to a memory storage system, the resulting combination is referred to in today's usage as a microcomputer. It should be added that each microprocessor is supplied with an instruction Set, and this software manual may be just as important to the user as the hardware.

MULTIPLEXING:

Multiplexing describes a process of transmitting more than one signal at a time over a single link, route, or channel. Of the two methods in use, one frequency shares the bandwidth of a channel in the same way hurdlers run and jump in their assigned lanes, thus permitting many contestants to compete simultaneously on the same track. The second way is to time-same multiple signals in the same way that pole vaulters jump over the same bar one after the other. The two methods may be described as parallel and serial processing. Time-sharing may not seem "simultaneous" but it should be remembered that the signal speed is so fast that it is possible to multiplex four different numbers through a single decoder-driver and have them appear on four different displays without a flicker to disturb the eye.

N

NESTING:

Nesting is referred to when a subroutine is enclosed inside a larger routine, but is not necessarily part of the outer routine. A series of looping instructions may be nested within each other.

O

OBJECT PROGRAM:

The end result of the source language program after it has been translated into machine language.

OCTAL:

Whole numbers in positional notation using 8 as a base. The decimal or base 10 number, 125, becomes 175 in octal or base 8. Here is a convenient way to convert a decimal number into an octal number:

```
   1   7
 8 15  5
 8 125
```

Divide the decimal number by 8. The answer is 15 and 5 left over. Divide the answer, 15, by 8 again. The answer is 1 and 7 left over. The octal number is 175.

To prove your answer is correct, do the following:

```
5x1 =   5
7x8 =  56
1x64 = 64
       125
```

Arrange the octal number vertically with the least significant digit on top. The least significant digit represents one's, so multiply 5x1 = 5. The next digit in the octal number represents 8's, so multiply 7x8 = 56. The third digit of the octal number represents 64's so multiply 1x64 = 64. The sum is the decimal number 125.

OPERAND:

A quantity on which a mathematical operation is performed. One of the instruction fields in an addressing statement. Usually the statement consists of an operator and an operand. The operator may indicate an "add" instruction; the operand will indicate what is to be added.

OVERFLOW:

Overflow results when an arithmetic operation generates a quantity beyond the capacity of the register. Also referred to as arithmetical overflow. An overflow status bit in the condition code register can be checked to determine if the previous operation caused an overflow.

OPERATING CODE (OPCODE):

Source statements which generate machine codes after assembly are referred to as operating codes.

P

PARALLEL OPERATION:

Processing all the digits of a word or byte simultaneously by transmitting each digit on a separate channel or bus line.

PARTY-LINE:

Party-line as used in its telephone sense to indicate a large number of devices connected to a single line originating in the CPU.

PCI (PARALLEL COMMUNICATIONS INTERFACE):

A Motorola device which interfaces the microprocessor's bus-organized system with incoming serial synchronous communication information. The parallel data of the multi-bus system is serially transmitted by the asynchronous data terminal. The PCI interfaces directly with low-speed Modems to enable microprocessor communications over telephone lines.

PIPELINE:

Computers which execute serial programs only are referred to as pipeline computers.

PLA (PROGRAMMED LOGIC ARRAYS):

The PLA is an orderly arrangement of logical AND logical OR functions. Its application is very much like a glorified ROM. It is primarily a combinational logic device.

POLLING:

Polling is the method used to identify the source of interrupt requests. When several interrupts occur at one time, the control program decides which one to service first.

PORT:

Device terminals which provide electrical access to a system or circuit. The point at which the I/O is in contact with the outside world.

PROGRAM:

A procedure for solving a problem and frequently referred to as Software.

PROGRAM COUNTER:

One of the registers in the CPU which holds addresses necessary to step the machine through the program. During interrupts, the program counter saves the address of the instruction. Branching also

requires loading of the return address in the program counter.

PUSH DOWN STACK:

A register that receives information from the Program Counter and stores the address locations of the instructions which have been pushed down during an interrupt. This stack can be used for subroutining. Its size determines the level of subroutine nesting (one less than its size or 15 levels of subroutine nesting in a 16 word register. When instructions are returned they are popped back on a last-in-first-out (LIFO) basis.

R

RALU (REGISTER, ARITHMETIC, AND LOGIC UNIT):

Unlike the discrete ALU package which functions as an Arithmetic and Logic unit only, the ALU in the microprocessor is equipped with a number of registers.

RAM (RANDOM ACCESS MEMORY):

Random in the sense of providing access to any storage location point in the memory immediately by means of vertical and horizontal co-ordinates. Information may be "written" in or "read" out in the same rapid way.

RANDOM LOGIC DESIGN:

Designing a system using discrete logic circuits. Numerous gates are required to implement the logic equations until the problem is solved. Even then, the design is not completed until all redundant gates are weeded out. Random logic design is no guarantee of optimum gate count.

REAL TIME OPERATION:

Data processing technique used to allow the machine to utilize information as it becomes available, as opposed to batch processing at a time unrelated to the time the information was generated.

REGISTER:

A register is a memory on a smaller scale. The words stored therein may involve arithmetical, logical, or transferral operation. Storage in registers may be temporary, but even more important is their accessibility by the CPU. The number of registers in a microprocessor is considered one of the most important features of its architecture.

RELATIVE ADDRESSING:

The relative addressing mode specifies a memory location in the CPU's Program Location Counter register. This addressing mode is used for Branch instructions in which case an opcode is added to the Relative Address to complete the branching instruction.

ROM (READ ONLY MEMORY):

In its virgin state the ROM consists of a mosaic of undifferentiated cells. One type of ROM is programmed by mask pattern as part of the last manufacturing stage. Another more popular type, better known as P/ROM, is programmable in the field with the aid of programmer equipment. Program data stored in ROMs are often called firmware because they cannot be altered. However, another type of P/ROM is now on the market called EPROM which is erasible by ultra violet irradiation and electrically reprogrammable.

S

SCRATCHPAD:

This term is applied to information which the Processing unit stores or holds temporarily. It is a memory containing subtotals for various unknowns which are needed for final results.

SEQUENTIAL LOGIC:

A circuit arrangement in which the output state is determined by the previous state of the input. (See also Combination Logic.)

SOFTWARE:

What sheet music is to the piano, software is to the computer. Looked at from a practical point of view, one might say that software is the computer's instruction manual. The name, software, was obviously chosen to contrast with the formidable hardware which confronted the first programmers. Software is the language used by a programmer to communicate with the computer. Since the only language spoken by a computer is mathematical, the programmer must convert his verbal instructions into numbers. In the case of microprocessors, which vary from maker to maker, software libraries are assembled by the manufacturer for the benefit of the user.

SOURCE STATEMENT:

A program written in other than machine language, usually in three letter mnemonic symbols, that suggest the definition of the instruction. There are two kinds of source statements: "executive instructions" which translate into operating machine code (opcode); and "assembly directives" which are useful in documenting the source program, but generate no code.

SIMULATOR:

A special program that simulates the logical operation of the microprocessor. It is designed to execute object programs generated by a cross-assembler on a machine other than the one being worked on and is useful for checking and debugging programs prior to committing them to ROM firmware.

STACK:

The stack is a block of successive memory locations which is accessible from one end on a last-in-first-out basis (LIFO). The stack is coordinated with the stack pointer which keeps track of storage and retrieval of each type of information in the stack. A stack may be any block of successive information locations in the read/write memory.

SLICE:

A type of chip architecture which permits the cascading or stacking of devices to increase word bit size.

STACK POINTER:

The stack pointer is coordinated with the storing and retrieval of information in the stack. The stack pointer is decremented by one immediately following the storage in the stack of each byte of information. Conversely, the stackpointer is incremented by one immediately before retrieving each byte of information from the stack. The stack pointer may be manipulated for transferring its contents to the Index register or vice versa.

STATUS WORD REGISTER:

A group of binary numbers which informs the user of the present condition of the microprocessor. In the Fairchild F8, the Status Register provides the following five pieces of information: plus or minus sign of the value in Accumulator, overflow indication, carry bit, all zero's in accumulator, and interrupt bit status.

STORAGE:

The word starage is used interchangeably with memory. In fact, it has been recommended as the preferred term by people who would rather not imply that the computer has any relationship with the human brain.

SUBROUTINE:

Part of the master routine which may be used at will in a variety of master routines. The object of a Branch or Jump Command.

T

THROUGHPUT:

The speed with which problems or segments of problems are performed is called Throughput. Divined in this way, it is obvious that throughput will vary from application to application. As an index of speed, throughput is meaningful only in terms of your own application.

TWO'S COMPLEMENT NUMBERS:

The ALU performs standard binary addition using the 2's complement numbering system to represent both positive and negative numbers. The positive numbers in 2's complement representation are identical to the positive numbers in standard binary.

+127 in standard binary = 01111111 +127 in 2's complement = 01111111. Note that the eighth or most significant digit indicates the sign: 0 = plus, 1 = minus.

However, the negative 2's complement is the reverse of the negative standard binary plus 1.

-127 in standard binary = 11111111. To form the 2's complement of -127:

First reverse all the digits except the sign:
= 10000000
1
10000001 = -127 in 2's complement.

U

UART (UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER):

This device will interface a word parallel controller or data terminal to a bit serial communication network.

V

VECTORED INTERRUPT:

This term is used to describe a microprocessor system in which each interrupt, both internal and external, have their own uniquely recognizable address. This enables the microprocessor to perform a set of specified operations which are preprogrammed by the user to handle each interrupt in a distinctively different manner.

W

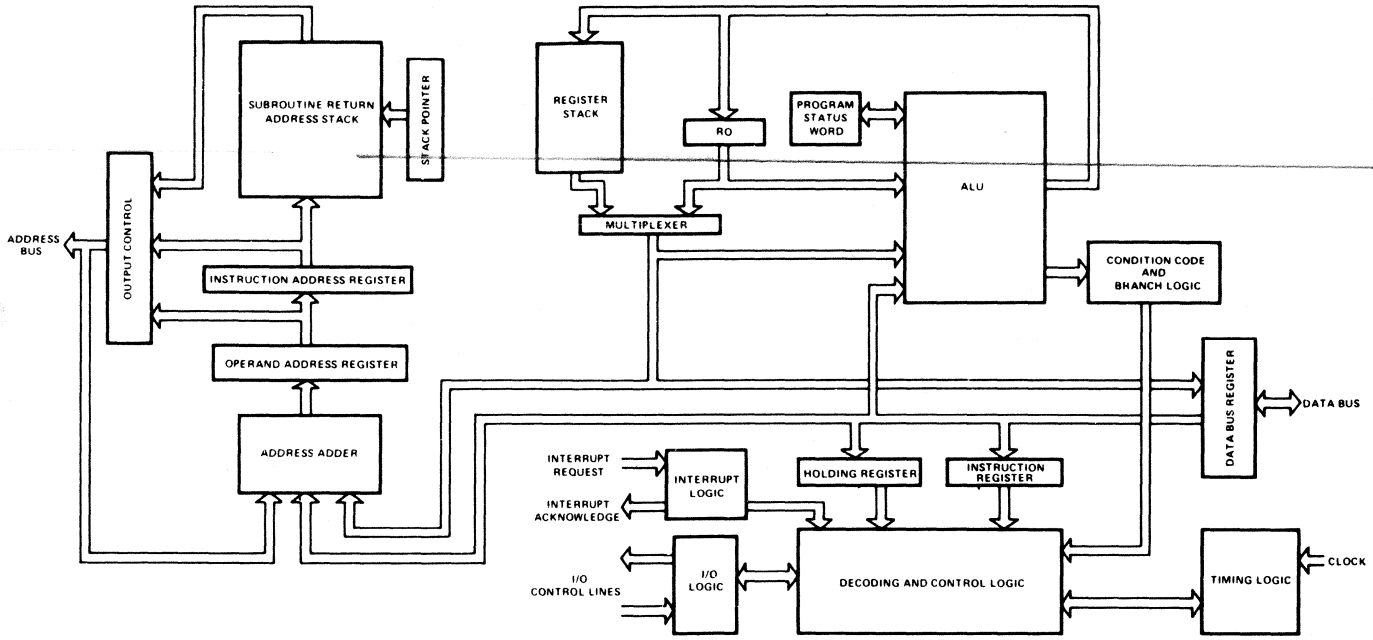
WORD:

A group of "characters" treated as a unit and given a single location in computer memory. Presumably a byte is a group of bits in contrast to a word which is a group of numeric and/or alphabetic characters and symbols, but the two words are used interchangeably more often than not.

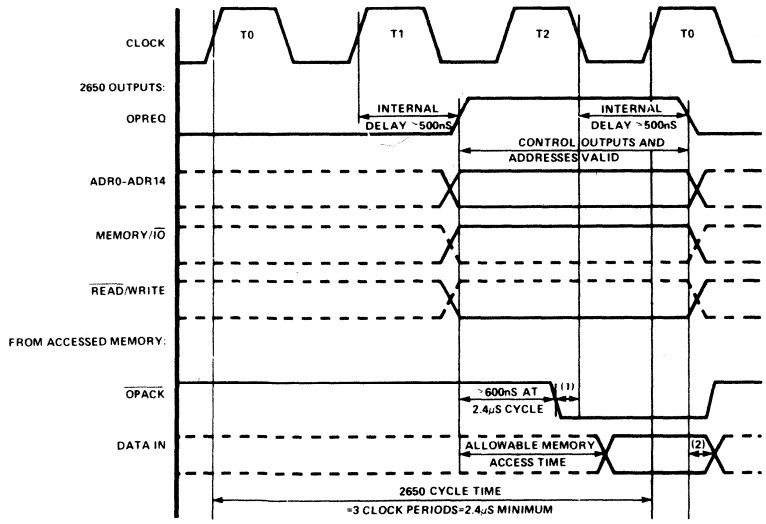
APPENDIX A
Signetics 2650 Microprocessor Specifications

The following data summarize the hardware and software characteristics of the Signetics 2650 Microprocessor. For a more detailed description of the 2650 characteristics and operation, the reader should refer to the Signetics 2650 Microprocessor manual.

BLOCK DIAGRAM 2650 MICROPROCESSOR

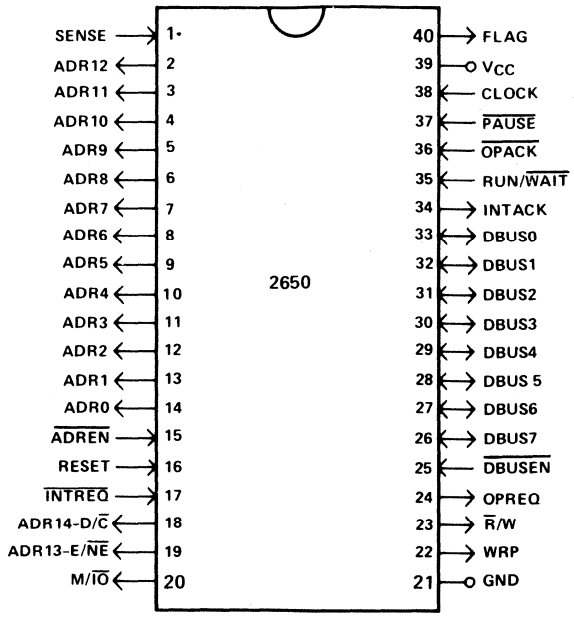


MEMORY READ CYCLE TIMING



NOTES: (1) $\overline{\text{OPACK}}$ must go low at least 100 nS before the trailing edge of T2 in order not to slow down the 2650.
(2) DATA IN signals must be valid for 50nS after the trailing edge of OPREQ.

PIN CONFIGURATION



INSTRUCTION FORMATS

(Z) REGISTER ADDRESSING

(I) IMMEDIATE ADDRESSING

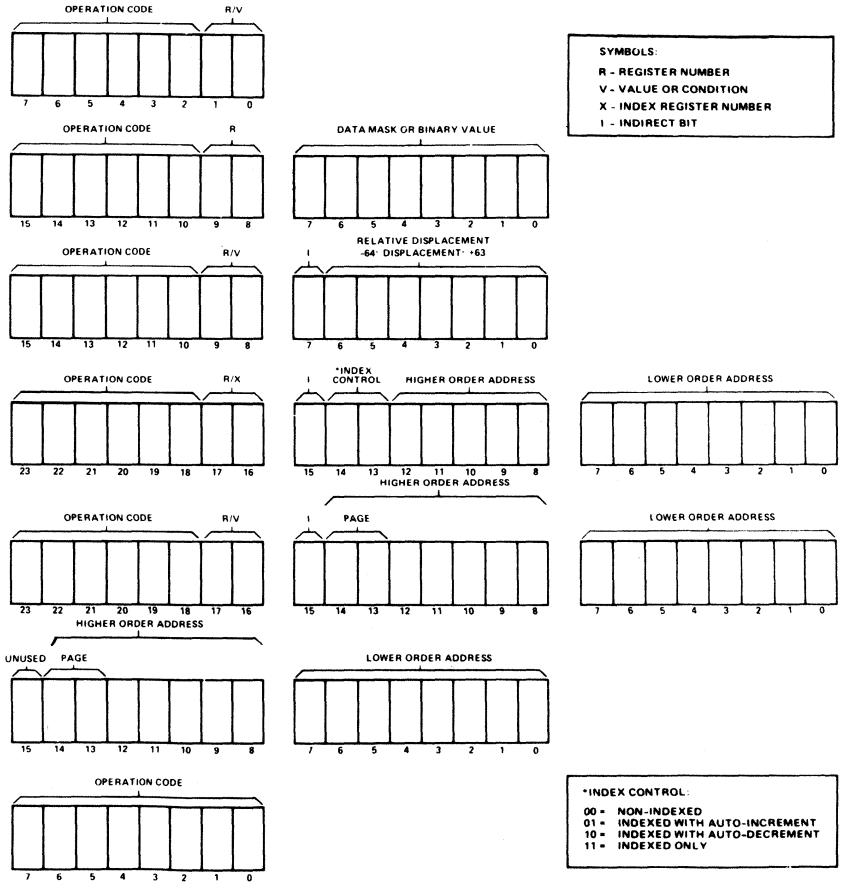
(R) RELATIVE ADDRESSING

(A) ABSOLUTE ADDRESSING (NON-BRANCH INSTRUCTIONS)

(B) ABSOLUTE ADDRESSING (BRANCH INSTRUCTIONS)

INDIRECT ADDRESSING

(E) MISCELLANEOUS INSTRUCTIONS



2650 MICROPROCESSOR INSTRUCTION SET

MNEMONIC	OP CODE	FORMAT*	DESCRIPTION OF OPERATION	AFFECTS	CYCLES
LOD Z	000 000	1Z	Load Register Zero	CC (Note 1)	2
LOD I	000 001	2I	Load Immediate	CC (Note 1)	2
LOD R	000 010	2R	Load Relative	CC (Note 1)	3
LOD A	000 011	3A	Load Absolute	CC (Note 1)	4
STR Z	110 000	1Z	Store Register Zero (r ≠ 0)	CC (Note 1)	2
STR R	110 010	2R	Store Relative		3
STR A	110 011	3A	Store Absolute		4
ADD Z	100 000	1Z	Add to Register Zero w/wo Carry	C, CC (Note 1), IDC, OVF	2
ADD I	100 001	2I	Add Immediate w/wo Carry	C, CC (Note 1), IDC, OVF	2
ADD R	100 010	2R	Add Relative w/wo Carry	C, CC (Note 1), IDC, OVF	3
ADD A	100 011	3A	Add Absolute w/wo Carry	C, CC (Note 1), IDC, OVF	4
SUB Z	101 000	1Z	Subtract from Register Zero w/wo Borrow	C, CC (Note 1), IDC, OVF	2
SUB I	101 001	2I	Subtract Immediate w/wo Borrow	C, CC (Note 1), IDC, OVF	2
SUB R	101 010	2R	Subtract Relative w/wo Borrow	C, CC (Note 1), IDC, OVF	3
SUB A	101 011	3A	Subtract Absolute w/wo Borrow	C, CC (Note 1), IDC, OVF	4
DAR	100 101	1Z	Decimal Adjust Register	CC (Note 2)	3
AND Z	010 000	1Z	AND to Register Zero (r ≠ 0)	CC (Note 1)	2
AND I	010 001	2I	AND Immediate	CC (Note 1)	2
AND R	010 010	2R	AND Relative	CC (Note 1)	3
AND A	010 011	3A	AND Absolute	CC (Note 1)	4
IOR Z	011 000	1Z	Inclusive OR to Register Zero	CC (Note 1)	2
IOR I	011 001	2I	Inclusive OR Immediate	CC (Note 1)	2
IOR R	011 010	2R	Inclusive OR Relative	CC (Note 1)	3
IOR A	011 011	3A	Inclusive OR Absolute	CC (Note 1)	4
EOR Z	001 000	1Z	Exclusive OR to Register Zero	CC (Note 1)	2
EOR I	001 001	2I	Exclusive OR Immediate	CC (Note 1)	2
EOR R	001 010	2R	Exclusive OR Relative	CC (Note 1)	3
EOR A	001 011	3A	Exclusive OR Absolute	CC (Note 1)	4
COM Z	111 000	1Z	Compare to Register Zero Arithmetic/Logical	CC (Note 3)	2
COM I	111 001	2I	Compare Immediate Arithmetic/Logical	CC (Note 4)	2
COM R	111 010	2R	Compare Relative Arithmetic/Logical	CC (Note 4)	3
COM A	111 011	3A	Compare Absolute Arithmetic/Logical	CC (Note 4)	4
RRR	010 100	1Z	Rotate Register Right w/wo Carry	C, CC, IDC, OVF	2
RRL	110 100	1Z	Rotate Register Left w/wo Carry	C, CC, IDC, OVF	2
BCT R	000 110	2R	Branch On Condition True Relative	–	3
BCT A	000 111	3B	Branch On Condition True Absolute	–	3
BCF R	100 110	2R	Branch On Condition False Relative	–	3
BCF A	100 111	3B	Branch On Condition False Absolute	–	3
BRN R	010 110	2R	Branch On Register Non-Zero Relative	–	3
BRN A	010 111	3B	Branch On Register Non-Zero Absolute	–	3
BIR R	110 110	2R	Branch On Incrementing Register Relative	–	3
BIR A	110 111	3B	Branch On Incrementing Register Absolute	–	3
BDR R	111 110	2R	Branch On Decrementing Register Relative	–	3
BDR A	111 111	3B	Branch On Decrementing Register Absolute	–	3
ZBRR	100 110 11	2ER	Zero Branch Relative, Unconditional	–	3
BXA	100 111 11	3EB	Branch Indexed Absolute, Unconditional (Note 5)	–	3

***FORMAT CODE: The number indicates the number of bytes. The letter(s) indicate the format type(s). See other side**

NOTES:
1. Condition code (CC1, CC0): 01 if positive, 00 if zero, 10 if negative.
2. Condition code is set to a meaningless value.
3. Condition code (CC1, CC0): 01 if R0 > r, 00 if R0 = r, 10 if R0 < r.
4. Condition code (CC1, CC0): 01 if r > V, 00 if r = V, 10 if r < V.
5. Index register must be register 3 or 3'.
6. Condition code (CC1, CC0): 00 if all selected bits are 1s, 10 if not all the selected bits are 1s.

PROGRAM STATUS WORD

PSU

7	6	5	4	3	2	1	0
S	F	II	Not Used	Not Used	SP2	SP1	SP0

S Sense
F Flag
II Interrupt Inhibit
SP2 Stack Pointer Two
SP1 Stack Pointer One
SP0 Stack Pointer Zero

PSL

7	6	5	4	3	2	1	0
CC1	CC0	IDC	RS	WC	OVF	COM	C

CC1 Condition Code One
CC0 Condition Code Zero
IDC Interdigit Carry
RS Register Bank Select
WC With/Without Carry
OVF Overflow
COM Logical/Arith. Compare
C Carry/Borrow

2650 MICROPROCESSOR INSTRUCTION SET

	MNEMONIC	OP CODE	FORMAT*	DESCRIPTION OF OPERATION	AFFECTS	CYCLES
SUBROUTINE BRANCH/RETURN	BST R	001 110	2R	Branch To Subroutine On Condition True, Relative	SP	3
	BST A	001 111	3B	Branch To Subroutine On Condition True, Absolute	SP	3
	BSF R	101 110	2R	Branch To Subroutine On Condition False, Relative	SP	3
	BSF A	101 111	3B	Branch To Subroutine On Condition False, Absolute	SP	3
	BSN R	011 110	2R	Branch To Subroutine On Non-Zero Register, Relative	SP	3
	BSN A	011 111	3B	Branch To Subroutine On Non-Zero Register, Absolute	SP	3
	ZBSR	101 110 11	2ER	Zero Branch To Subroutine Relative, Unconditional	SP	3
	BSXA	101 111 11	3EB	Branch To Subroutine, Indexed, Absolute Unconditional (note 5)	SP	3
	RET C	000 101	1Z	Return From Subroutine, Conditional	SP	3
	RET E	001 101	1Z	Return From Subroutine and Enable Interrupt, Conditional	SP, II	3
INPUT/OUTPUT	WRTD	111 100	1Z	Write Data	–	2
	REDD	011 100	1Z	Read Data	CC (Note 1)	2
	WRTC	101 100	1Z	Write Control	–	2
	REDC	001 100	1Z	Read Control	CC (Note 1)	2
	WRTE	110 101	2I	Write Extended	–	3
	REDE	010 101	2I	Read Extended	CC (Note 1)	3
MISC.	HALT	010 000 00	1E	Halt, Enter Wait State	–	2
	NOP	110 000 00	1E	No Operation	–	2
	TMI	111 101	2I	Test Under Mask Immediate	CC (Note 6)	3
PROGRAM STATUS	LPS U	100 100 10	1E	Load Program Status, Upper	F, II, SP	2
	LPS L	100 100 11	1E	Load Program Status, Lower	CC, IDC, RS, WC, OVF, COM, C	2
	SPS U	000 100 10	1E	Store Program Status, Upper	CC (Note 1)	2
	SPS L	000 100 11	1E	Store Program Status, Lower	CC (Note 1)	2
	CPS U	011 101 00	2EI	Clear Program Status, Upper, Masked	F, II, SP	3
	CPS L	011 101 01	2EI	Clear Program Status, Lower, Masked	CC, IDC, RS, WC, OVF, COM, C	3
	PPS U	011 101 10	2EI	Preset Program Status, Upper, Masked	F, II, SP	3
	PPS L	011 101 11	2EI	Preset Program Status, Lower, Masked	CC, IDC, RS, WC, OVF, COM, C	3
	TPS U	101 101 00	2EI	Test Program Status, Upper, Masked	CC (Note 6)	3
	TPS L	101 101 01	2EI	Test Program Status, Lower, Masked	CC (Note 6)	3

***FORMAT CODE: The number indicates the number of bytes. The letter(s) indicate the format type(s). See other side**

NOTES:
1. Condition code (CC1, CC0): 01 if positive, 00 if zero, 10 if negative.
2. Condition code is set to a meaningless value.
3. Condition code (CC1, CC0): 01 if R0 > r, 00 if R0 = r, 10 if R0 < r.
4. Condition code (CC1, CC0): 01 if r > V, 00 if r = V, 10 if r < V.
5. Index register must be register 3. or 3'
6. Condition code (CC1, CC0): 00 if all selected bits are 1s, 10 if **not** all the selected bits are 1s.

PROGRAM STATUS WORD

PSU

7	6	5	4	3	2	1	0
S	F	II	Not Used	Not Used	SP2	SP1	SP0

S Sense
F Flag
II Interrupt Inhibit
SP2 Stack Pointer Two
SP1 Stack Pointer One
SP0 Stack Pointer Zero

PSL

7	6	5	4	3	2	1	0
CC1	CC0	IDC	RS	WC	OVF	COM	C

CC1 Condition Code One
CC0 Condition Code Zero
IDC Interdigit Carry
RS Register Bank Select
WC With/Without Carry
OVF Overflow
COM Logical/Arith. Compare
C Carry/Borrow

APPENDIX B
Microcomputer Programming Techniques with Illustrations

The successful design and implementation of microcomputer-based systems is largely determined by the extent to which the microcomputer program is systematically organized. A well-structured program has significant benefits in terms of modularity and ease of testing/debugging/modification. These features, in turn, lead to significant cost savings during the system development cycle (outlined in Chapter I).

As a result of the research work done by a number of people, certain useful programming methodologies have evolved that lead to well-structured programs. The objective of this section is to present the technique called structured flowcharting together with other conventional programming methods.

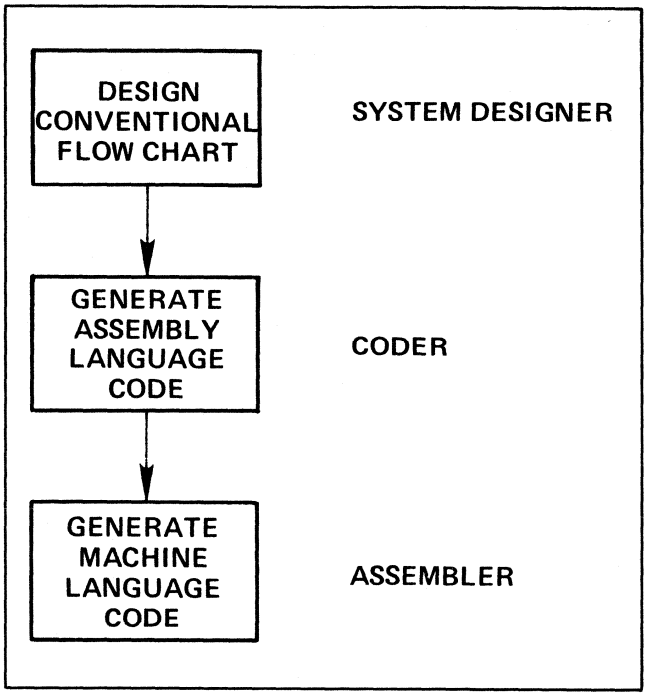


Figure B.1 Conventional Programming Process

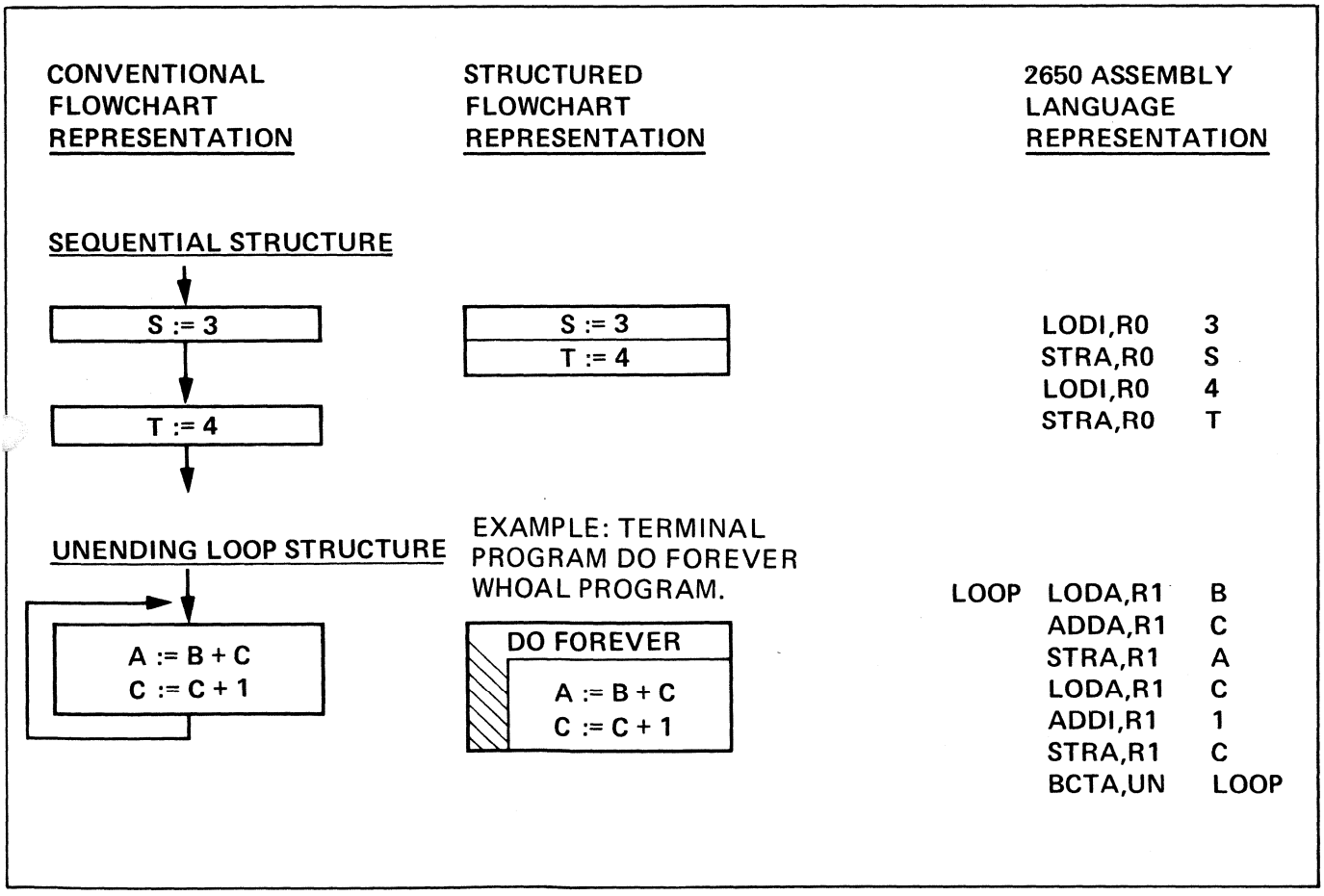


Figure B.2 Iterative Loop Program Structures

For illustrative purposes, the structured flow charts for the ITS are included; these can be compared to the assembly language listing for the ITS, included in Appendix C. The presentation is mainly for the more advanced reader and is meant to be informative rather than tutorial.

There are several methods of generating a microcomputer program to implement a functional specification stated in plain English. The conventional way of programming is illustrated in Figure B.1. The process begins by drawing a conventional flow chart depicting a sequence of steps, based on a rather loose set of rules; the end product is a machine language (i.e., binary) program. The main disadvantages of this technique are that the conventional flow chart is (1) not easy to visualize, and (2) difficult to debug. To alleviate these difficulties, a technique called structured flowcharting was developed. The essential features of this technique are that it starts with a set of program blocks that are essentially the same as the functional specification and, then, the programmer works down to the detailed block diagrams in a systematic, well-defined fashion. Thus, the overall program structure is easy to visualize, at all times. To aid in the understanding of basic structured flowcharting blocks, the conventional and structured flow charts are shown together with the corresponding assembly language code. Figure B.2 shows the basic building blocks for iteration loops of three types: (a) forever, (b) fixed number of times, and (c) based on a decision.

Figure B.3 presents the possible conditional logic flow charts. Figure B.4 shows the flow charts associated with usage of subroutines and Figure B.5 presents those for communication in the input and output to external devices.

The objective of the programmer is to use only these structured programming blocks to implement the functional specification. By way of illustration, the structured flow charts for the ITS discussed in Section 3.2 are presented in Figures B.6 to B.17. The corresponding assembly language listing is presented in Appendix C. The interested reader is urged to compare the structured flow charts in this section with the corresponding subroutines in Appendix C.

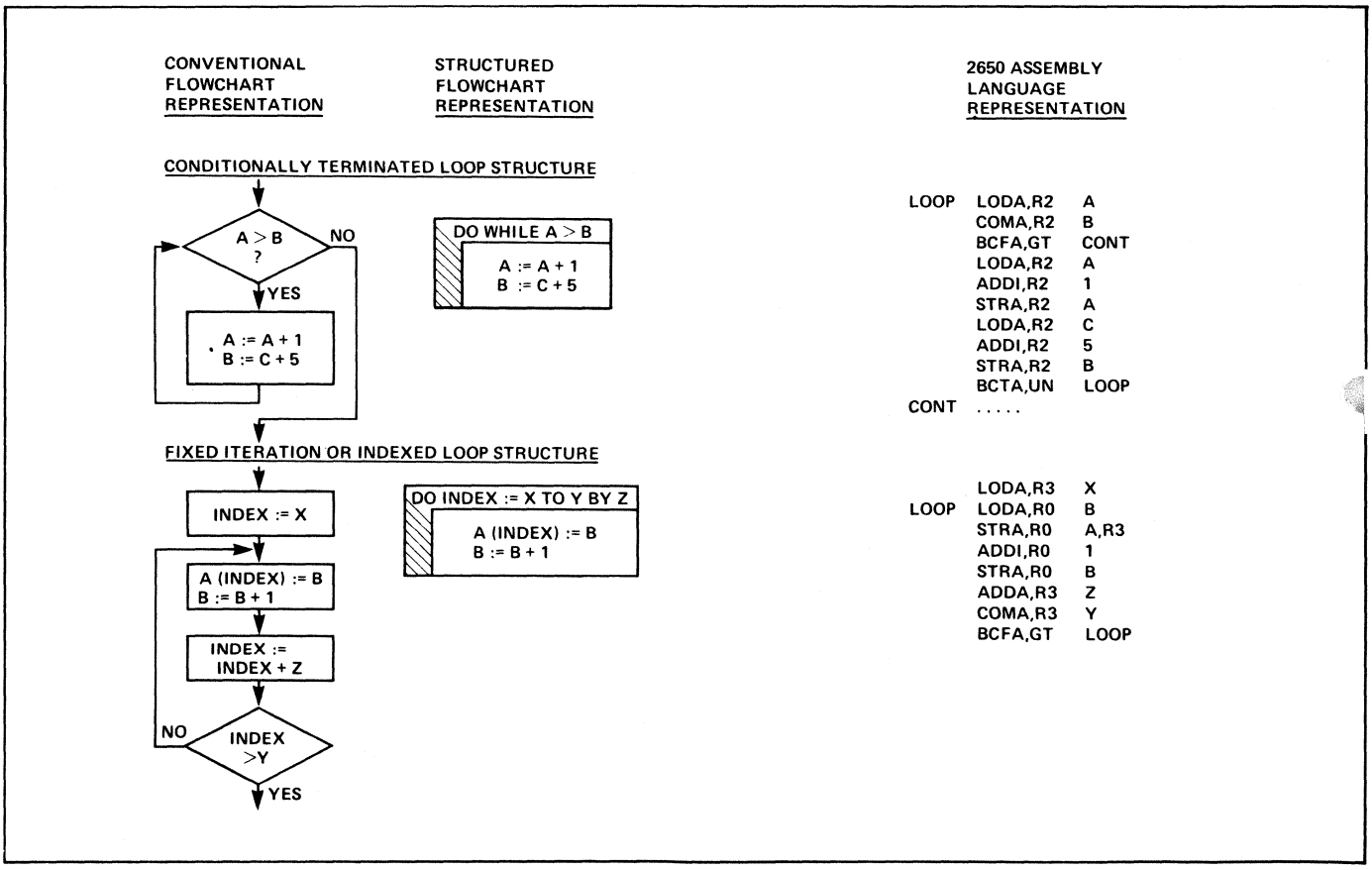


Figure B.2 Iterative Loop Program Structures (Cont.)

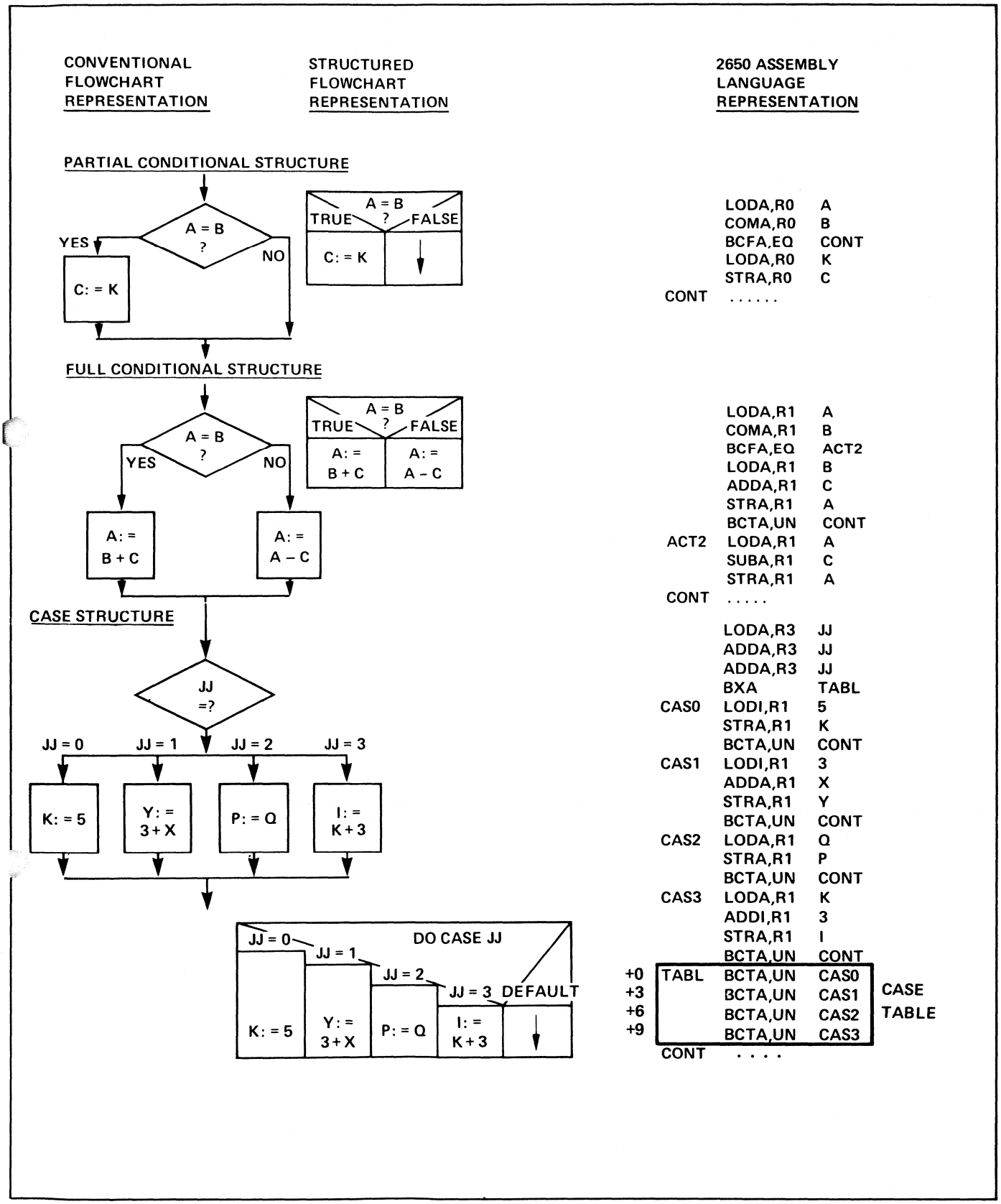


Figure B.3 Conditional Logic Program Structures

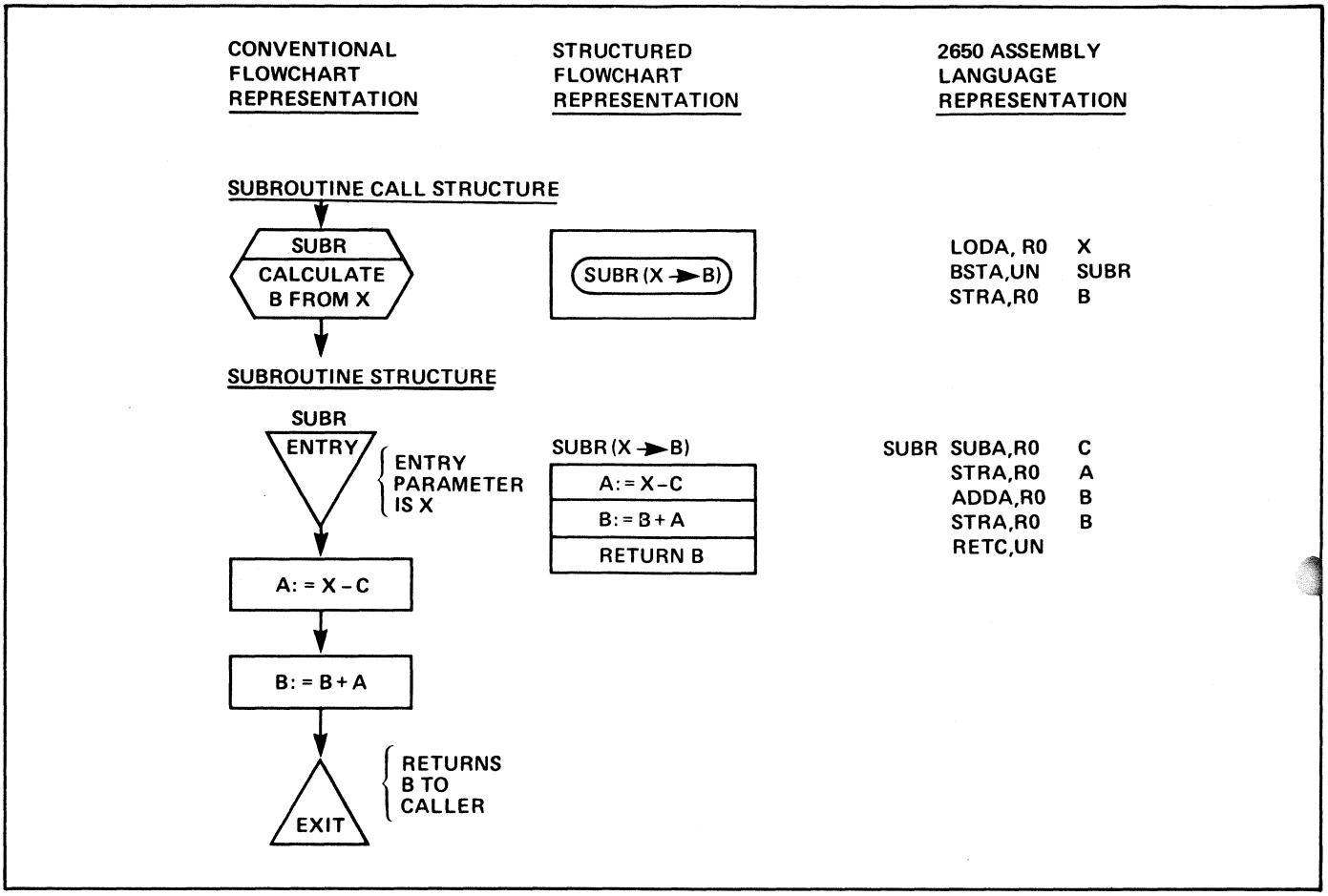


Figure B.4 Subroutine Program Structures

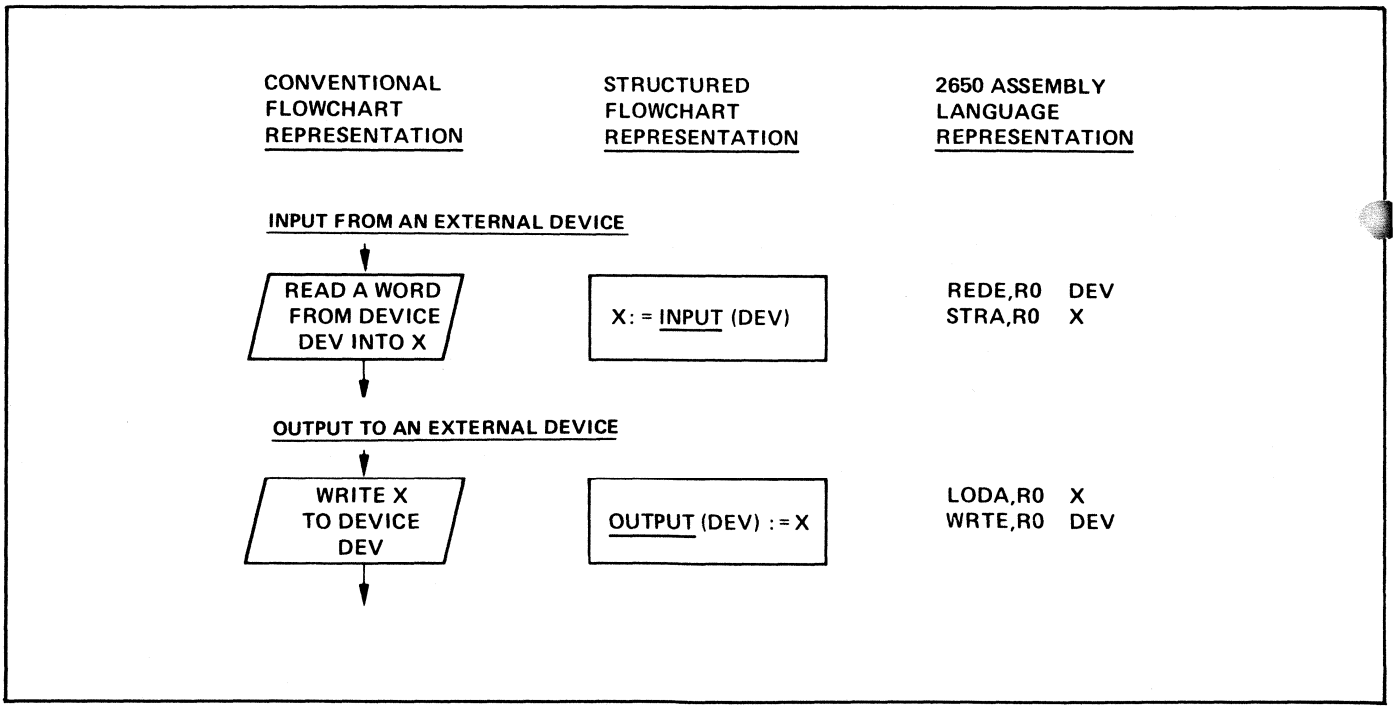
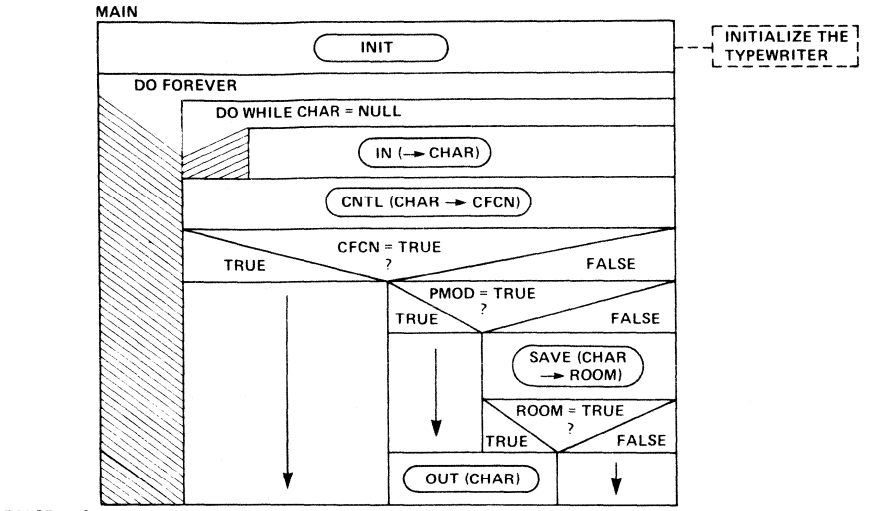


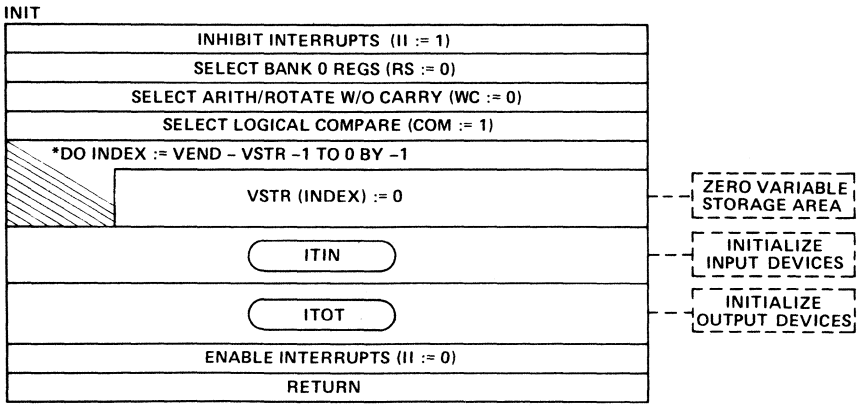
Figure B.5 Input/Output Program Structures

The following pages document the structured flow charts for the general purpose, interrupt driven, full duplex, parallel I/O version of the ITS discussed in Section 3.4



FALSE = 0
TRUE = -1
NULL = 0
CHAR = CHARACTER OBTAINED FROM THE KEYBOARD INPUT BUFFER (CARRIED IN R1 IN PROGRAM)
PMOD = BOOLEAN FLAG WHICH INDICATES IF THE TYPEWRITER IS IN PRINT MODE
CFCN = BOOLEAN FLAG WHICH INDICATES IF CHAR REQUESTS A CONTROL FUNCTION (INDICATED BY CC IN PROGRAM)
ROOM = BOOLEAN FLAG WHICH INDICATES IF THERE IS ROOM FOR CHAR IN THE TYPEWRITER TEXT BUFFER (INDICATED BY CC IN PROGRAM)

Figure B.6 TSB3 Program



VSTR = ADDRESS OF THE 1ST BYTE OF VARIABLE STORAGE
VEND = ADDRESS OF THE LAST BYTE OF VARIABLE STORAGE +1

*THE FOLLOWING DATA ARE SET BY THIS LOOP:

KBUF = 0 — KEYBOARD PROCESSING MODULE DATA

POPT = 0, PIPT = 0
PBUF (0) TO PBUF (PEND -1) = 0
PTFG = FALSE
— PRINTER PROCESSING MODULE DATA

SOPT = 0, SIPT = 0
SBUF (0) TO SBUF (SEND -1) = 0
— SAVE BUFFER MODULE DATA

PMOD = FALSE — GLOBAL DATA

Figure B.7 Initialize Typewrite Module

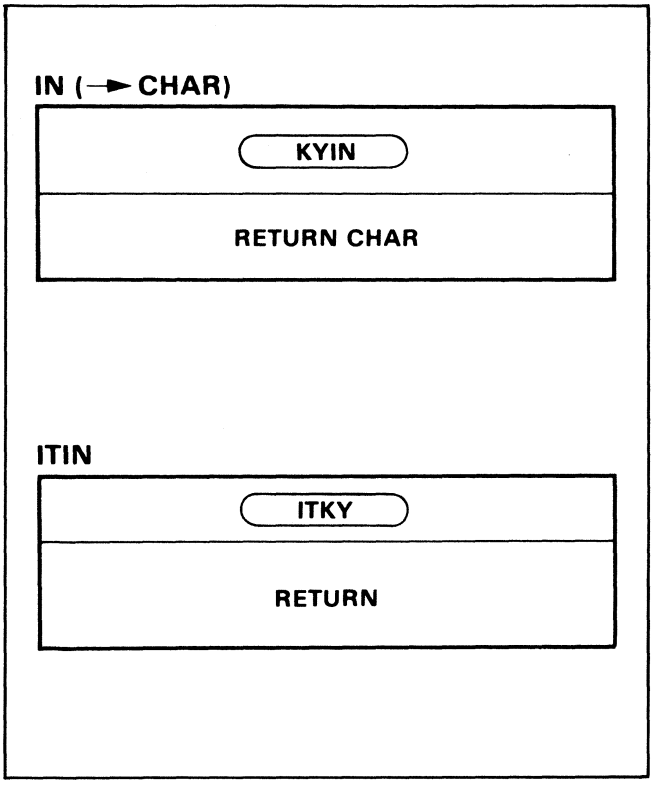


Figure B.8 Input Control Module

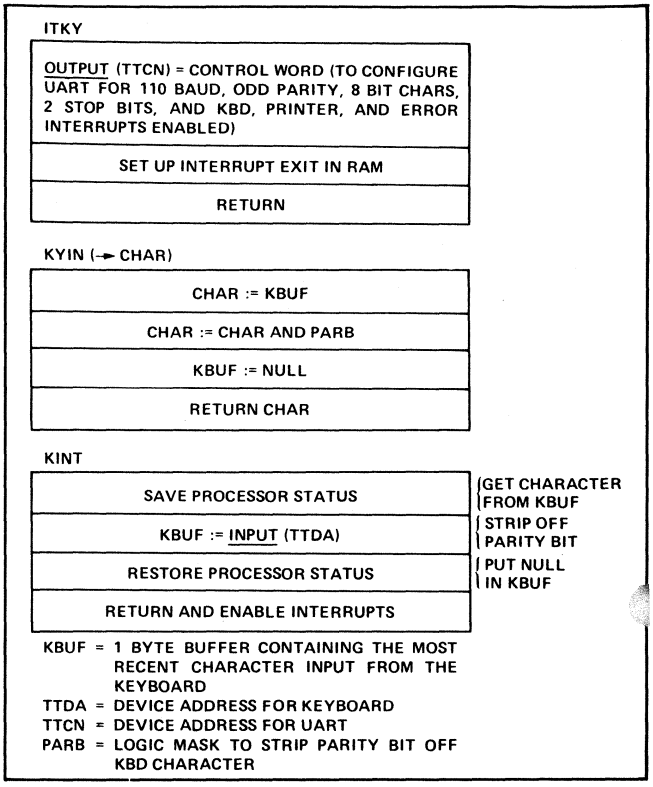


Figure B.9 Keyboard Processing Module

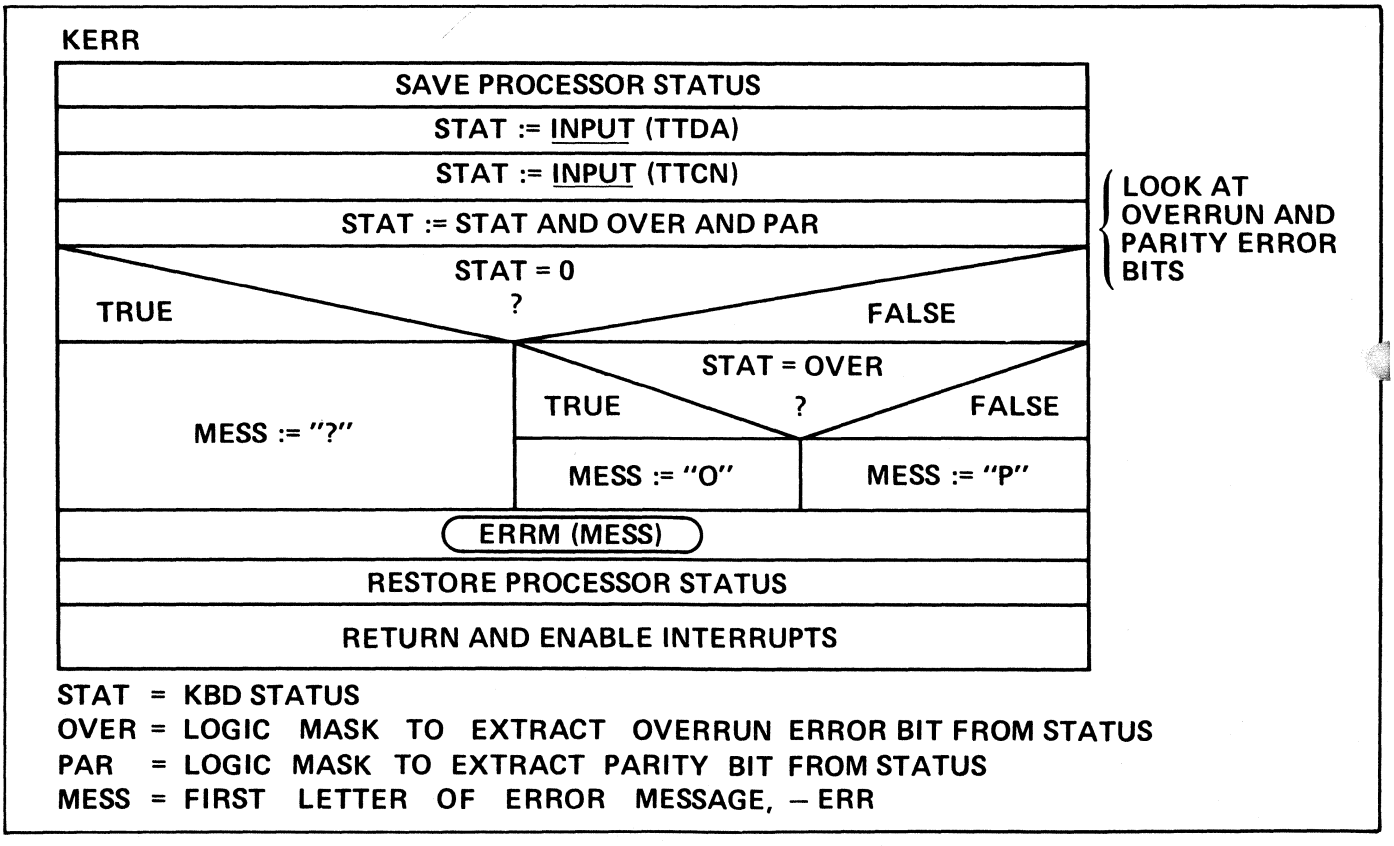


Figure B.10 Keyboard Processing Module (Cont.)

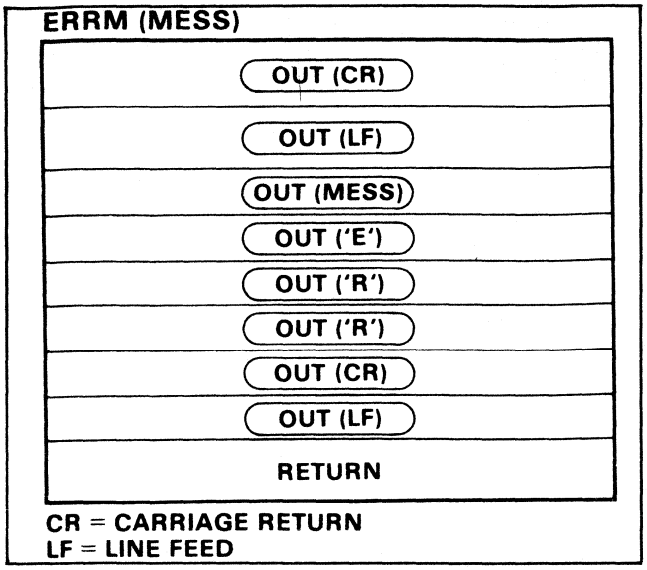


Figure B.11 Keyboard Processing Module (Cont.)

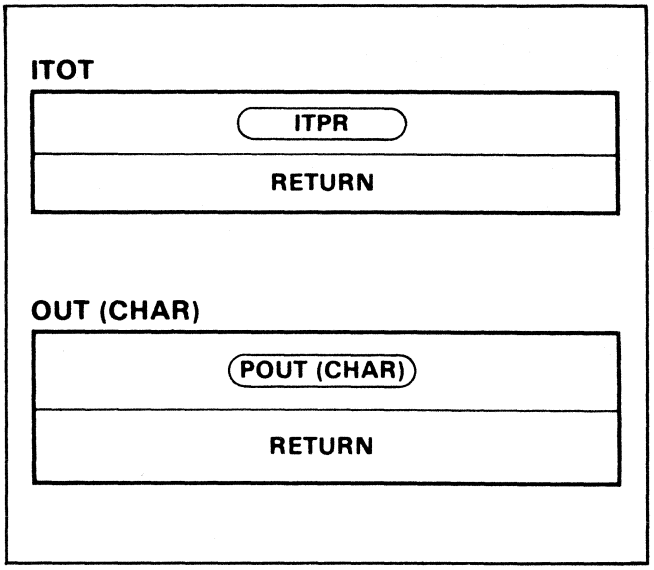


Figure B.12 Output Control Module

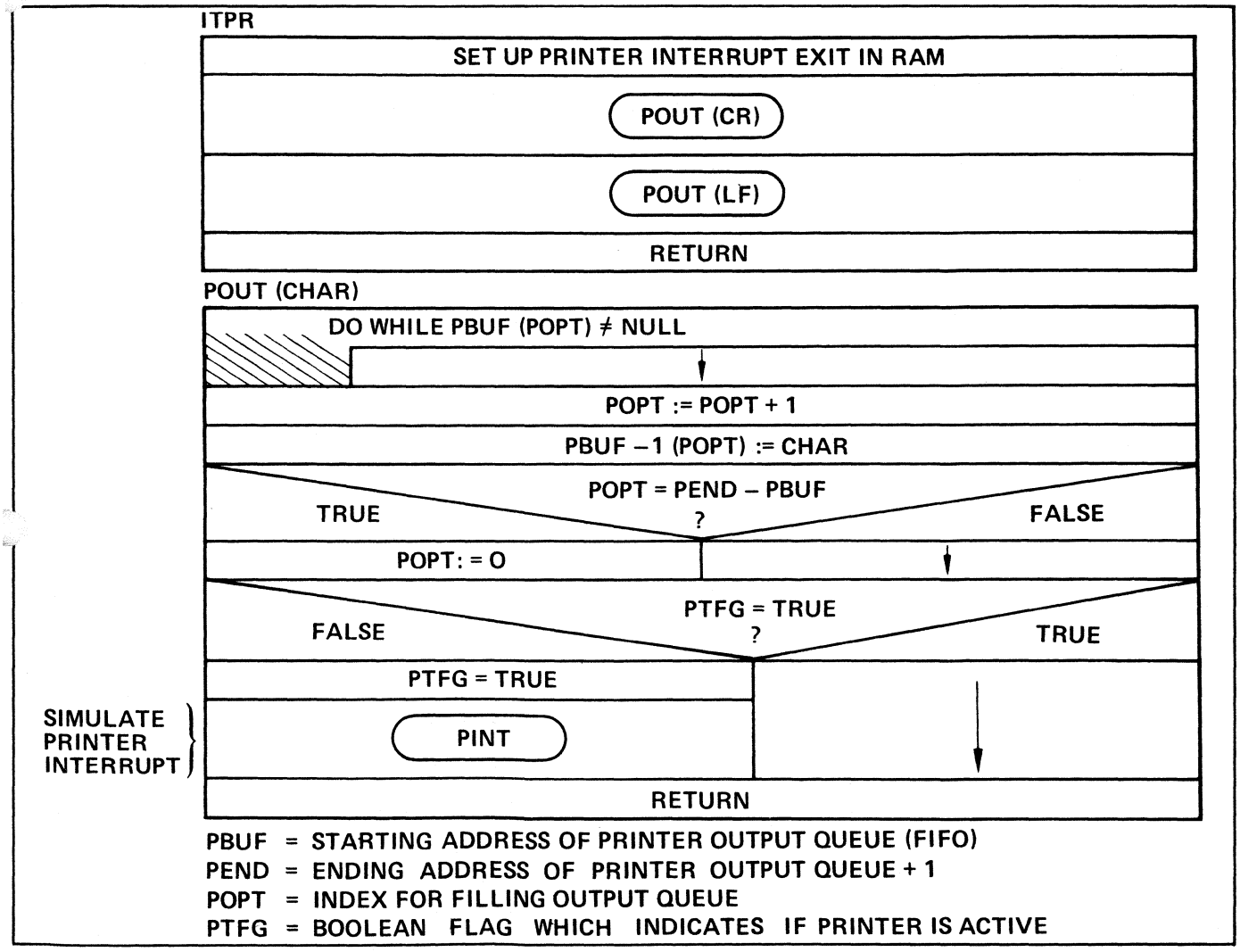


Figure B.13 Printer Processing Module

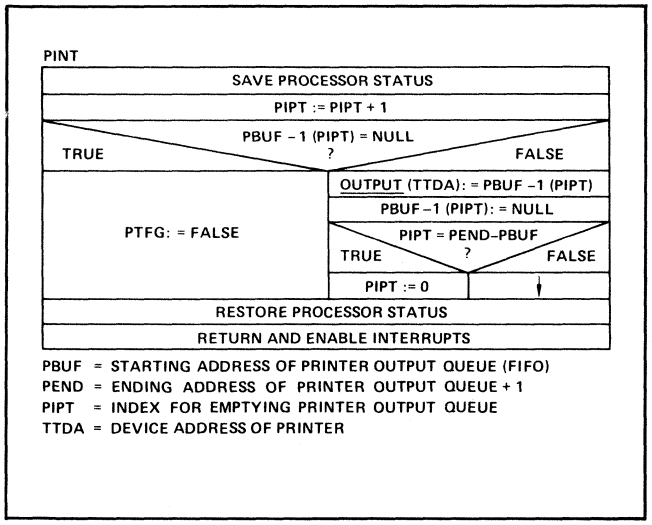


Figure B.14 Printer Processing Module (Cont.)

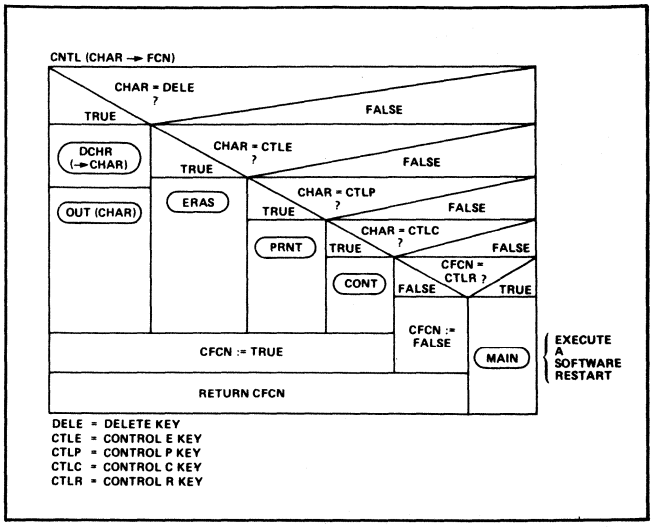


Figure B.15 Input Character Decode Module

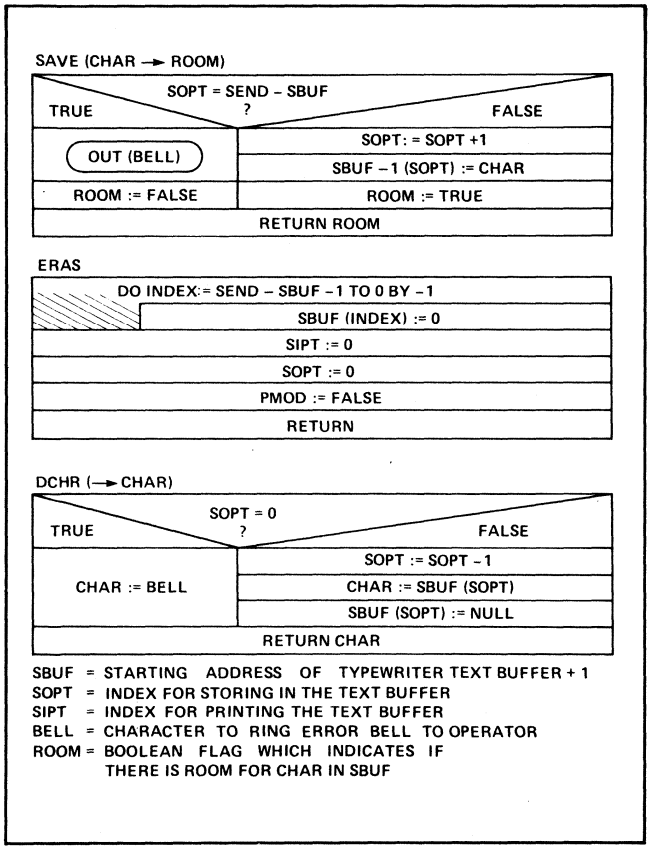


Figure B.16 Save Buffer Module

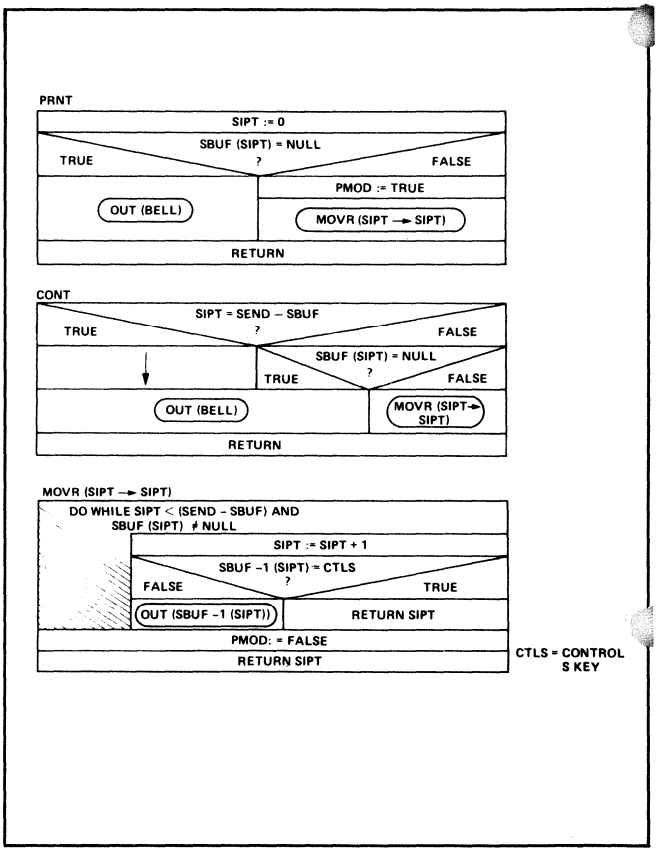


Figure B.17 Save Buffer Module (Cont.)

APPENDIX C
Intelligent Typewriter Program Listing

The ITS program listing for the general purpose interrupt driven full-duplex parallel I/O version of system, designated as ITSB3, is documented in the following. The ITS program listing for the serial I/O version discussed in Section 3.4, together with a hardware kit is available from Signetics Corporation.

```
     PIP ASSEMBLER VERSION 2 LEVEL 1 ITSB3-2650 INTELLIGENT TYPEWRITER SYSTEM PAGE    1

LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

   2                          ********************************************************************************
   3                          * INTELLIGENT TYPEWRITER SYSTEM - MODEL 8 - VERSION 3
   4                          ********************************************************************************
   5                          * PROCESSOR SYMBOLS
   6        0000              R0      EQU       0             PROCESSOR REGISTERS
   7        0001              R1      EQU       1
   8        0002              R2      EQU       2
   9        0003              R3      EQU       3
  10        0080              SENS    EQU       H'80'         PSU - SENSE
  11        0040              FLAG    EQU       H'40'         PSU - FLAG
  12        0020              II      EQU       H'20'         PSU - INTERRUPT INHIBIT
  13        0007              SP      EQU       H'07'         PSU - STACK POINTER
  14        00C0              CC      EQU       H'C0'         PSL - CONDITION CODE
  15        0020              IDC     EQU       H'20'         PSL - INTERDIGIT CARRY
  16        0010              RS      EQU       H'10'         PSL - REGISTER BANK SELECT
  17        0008              WC      EQU       H'08'         PSL - 1=WITH CARRY, 0=WITHOUT
  18        0004              OVF     EQU       H'04'         PSL - OVERFLOW
  19        0002              COM     EQU       H'02'         PSL - 1=LOGICAL COMPARE, 0=ARITHMETIC COMP
  20        0001              C       EQU       H'01'         PSL - CARRY/BORROW
  21        0000              Z       EQU       0             BRANCH CONDITIONS - ZERO
  22        0001              P       EQU       1                                 POSITIVE
  23        0002              N       EQU       2                                 NEGATIVE
  24        0000              EQ      EQU       0                                 EQUAL
  25        0001              GT      EQU       1                                 GREATER THAN
  26        0002              LT      EQU       2                                 LESS THAN
  27        0003              UN      EQU       3                                 UNCONDITIONAL
  28                          *
  29                          * I/O SYMBOLS
  30        0000              NULL    EQU       H'00'
  31        000D              CR      EQU       H'0D'         CARRIAGE RETURN
  32        000A              LF      EQU       H'0A'         LINE FEED
  33        0020              SPAC    EQU       H'20'         SPACE KEY
  34        007F              DELE    EQU       H'7F'         DELETE KEY
  35        0002              CTLB    EQU       H'02'         CONTROL+B KEYS
  36        0003              CTLC    EQU       H'03'         CONTROL+C KEYS
  37        0004              CTLD    EQU       H'04'         CONTROL+D KEYS
  38        0005              CTLE    EQU       H'05'         CONTROL+E KEYS
  39        0010              CTLP    EQU       H'10'         CONTROL+P KEYS
  40        0012              CTLR    EQU       H'12'         CONTROL+R KEYS
  41        0013              CTLS    EQU       H'13'         CONTROL+S KEYS
  42        0007              BELL    EQU       H'07'         OUTPUT CHAR TO RING BELL
  43        0001              TTCN    EQU       1             TTY CONTROL DEVICE ADDRESS
  44        0002              TTDA    EQU       2             TTY DATA DEVICE ADDRESS
  45        001F              PAR6    EQU       H'1F'         MASK TO STRIP PARITY BIT FROM 6 BIT CHAR
  46        007F              PAR8    EQU       H'7F'         MASK TO STRIP PARITY BIT FROM 8 BIT CHAR
  47                          * TTY UART STATUS BYTE - INPUT FROM DEVICE TTCN
  48                          *       1=CONDITION PRESENT, 0=CONDITION NOT PRESENT
  49        0001              CARR    EQU       H'01'         BIT 0 = CARRIER DETECT
  50        0002              RFUL    EQU       H'02'         BIT 1 = RECEIVER BUFFER FULL
  51        0004              XEMP    EQU       H'04'         BIT 2 = TRANSMITTER BUFFER EMPTY
  52        0008              CTS     EQU       H'08'         BIT 3= CLEAR TO SEND
  53                          *                               BITS 4 AND 5 ARE SPARE
```

```
.INE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

  54        0040                    PAR     EQU       H'40'       BIT 6 = PARITY ERROR
  55        0080                    OVER    EQU       H'80'       BIT 7 = OVERRUN ERROR
  56                                * TTY UART CONTROL BYTE (OUTPUT TO DEVICE TTCN TO CONFIGURE UART)
  57                                * BIT 0 = BAUD RATE: 0=110 BAUD, 1=300 BAUD
  58        0000                    BAU1    EQU       H'00'       MASK TO SELECT 110 BAUD
  59        0001                    BAU3    EQU       H'01'       MASK TO SELECT 300 BAUD
  60                                * BIT 1 = CHARACTER LENGTH (INCLUDING PARITY BIT IF ANY):
  61                                *    0=8 BITS, 1=6 BITS
  62        0000                    CHR8    EQU       H'00'       MASK TO SELECT 8 BIT LENGTH
  63        0002                    CHR6    EQU       H'02'       MASK TO SELECT 6 BIT LENGTH
  64                                * BITS 2-3 = PARITY SELECT: 0=NO PARITY, 1=EVEN PARITY, 2=UNDEFINED,
  65                                *    3=ODD PARITY
  66        0000                    POFF    EQU       H'00'       MASK TO SELECT NO PARITY
  67        0004                    PEVN    EQU       H'04'       MASK TO SELECT EVEN PARITY
  68        000C                    PODD    EQU       H'0C'       MASK TO SELECT ODD PARITY
  69                                * BIT 4 = NUMBER OF STOP BITS: 0=1 STOP BIT, 1=2 STOP BITS
  70        0000                    STP1    EQU       H'00'       MASK TO SELECT 1 STOP BIT
  71        0010                    STP2    EQU       H'10'       MASK TO SELECT 2 STOP BITS
  72                                * BIT 5 = RECEIVER BUFFER FULL-DATA OK INTERRUPT: 0=DISABLED, 1=ENABLED
  73        0020                    RCIE    EQU       H'20'       MASK TO ENABLE RECEIVER BUFFER FULL INT.
  74                                * BIT 6 = XMTR BUFFER EMPTY INTERRUPT: 0=DISABLED, 1=ENABLED
  75        0040                    XMIE    EQU       H'40'       MASKTO ENABLE XMTR BUFFER EMPTY INTERRUPT
  76                                * BIT 7 = RCVR BUFFER FULL- ERROR INTERRUPT: 0=DISABLED, 1=ENABLED
  77        0080                    REIE    EQU       H'80'       MASK TO ENABLE RECEIVER ERROR INTERRUPT
  78                                *
  79                                *
  80                                ********************************************************************
  81                                        ORG       H'300'      RAM PORTION OF PROGRAM
  82                                * VARIABLE STORAGE
  83        0300                    VSTR    RES       0           START OF VARIABLE STORAGE
  84                                * TTY KEYBOARD MODULE VARIABLES
  85        0300                    KTMP    RES       1           TEMP  REGISTER  STORAGE FOR KEYBOARD INTERR
  86        0301                    KBUF    RES       1           KEYBOARD INPUT BUFFER
  87  0302  0302  77 00              KIEX    PPSL      0           KEYBOARD INTERRUPT MUST RETURN THRU RAM
  88  0304        37                        RETE,UN               TO RESTORE STATUS
  89                                * TTY PRINTER MODULE VARIABLES
  90        0305                    PBUF    RES       10          PRINTER OUTPUT BUF (ACCESSED AS FIFO QUEU
  91        030F                    PEND    RES       0           INDICATES END OF PRINTER BUFFER
  92        030F                    POPT    RES       1           POINTER FOR FILLING OUTPUT BUFFER
  93        0310                    PIPT    RES       1           POINTER FOR EMPTYING OUTPUT BUFFER
  94        0311                    PTMP    RES       1           TEMP REGISTER STORAGE FOR PRINTER INTERRUPT
  95        0312                    PTFG    RES       1           =H'FF' IF PRINTER ACTIVE,=0  IF  INACTIVE
  96  0313  0313  77 00              PIEX    PPSL      0           PRINTER INTERRUPT MUST RETURN THRU RAM
  97  0315        37                        RETE,UN               TO RESTORE STATUS
  98                                * SAVE BUFFER MODULE VARIABLES
  99        0316                    SBUF    RES       120         SAVE BUFFER FOR LATER PRINTING
 100        038E                    SEND    RES       0           END OF SAVE BUFFER
 101        038E                    SOPT    RES       1           POINTER FOR FILLING SAVE BUFFER
 102        038F                    SIPT    RES       1           POINTER FOR PRINTING SAVE BUFFER
 103                                * GLOBAL VARIABLES
 104        0390                    PMOD    RES       1           PRINT MODE    SWITCH=0 IF OFF,=H'FF' IF ON
 105        0391                    VEND    RES       0           END OF VARIABLE STORAGE AREA
```



```
.INE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

 107                                *       ORG       0           PROM/ROM PORTION OF PROGRAM
 108                                *
 109                                * INTERRUPT VECTOR (USED FOR INDIRECT BRANCH TO ROUTINE
 110  0000  0000  1F 00 09          RSET    BCTA,UN   MAIN        RESET ENTRANCE - MAIN PROGRAM
 111  0003        00 61                     ACON      KINT        KEYBOARD INTERRRUPT ROUTINE-FROM ZBSR *3
 112  0005  0005  01 0C             PTRI    ACON      PINT        PRINTER INTERRUPT ROUTINE-FROM ZBSR *5
 113  0007        00 75                     ACON      KERR        KBD ERROR INTERRUPT ROUTINE-FROM ZBSR *7
 114                                *
 115                                ********************************************************************
 116                                * MAIN PROGRAM - INTELLIGENT TYPEWRITER SYSTEM
 117                                *
 118  0009  0009  3F 00 27          MAIN    BSTA,UN   INIT        INITIALIZE TYPEWRITER
 119  000C  000C  3F 00 41          LOOP    BSTA,UN   IN          GET INPUT CHAR INTO R1
 120  000F        E5 00                     COMI,R1   NULL        IF ITS A NULL
 121  0011        18 79                     BCTR,EQ   LOOP        THEN GET NEXT CHARACTER
 122  0013        3F 01 3E                  BSTA,UN   CNTL        IF CONTROL FUNCTION - EXECUTE  FCN
 123  0016        18 74                     BCTR,Z    LOOP              AND GET NEXT CHARACTER
 124  0018        0C 03 90                  LODA,R0   PMOD        IF PRINT MODE ON
 125  001B        18 05                     BCTR,Z    LOP1        THEN DONT SAVE CHARS - ONLY ECHO THEM
 126  001D        3F 01 70                  BSTA,UN   SAVE        ELSE PLACE CHAR IN SAVE BUF IF ROOM
 127  0020        98 6A                     BCFR,Z    LOOP        IF NO ROOM, DONT ECHO CHAR
 128  0022  0022  3F 00 CE          LOP1    BSTA,UN   OUT         ELSE ECHO LITERAL CHARACTER
 129  0025        1B 65                     BCTR,UN   LOOP                AND GET NEXT CHARACTER
 130                                *
 131                                ********************************************************************
 132                                * INITIALIZE SYSTEM
 133  0027  0027  76 20             INIT    PPSU      II          INHIBIT INTERRUPTS
 134  0029        75 18                     CPSL      RS+WC       SELECT BNK0 REGS + ARITH/ROTATE W/O CARRY
 135  002B        77 02                     PPSL      COM         SELECT LOGICAL COMPARE
 136  002D        07 91                     LODI,R3   VEND-VSTR   R3=LENGTH OF VARIABLE STORAGE
 137  002F        20                        EORZ      R0
 138  0030  0030  CF 43 00          INI1    STRA,R0   VSTR,R3,-   ZERO VARIABLE STORAGE AREA
 139  0033        5B 7B                     BRNR,R3   INI1
 140  0035        3F 00 3E                  BSTA,UN   ITIN        INITIALIZE INPUT DEVICES
 141  0038        3F 00 CB                  BSTA,UN   ITOT        INITIALIZE OUTPUT DEVICES
 142  003B        74 20                     CPSU      II          ENABLE INTERRUPTS
 143  003D        17                        RETC,UN               EXIT
 144                                * END OF INITIALIZATION MODULE
```

```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

                                ********************************************************************************
                                * INPUT CONTROL MODULE
                                *     THE FOLLOWING SUBROUTINES WOULD NORMALLY SWITCH INPUT CONTROL
                                *     BETWEEN THE VARIOUS INPUT DEVICES. SINCE THERE IS ONLY ONE INPUT
                                *     DEVICE THESE ROUTINES ARE EQUATED DIRECTLY TO THE KEYBOARD ROUTIN
                                *
003E  003E  1F 00 44            ITIN   BCTA,UN   ITKY          ROUTINE TO INITIALIZE THE INPUT DEVICE
0041  0041  1F 00 56            IN     BCTA,UN   KYIN          ROUTINE TO GET AN INPUT CHARACTER
                                * END OF INPUT CONTROL MODULE
                                *
                                ********************************************************************************
                                * KEYBOARD PROCESSING MODULE
                                *
                                * CONFIGURE UART FOR 110 BAUD, ODD PARITY, 8 BIT CHARS, 2 STOP BITS,
                                *     AND ENABLE XMTR, RCVR, AND ERROR INTERRUPTS
0044  0044  04 FC               ITKY   LODI,R0   BAUI+PODD+CHR8+STP2+RCIE+XMIE+REIE
0046        D4 01                      WRTE,R0   ITCN
0048        08 09                      LODR,R0   CKIX          SET UP KBD INTERRUPT EXIT IN RAM
004A        CC 03 02                   STRA,R0   KIEX
004D        08 06                      LODR,R0   CKIX+2
004F        CC 03 04                   STRA,R0   KIEX+2
0052        17                         RETC,UN                 EXIT
0053  0053  77 00               CKIX   PPSL      0             ROM COPY OF KEYBOARD INTERRUPT EXIT
0055        37                         RETE,UN
                                *
                                *GET A KEYBOARD CHARACTER
                                *     THIS ROUTINE EMPTIES INPUT BUF FILLED BY KINT INTERRUPT ROUTINE
0056  0056  0D 03 01            KYIN   LODA,R1   KBUF          KEYBOARD CHARACTER TO R1 FROM INPUT BUFFER
0059        45 7F                      ANDI,R1   PAR8          MASK OFF PARITY BIT
005B        04 00                      LODI,R0   NULL          CLEAR SLOT IN INPUT BUFFER
005D        CC 03 01                   STRA,R0   KBUF
0060        17                         RETC,UN                 EXIT
                                *
                                * KEYBOARD INTERRUPT ROUTINE
                                *     THIS ROUTINE IS ENTERED THRU AN INDIRECT JUMP ON THE INTERRUPT JU
                                *     TABLE AT LOC 0.  THE ROUTINE FILLS THE INPUT BUFFER WHICH IS EMPT
                                *     BY KYIN
0061  0061  CC 03 00            KINT   STRA,R0   KTMP          SAVE R0
0064        13                         SPSL                    SAVE PSL
0065        CC 03 03                   STRA,R0   KIEX+1
0068        54 02                      REDE,R0   TTDA          INPUT KEYBOARD CHARACTER
006A        CC 03 01                   STRA,R0   KBUF                AND STORE IN INPUT BUFFER
006D        0C 03 00                   LODA,R0   KTMP          RESTORE R0
0070        75 FF                      CPSL      H'FF'         CLEAR PSL
0072        1F 03 02                   BCTA,UN   KIEX          GO TO RAM TO RESTORE PSL AND EXIT
                                *
                                * KEYBOARD ERROR INTERRUPT ROUTINE
                                *     THIS ROUTINE IS ENTERED FROM AND INDIRECT JUMP THRU THE INTERRUPT
                                *     JUMP TABLE AT LOC 0.WHEN AN INPUT ERROR OCCURS AND THE RCVR
                                *     BUFFER IS FULL
0075  0075  CC 03 00            KERR   STRA,R0   KTMP          SAVE R0
0078        13                         SPSL                    SAVE PSL
```



```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

0079        CC 03 03                   STRA,R0   KIEX+1
007C        77 10                      PPSL      RS            SELECT ALTERNATE REGISTER BANK
007E        54 02                      REDE,R0   TTDA          INPUT BAD CHAR TO RESET RCVR FULL
                                *                               INDICATOR TO AVOID OVERRUN ERROR
0080        54 01                      REDE,R0   TTCN          GET TTY STATUS
0082        44 C0                      ANDI,R0   OVER+PAR      LOOK AT OVERRUN AND PARITY ERROR BITS
0084        18 10                      BCTR,Z    OERR          IF UNKNOWN ERROR THEN BRANCH
0086        1A 07                      BCTR,N    OERR          IF OVERRUN ERROR, THEN BRANCH
0088        04 50                      LODI,R0   A'P'          ITS A PARITY ERROR
008A        3F 00 A3                   BSTA,UN   ERRM          OUTPUT PERR MESSAGE
008D        1B 0C                      BCTR,UN   KER1
008F  008F  04 4F               OERR   LODI,R0   A'O'          ITS AN OVERRUN ERROR
0091        3F 00 A3                   BSTA,UN   ERRM          OUTPUT OERR MESSAGE
0094        1B 05                      BCTR,UN   KER1
0096  0096  04 3F               QERR   LODI,R0   A'?'          ITS AN UNKNOWN ERROR
0098        3F 00 A3                   BSTA,UN   ERRM          OUTPUT ?ERR MESSAGE
009B  009B  0C 03 00            KER1   LODA,R0   KTMP          RESTORE R0
009E        75 FF                      CPSL      H'FF'         CLEAR PSL
00A0        1F 03 02                   BCTA,UN   KIEX          GO TO RAM TO RESTORE PSL AND EXIT
                                *
                                * THIS ROUTINE OUTPUTS THE FOLLOWING ERROR MESSAGE,
                                *     XERR
                                *     WHERE X IS AN ASCII CHAR PASSED TO THIS ROUTINE IN R0
00A3  00A3  05 0D               ERRM   LODI,R1   CR            OUTPUT CRLF
00A5        3F 00 CE                   BSTA,UN   OUT
00A8        05 0A                      LODI,R1   LF
00AA        3F 00 CE                   BSTA,UN   OUT
00AD        C1                         STRZ      R1            OUTPUT ERROR LETTER
00AE        3F 00 CE                   BSTA,UN   OUT
00B1        05 45                      LODI,R1   A'E'          OUTPUT ERR
00B3        3F 00 CE                   BSTA,UN   OUT
00B6        05 52                      LODI,R1   A'R'
00B8        3F 00 CE                   BSTA,UN   OUT
00BB        05 52                      LODI,R1   A'R'
00BD        3F 00 CE                   BSTA,UN   OUT
00C0        05 0D                      LODI,R1   CR            OUTPUT CRLF
00C2        3F 00 CE                   BSTA,UN   OUT
00C5        05 0A                      LODI,R1   LF
00C7        3F 00 CE                   BSTA,UN   OUT
00CA        17                         RETC,UN                 EXIT
                                * END OF KEYBOARD PROCESSING MODULE
```

```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

 240                                  ********************************************************************************
 241                                  * OUTPUT CONTROL MODULE
 242                                  * THESE ROUTINES WOULD NORMALLY SWITCH OUTPUT CONTROL BETWEEN THE VARIO
 243                                  *     OUTPUT DEVICES.  SINCE THERE IS ONLY ONE OUTPUT DEVICE THESE
 244                                  *     ROUTINES ARE EQUATED DIRECTLY TO THE PRINTER ROUTINES
 245                                  *
 246  00C8  00C8  1F 00 D1           ITOT   BCTA,UN   ITPR          INITIALIZE OUTPUT DEVICE
 247  00CE  00CE  1F 00 E9           OUT    BCTA,UN   POUT          WRITE A CHARACTER TO OUTPUT DEVICE
 248                                  * END OF OUTPUT CONTROL MODULE
 249                                  *
 250                                  ********************************************************************************
 251                                  * PRINTER PROCESSING MODULE
 252                                  *
 253                                  * PRINTER INITIALIZATION ROUTINE
 254  00D1  00D1  08 13               ITPR   LODR,R0   CPIX          SET UP THE PRINTER INTERRUPT EXIT IN RAM
 255  00D3        CC 03 13                   STRA,R0   PIEX
 256  00D6        08 10                      LODR,R0   CPIX+2
 257  00D8        CC 03 15                   STRA,R0   PIEX+2
 258  00DB        05 0D                      LODI,R1   CR            OUTPUT CARRIAGE RETURN AND LINE FEED
 259  00DD        3F 00 E9                   BSTA,UN   POUT          CALL POUT DIRECTLY SINCE THIS IS PRINTER
 260  00E0        05 0A                      LODI,R1   LF               PECULIAR INITIALIZATION
 261  00E2        3F 00 E9                   BSTA,UN   POUT
 262  00E5        17                         RETC,UN                 EXIT
 263  00E6  00E6  77 00               CPIX   PPSL      0            ROM COPY OF THE PRINTER INTERRUPT EXIT
 264  00E8        37                         RETE,UN
 265                                  *
 266                                  * PRINTER OUTPUT ROUTINE
 267                                  * THIS ROUTINE FILLS THE PRINTER OUTPUT QUEUE (FIFO) WHICH IS EMPTIED
 268                                  *     BY THE PRINTER INTERRUPT ROUTINE PINT
 269  00E9  00E9  0F 03 0F           POUT   LODA,R3   POPT          GET POINTER TO NEXT SLOT IN OUTPUT BUFFER
 270  00EC  00EC  0F 63 05           PLOP   LODA,R0   PBUF,R3       LOOK AT NEXT BUFFER SLOT (DONT INCR INDEX)
 271  00EF        E4 00                      COMI,R0   NULL          IF NO ROOM IN OUTPUT BUFFER
 272  00F1        98 79                      BCFR,EQ   PLOP          THEN LOOP HERE UNTIL PINT INTERR.MAKES ROO
 273  00F3        01                         LODZ      R1            GET OUTPUT CHAR FROM R1
 274  00F4        CF 23 04                   STRA,R0   PBUF-1,R3,+   STORE CHAR INTO OUTPUT BUFFER
 275  00F7        E7 0A                      COMI,R3   PEND-PBUF     IS POINTER AT END OF BUFFER
 276  00F9        98 02                      BCFR,EQ   POU1          NO,CONTINUE
 277  00FB        07 00                      LODI,R3   0             YES, SET POINTER TO TOP OF BUFFER
 278  00FD  00FD  CF 03 0F           POU1   STRA,R3   POPT          SAVE NEW POINTER
 279  0100        0C 03 12                   LODA,R0   PTFG          IF PRINTER ACTIVE
 280  0103        16                         RETC,N                  EXIT
 281  0104        04 FF                      LODI,R0   H'FF'         ELSE SET PRINTER ACTIVE FLAG
 282  0106        CC 03 12                   STRA,R0   PTFG
 283  0109        BB 85                      ZBSR      *PTRI            AND SIMULATE PRINTER INTERRUPT
 284  0108        17                         RETC,UN                 EXIT
 285                                  *
 286                                  * PRINTER INTERRUPT ROUTINE
 287                                  * THIS ROUTINE EMPITES THE PRINTER OUTPUT QUEUE (FIFO) FILLED BY POUT
 288  010C  010C  CC 03 11           PINT   STRA,R0   PTMP          SAVE R0
 289  010F        13                         SPSL                    SAVE PSL
 290  0110        CC 03 14                   STRA,R0   PIEX+1
 291  0113        77 10                      PPSL      RS            SELECT ALTERNATE REGISTER BANK
```



```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

 292  0115        0F 03 10                   LODA,R3   PIPT          GET BUFFER POINTER
 293  0118        0F 23 04                   LODA,R0   PBUF-1,R3,+ CHECK IF ALL CHARS IN BUFFER HAVE BEEN OU
 294  011B        E4 00                      COMI,R0   NULL
 295  011D        98 07                      BCFR,EQ   PIN1          IF NOT EMPTY, OUTPUT NEXT CHAR
 296  011F        05 00                      LODI,R1   0             ELSE CLEAR PRINTER ACTIVE FLAG
 297  0121        CD 03 12                   STRA,R1   PTFG
 298  0124        1B 10                      BCTR,UN   PIN3
 299  0126  0126  D4 02               PIN1   WRTE,R0   TTDA          OUTPUT CHAR TO THE PRINTER
 300  0128        04 00                      LODI,R0   NULL          CLEAR THIS ENTRY IN BUFFER
 301  012A        CF 63 04                   STRA,R0   PBUF-1,R3
 302  012D        E5 78                      COMI,R1   SEND-SBUF     IF POINTER NOT AT END OF BUFFER
 303  012F        98 02                      BCFR,EQ   PIN2          THEN BRANCH
 304  0131        07 00                      LODI,R3   0             ELSE RESET IT TO START OF BUFFER
 305  0133  0133  CF 03 10           PIN2   STRA,R3   PIPT          SAVE NEW POINTER
 306  0136  0136  0C 03 11           PIN3   LODA,R0   PTMP          RESTORE R0
 307  0139        75 FF                      CPSL      H'FF'         CLEAR PSL
 308  013B        1F 03 13                   BCTA,UN   PIEX          GO TO RAM TO RESTORE PSL AND EXIT
 309                                  * END OF PRINTER PROCESSING MODULE
 310                                  *
```

```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

312                                 ********************************************************************************
313                                 * INPUT CHARACTER DECODE MODULE
314                                 *
315                                 * CONTROL FUNCTION ROUTINE - CHARACTER IS IN R1 UPON ENTRY
316                                 *     IF CHARACTER NOT A CONTROL CHARACTER, RETURN CC NOT=0,ELSE CC=0
317  013E  013E  E5 7F              CNTL  COMI,R1   DELE        IF DELETE CHARACTER,
318  0140        98 08                     BCFR,EQ   CNT1
319  0142        3F 01 9A                  BSTA,UN   DCHR        THEN DELETE LAST CHAR FROM SAVE BUFFER
320  0145        3F 00 CE                  BSTA,UN   OUT         ECHO DELETED CHAR (OR BELL IF BUF EMPTY)
321  0148        1B 21                     BCTR,UN   CEXI
322  014A  014A  E5 05              CNT1  COMI,R1   CTLE        IF ERASE FUNCTION
323  014C        98 05                     BCFR,EQ   CNT2
324  014E        3F 01 88                  BSTA,UN   ERAS        THEN ERASE ENTIRE SAVE BUFFER AND RESET P1
325  0151        1B 18                     BCTR,UN   CEXI
326  0153  0153  E5 10              CNT2  COMI,R1   CTLP        IF PRINT FUNCTION
327  0155        98 05                     BCFR,EQ   CNT3
328  0157        3F 01 B1                  BSTA,UN   PRNT        THEN PRINT SAVE BUFFER FROM THE TOP
329  015A        1B 0F                     BCTR,UN   CEXI
330  015C  015C  E5 03              CNT3  COMI,R1   CTLC        IF    CONTINUE FUNCTION
331  015E        98 05                     BCFR,EQ   CNT6
332  0160        3F 01 CC                  BSTA,UN   CONT        THEN PRINT SAVE BUFFER FROM LAST STOP
333  0163        1B 06                     BCTR,UN   CEXI
334  0165  0165  E5 12              CNT6  COMI,R1   CTLR        IF RESET FUNCTION
335  0167        98 04                     BCFR,EQ   CNT7
336  0169        9B 00                     ZBRR      RSET        EXECUTE SOFTWARE RESET
337  016B  016B  20                  CEXI  EORZ      R0          SET CC=0 TO INDICATE CHAR WAS A CNTL FCN
338  016C        17                        RETC,UN
339  016D  016D  04 FF              CNT7  LODI,R0   H'FF'       SET CC NOT= 0 TO INDICATE NOT A CNTL CHAR
340  016F        17                        RETC,UN
341                                 * END OF INPUT CHARACTER DECODE MODULE
342                                 ********************************************************************************
```



```
INE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

344                                 ********************************************************************************
345                                 * SAVE BUFFER MODULE
346                                 *
347                                 * SAVE BUFFER ROUTINE
348                                 *     THIS ROUTINE SAVES LITERAL CHARACTERS AND OUTPUT CONTROL
349                                 *     FUNCTIONS FOR LATER PRINTING WHEN THE PRINT COMMAND IS INPUT.
350                                 *     ON EXIT CC=0 IF ROOM FOR CHAR IN BUFFER, CC NOT=0 IF BUFF FULL
351  0170  0170  0F 03 8E           SAVE   LODA,R3   SOPT        GET SAVE BUFFER POINTER
352  0173        E7 78                  COMI,R3   SEND-SBUF   IF BUFFER FULL
353  0175        18 09                  BCTR,EQ   SFUL        THEN BRANCH
354  0177        01                     LODZ      R1          PLACE CHAR FROM R1 INTO SAVE BUFFER
355  0178        CF 23 15               STRA,R0   SBUF-1,R3,+
356  017B        CF 03 8E               STRA,R3   SOPT        SAVE NEW BUFFER POINTER
357  017E        20                     EORZ      R0          CC=0
358  017F        17                     RETC,UN
359  0180  0180  05 07           SFUL   LODI,R1   BELL        RING BELL - BUFFER IS FULL
360  0182        3F 00 CE               BSTA,UN   OUT
361  0185        04 FF                  LODI,R0   H'FF'       CC NOT= 0 TO SHOW BUFFER FULL
362  0187        17                     RETC,UN   EXIT
363                                 *
364                                 * ERASE ROUTINE
365                                 *     THIS ROUTINE ERASES THE ENTIRE SAVE BUFFER AND RESETS BOTH POIN
366  0188  0188  07 78           ERAS   LODI,R3   SEND-SBUF   ZERO SAVE BUFFER
367  018A        20                     EORZ      R0
368  018B  018B  CF 43 16         ERA1   STRA,R0   SBUF,R3,-
369  018E        5B 7B                  BRNR,R3   ERA1
370  0190        CF 03 8F               STRA,R3   SIPT        ZERO SAVE BUFFER PRINT POINTER
371  0193        CF 03 8E               STRA,R3   SOPT        ZERO SAVE BUFFER LOAD BUFFER
372  0196        CC 03 90           STRA,R0   PMOD            TURN OFF PRINT MODE SWITCH
373  0199        17                     RETC,UN
374                                 *
375                                 *DELETE CHARACTER ROUTINE
376                                 *     THIS ROUTINE DELETES THE LAST CHARACTER IN THE SAVE BUFFER
377                                 *     AND RETURNS THE CHARACTER IN R1.  IF THE BUFFER IS EMPTY,
378                                 *     IT RETURNS WITH THE BELL CODE IN R1.
379  019A  019A  0F 03 8E           DCHR   LODA,R3   SOPT        GET BUFFER POINTER
 80  019D        E7 00                  COMI,R3   0           IF BUFFER EMPTY
 81  019F        18 0D                  BCTR,EQ   DCH1        THEN BRANCH
382  01A1        0F 43 16               LODA,R0   SBUF,R3,-   ELSE GET LAST CHAR AND DECREMENT POINTER
383  01A4        C1                     STRZ      R1          CHAR TO R1
384  01A5        04 00                  LODI,R0   NULL        AND REPLACE IT WITH A NULL
385  01A7        CF 63 16               STRA,R0   SBUF,R3
386  01AA        CF 03 8E               STRA,R3   SOPT        SAVE NEW POINTER
387  01AD        17                     RETC,UN
388  01AE  01AE  05 07           DCH1   LODI,R1   BELL        OUTPUT BELL FOR IMPROPER REQUEST
389  01B0        17                     RETC,UN
390                                 *
391                                 * PRINT SAVE BUFFER ROUTINE
392                                 *     THIS ROUTINE PRINTS THE SAVE BUFFER STARTING AT THE BEGINNING
393                                 *     UNTIL THE END OF DATA OR A STOP CODE IS REACHED
394  01B1  01B1  07 00           PRNT   LODI,R3   0           SET POINTER TO START OIF BUFFER
395  01B3        0F 63 16               LODA,R0   SBUF,R3     GET NEXT PRINT CHAR BUT DONT INCR INDEX
```

```
LINE  ADDR  LABL  B1 B2 B3 B4 ERROR SOURCE

 396  01B6        E4 00                    COMI,R0   NULL         IF BUFFER EMPTY
 397  01B8        18 0C                    BCTR,EQ   PBEL         THEN BRANCH
 398  01BA        04 FF                    LODI,R0   H'FF'        TURN ON PRINT MODE SWITCH
 399  01BC        CC 03 90                 STRA,R0   PMOD
 400  01BF        3F 01 E9                 BSTA,UN   MOVR         MOVE CHARACTERS TO OUTPUT BUFFER
 401  01C2        CF 03 8F                 STRA,R3   SIPT         SAVE NEW PRINT POINTER
 402  01C5        17                       RETC,UN
 403  01C6  01C6  05 07            PBEL    LODI,R1   BELL         OUTPUT BELL FOR IMPROPER REQUEST
 404  01C8        3F 00 CE                 BSTA,UN   OUT
 405  01CB        17                       RETC,UN
 406                                  *
 407                                  * CONTINUE PRINTING ROUTINE
 408                                  *      THIS ROUTINE CONTINUES PRINTING THE SAVE BUFFER FROM WHERE THE
 409                                  *      THE LAST PRINT OR CONTINUE REQUEST LEFT OFF, UNTIL THE END OF
 410                                  *      DATA OR A STOP CODE IS REACHED.
 411  01CC  01CC  0F 03 8F            CONT    LODA,R3   SIPT         GET PRINT POINTER
 412  01CF        E7 00                    COMI,R3   0            IF SAVE BUFFER EMPTY
 413  01D1        18 10                    BCTR,EQ   CBEL         THEN BRANCH
 414  01D3        E7 78                    COMI,R3   SEND-SBUF    IF POINTER AT END OF BUFFER
 415  01D5        18 0C                    BCTR,EQ   CBEL         THEN BRANCH
 416  01D7        0F 63 16                 LODA,R0   SBUF,R3      GET NEXT PRINT CHAT BUT DONT INCR INDEX
 417  01DA        18 07                    BCTR,EQ   CBEL         IF ITS A NULL (END OF DATA) BRANCH
 418  01DC        3F 01 E9                 BSTA,UN   MOVR         MOVE CHARACTERS TO OUTPUT BUFFER
 419  01DF        CF 03 8F                 STRA,R3   SIPT         SAVE NEW PRINT POINTER
 420  01E2        17                       RETC,UN
 421  01E3  01E3  05 07            CBEL    LODI,R1   BELL         OUTPUT BELL FOR IMPROPER REQUEST
 422  01E5        3F 00 CE                 BSTA,UN   OUT
 423  01E8        17                       RETC,UN
 424                                  *
 425                                  * MOVER ROUTINE
 426                                  *      THIS ROUTINE TRANSFERS CHARACTERS FROM THE SAVE BUFFER TO THE
 427                                  *      OUTPUT BUFFER.  IT EXPECTS R3 TO BE SET UP AS THE STARTING INDEX
 428                                  *      TO THE SAVE BUFFER. TRANSFERS WILL STOP WHEN A STOP CODE OR THE
 429                                  *      END OF DATA IS REACHED
 430  01E9  01E9  E7 78            MOVR    COMI,R3   SEND-SBUF    IF END OF BUFFER REACHED
 431  01EB        18 12                    BCTR,EQ   MOV2         THEN EXIT
 432  01ED        0F 23 15                 LODA,R0   SBUF-1,R3,+  GET NEXT CHAR FROM SAVE BUFFER
 433  01F0        E4 00                    COMI,R0   NULL         IF END OF DATA REACHED
 434  01F2        18 09                    BCTR,EQ   MOV1         THEN EXIT
 435  01F4        E4 13                    COMI,R0   CTLS         IF STOP CODE IN BUFFER
 436  01F6        14                       RETC,EQ                THEN EXIT
 437  01F7        C1                       STRZ      R1           CHAR TO R1
 438  01F8        3F 00 CE                 BSTA,UN   OUT          AND OUTPUT IT
 439  01FB        1B 6C                    BCTR,UN   MOVR         CONTINUE MOVES
 440  01FD  01FD  A7 01            MOV1    SUBI,R3   1            MOVE INDEX BACK TO NULL CHARACTER
 441  01FF  01FF  20               MOV2    EORZ      R0           TURN OFF PRINT MODE SWITCH
 442  0200        CC 03 90                 STRA,R0   PMOD
 443  0203        17                       RETC,UN
 444                                  *
 445                                  * END SAVE BUFFER MODULE
 446                                  ********************************************************************************
 447                                          END       MAIN


TOTAL ASSEMBLER ERRORS =     0
```

Electronic components and materials

for professional, industrial and consumer uses

Argentina: FAPESA I.y.C., Av. Crovara 2550, Tablada, Prov. de BUENOS AIRES, Tel. 652-7438/7478.
Australia: PHILIPS INDUSTRIES HOLDINGS LTD., Elcoma Division, 67 Mars Road, LANE COVE, 2066, N.S.W., Tel. 42 1261.
Austria: ÖSTERREICHISCHE PHILIPS BAUELEMENTE Industrie G.m.b.H., Zieglergasse 6, A-1072 WIEN, Tel. 93 26 22.
Belgium: M.B.L.E., 80, rue des Deux Gares, B-1070 BRUXELLES, Tel 523 00 00.
Brazil: IBRAPE S.A., Caixa Postal 7383, Av. Paulista 2073-S/Loja, SAO PAULO, SP, Tel. 287-7144.
Canada: PHILIPS ELECTRONICS INDUSTRIES LTD., Electron Devices Div., 116 Vanderhoof Ave., TORONTO 17, Ontario, Tel. 425-5161.
Chile: PHILIPS CHILENA S.A., Av. Santa Maria 0760, SANTIAGO, Tel. 39-40 01.
Colombia: SADAPE S.A., P.O. Box 9805 Calle 13, No. 51 + 39, BOGOTA D.E. 1., Tel. 600 600.
Denmark: MINIWATT A/S, Emdrupvej 115A, DK-2400 KØBENHAVN NV., Tel. (01) 69 16 22.
Finland: OY PHILIPS AB, Elcoma Division, Kaivokatu 8, SF-00100 HELSINKI 10, Tel. 1 72 71.
France: R.T.C. LA RADIOTECHNIQUE-COMPELEC, 130 Avenue Ledru Rollin, F-75540 PARIS 11, Tel. 355-44-99.
Germany: VALVO, UB Bauelemente der Philips G.m.b.H., Valvo Haus, Burchardstrasse 19, D-2 HAMBURG 1, Tel. (040) 3296-1.
Greece: PHILIPS S.A. HELLENIQUE, Elcoma Division, 52, Av. Syngrou, ATHENS, Tel. 915 311.
Hong Kong: PHILIPS HONG KONG LTD., Components Dept., 11th Fl., Din Wai Ind. Bldg., 49 Hoi Yuen Rd, KWUNTONG, Tel. K-42 72 32.
India: PHILIPS INDIA LTD., Elcoma Div., Band Box House, 254-D, Dr. Annie Besant Rd., Prabhadevi, BOMBAY-25-DD, Tel. 457 311-5.
Indonesia: P.T. PHILIPS-RALIN ELECTRONICS, Elcoma Division, 'Timah' Building, Jl. Jen. Gatot Subroto, JAKARTA, Tel. 44 163.
Ireland: PHILIPS ELECTRICAL (IRELAND) LTD., Newstead, Clonskeagh, DUBLIN 14, Tel. 69 33 55.
Italy: PHILIPS S.P.A., Sezione Elcoma, Piazza IV Novembre 3, I-20124 MILANO, Tel. 2-6994.
Japan: NIHON PHILIPS CORP., 32nd Fl., World Trade Center Bldg., 5, 3-chome, Shiba Hamamatsu-cho, Minato-ku, TOKYO, Tel. 03-435-5268.
(IC Products) SIGNETICS JAPAN, LTD., TOKYO, Tel. (03) 230-1521.
Korea: PHILIPS ELECTRONICS (KOREA) LTD., Philips House, 260-199 Itaewon-dong, Yongsan-ku, C.P.O. Box 3680, SEOUL, Tel. 44-4202.
Mexico: ELECTRONICA S.A. de C.V., Varsovia No. 36, MEXICO 6, D.F., Tel. 5-33-11-80.
Netherlands: PHILIPS NEDERLAND B.V., Afd. Elonco, Boschdijk 525, NL-4510 EINDHOVEN, Tel. (040) 79 33 33.
New Zealand: EDAC LTD., 70-72 Kingsford Smith Street, WELLINGTON, Tel. 873 159.
Norway: ELECTRONICA A/S., Vitaminveien 11, P.O. Box 29, Grefsen, OSLO 4, Tel. (02) 15 05 90.
Peru: CADESA, Jr. Ilo, No. 216, Apartado 10132, LIMA, Tel. 27 73 17.
Philippines: ELDAC, Philips Industrial Dev. Inc., 2246 Pasong Tamo, MAKATI-RIZAL, Tel. 86-89-51 to 59.
Portugal PHILIPS PORTUGESA S.A.R.L., Av. Eng. Duharte Pacheco 6, LISBOA 1, Tel. 68 31 21.
Singapore: PHILIPS SINGAPORE PTE LTD., Elcoma Div., POB 340, Toa Payoh CPO, Lorong 1, Toa Payoh, SINGAPORE 12, Tel. 53 88 11.
South Africa: EDAC (Pty.) Ltd., South Park Lane, New Doornfontein, JOHANNESBURG, Tel. 24/6701-2.
Spain: COPRESA S.A., Balmes 22, BARCELONA 7, Tel. 301 63 12.
Sweden: A.B. ELCOMA, Lidingövägen 50, S-10 250 STOCKHOLM 27, Tel. 08/67 97 80.
Switzerland: PHILIPS A.G., Elcoma Dept., Edenstrasse 20, CH-8027 ZÜRICH, Tel. 01/44 22 11.
Taiwan: PHILIPS TAIWAN LTD., 3rd Fl., San Min Building, 57-1, Chung Shan N. Rd, Section 2, P.O. Box 22978, TAIPEI, Tel. 5513101-5.
Turkey: TÜRK PHILIPS TICARET A.S., EMET Department, Gümüssuyu Cad. 78-80, Beyoglü, ISTANBUL, Tel. 45 32 50.
United Kingdom: MULLARD LTD., Mullard House, Torrington Place, LONDON WC1E 7HD, Tel. 01-580 6633.
United States: (Active devices & Materials) AMPEREX SALES CORP., 230, Duffy Avenue, HICKSVILLE, N.Y. 11802, Tel. (516) 931-6200.
(Passive devices) MEPCO/ELECTRA INC., Columbia Rd., MORRISTOWN, N.J. 07960, Tel. (201) 539-2000.
(IC Products) SIGNETICS CORPORATION, 811 East Arques Avenue, SUNNYVALE, California 94086, Tel. (408) 739-7700.
Uruguay: LUZILECTRON S.A., Rondeau 1567, piso 5, MONTEVIDEO, Tel. 9 43 21.
Venezuela: IND. VENEZOLANAS PHILIPS S.A., Elcoma Dept., A. Ppal de los Ruices, Edif. Centro Colgate, Apdo 1167, CARACAS, Tel. 36 05 11.

Printed in The Netherlands 7-76 9399 509 54861