TEXAS INSTRUMENTS

TM 990

Introduction to Microprocessors

Hardware and Software

Introduction to Microprocessors

MICROPROCESSOR SERIES™

October 1979

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time in order to improve design and to supply the best product possible.

TI cannot assume any responsibility for any circuits shown or represent that they are free from patent infringement.

INTRODUCTION TO MICROPROCESSORS HARDWARE AND SOFTWARE

Learning with the TM 990/189 University Board

Prepared by

GEORGE GOODE & ASSOCIATES, DALLAS, TEXAS

First Edition

Third Printing
October, 1979

P R E F A C E

This book was written primarily to satisfy the textbook requirement at universities and colleges for a three credit-hour course introducing students to microcomputers and to assembly language programming. It complements Texas Instruments' TM 990/189 (University Board), a single-board microcomputer with an on-board terminal developed to facilitate microprocessor training and experiments with on- and off-board peripherals. The board utilizes the TMS 9980A 16-bit microprocessor and associated support chips.

Readers of this text are assumed to possess only a minimal background in digital logic and computer operation. However, other things being equal, a strong motivation to "learn-by-doing" is doubtless the key trait required for success in mastering the topics covered in the text. Working professionals should find this text and the University Board to be a convenient means of "bootstrapping" into the new field of programmed logic by means of self-paced, free-time study with many exciting new vistas along the way.

For formal classroom instruction, teachers will find the contents of the book suitable for the typical one-semester course requirement of three hours of lecture per week with one to two hours of supervised or self-paced lab, the latter requiring use of the TM 990/189 Microcomputer Board. The bulk of the lecture material can be covered in about 12 weeks, thereby allowing time for selected advanced topics, semester microcomputer projects, or more time for introductory material (if needed). One of the key features of the book is the use of many program examples which illustrate the instructions, programming techniques and I/O interfacing. A number of graduated exercises and lab experiments are provided at the end of most of the chapters to test the student's progress. Answers are provided in Appendix B for the odd-numbered exercises.

An effort has been made to present the material in an orderly, progressive manner which has been based, for the most part, on experience in teaching courses on microprocessors to hundreds of students in industry and at the university level.

Beginning with Chapter 1, the reader is introduced to computer architecture, computer operation, and some of the recent architectual enhancements embodied in microprocessors. The chapter closes with a description of the unique memory-to-memory architecture of the TMS 9980A microprocessor and the features of the University Board. Computer arithmetic and logic are reviewed in Chapter 2, with the emphasis on the operation of the arithmetic logic unit (ALU). Additionally, the firmware monitor (UNIBUG) commands are described

and a number of experiments are given to exercise the on-board terminal and to verify arithmetic and logic operations.

The key topic of computer addressing is introduced in Chapter 3 together with the first subset of TMS 9980A instructions. One feature of the book is the way in which the instructions are described, including examples and figures showing the configuration of the ALU to achieve arithmetic, logic, compare, and jump operations. Flowcharting and machine coding are presented in working out the first program example.

An overview of assembler functions is provided in Chapter 4 followed by details of the University Board symbolic assembler. Directives, labels, and instruction syntax are discussed prior to coverage of additional instructions and a sort algorithm program example.

Memory systems for microcomputers are the focus of Chapter 5. In addition, another subset of instructions is described followed by a memory test program example. Input and output concepts are introduced and overviewed in Chapter 6 with a versatile cycle generator employed as the program example. Chapter 7 continues with input/output design details including interfacing and special peripheral component chip operation. A traffic-light controller is used as the program example.

Modular programming, the powerful context switch, and user accessible utilities resident in the UNIBUG firmware are discussed in Chapter 8. A Morse code translator is developed as the program example. Chapter 9 continues with the more advanced software engineering concepts including hardware/software tradeoffs, top-down design, linking, interrupt servicing, and real-time considerations. Appropriately, the program example is a time-of-day clock.

Practical aspects of microprocessor/microcomputer product development are discussed in the final chapter. System design and development, debugging, testing, and delivery are topics presented from a realistic perspective.

As a historical note, the work of George Goode & Associates with the TMS 9900 family of microprocessors began in 1976 with in-plant seminars attended by engineers, programmers, managers, and engineering technicians. Later on, much of the material was adapted for a course in microprocessors taught at Southern Methodist University, School of Engineering. The many questions raised by the students and the subsequent discussions from over fifty industrial seminars and a half dozen college-level courses on microprocessors has provided the insight to continuously revise the course material to improve its effectiveness toward learning. Additionally, the interest of the students in the course material pointed out the need for publishing it. Thus, to these many hundreds of students, we are indebted.

C O N T E N T S

APPENDIXES

CHAPTER 1

OVERVIEW OF COMPUTERS, MICROPROCESSORS, AND MICROCOMPUTERS

1.1 INTRODUCTION

This is a book about a particular microcomputer--the TM 990/189M University Board manufactured by Texas Instruments Incorporated. Before going into details of this board, however, it is appropriate to survey a number of topics relating to computers to explain to the reader (especially the novice) certain fundamental concepts and definitions that are essential for an understanding of the material in succeeding chapters.

Chapter 1 is an introduction to computers, microprocessors, and microcomputers. An endeavor has been made to cover a wide range of subjects in overview with details brought out in chapters that follow. In many cases mention is made that "This will be brought out in detail in Chapter ____...," and in each case the reader can refer to this detail immediately if he wishes, or, he can proceed toward it in progression, whichever he chooses. However, this book is intended to be used as progressive teaching material, and the student can best benefit from it by following the orderly exposition of the subjects as they are offered.

What is a computer? This is a question that needs to be answered. First of all, it is a tool, a machine, a thing that assists in the performance of a human task. It is designed to process and manipulate data.

If a task is to be performed by a computer, there are at least three vital elements which must be involved:

- First is data. This can be in the form of numbers, letters of the alphabet, or symbols of any type which describe machine status, or virtually any information that needs to be manipulated and processed.
- Second, there must be a program. This consists of a sequence of instructions carefully developed one after the other in such a way that when executed in proper order the task will be completed.
- Third is the need for peripheral equipment. At a minimum this is some type of device outside of the computer that provides data to or receives data from the computer. It could be a printer, a display device, a tape reader, a keyboard, or it could be some external machine or instrument...or any combination of these devices.

The desired result of computation or processing is the completion of a desired task or function with a minimum of human intervention. Commmonly such a task could be the performance of some mathematical calculation, handling of a payroll, controlling a manufacturing process, or any of a myriad of tasks that the typical business or scientific laboratory might require. More recently, in view of the advent and availability of minicomputers and microprocessors, one of the fastest growing applications of computers is that of control--control of virtually any machine or instrument. It could be control of a production line, a petrochemical process, or an automated metal-drilling operation. With the new and inexpensive LSI (Large Scale Integration) microprocessors and microcomputers, computer control of even ordinary kitchen appliances such as microwave ovens and mixers is now common. Games, all types of instruments, machine tools, vehicle engines, and home heating and cooling can be programmed and controlled easily and inexpensively.

Nature of Data

Data for computer use is broken into two categories: digital and analog. Digital means decimal or binary numbers, or digits of any radix. One should also note that on-off signals, switch settings, etc., fit into this category of digital data. In general, any symbol that represents one of a finite set of discrete symbols (such as letters in the alphabet) is an example of digital data. For use with computers, digital means expressing such data with a pattern of ones and zeroes.

Most of the data or information in the world we live in could be categorized as analog. It is called analog because it has an infinite number of possible values which present a numerical analogy of motion, position, state of being, etc. An example is the exact position of a shaft that controls, say, a valve. Other examples are a meter reading of varying voltage, or a voice-signal voltage on a telephone line. These all represent analogs, which have maximum and minimum values (limits); however, the number of possible positions or readings between the limits is infinite. Fortunately, in the real world, analog data can be converted (digitized) to a finite set of values or numbers. As long as the analog-to-digital conversion results in adequate resolution of the analog data for the task to be performed, such a conversion permits use of digital circuits and computers for processing and manipulating analog data. Although analog data can be manipulated and processed with analog circuits, for an expanding number of applications a less expensive and more versatile method is to convert it to digital data and use microprocessors to process it.

Role of the Programmer

For the moment, assume there is a new and different task that needs to be handled by a digital computer. Typically, what does it take to create a program to handle it? First, the new task or function must be analyzed and partitioned into various known subtasks. Further, each of these subtasks must be broken down into a logical sequence of steps. This latter process is facilitated by constructing a flowchart. Then for each step to be accomplished, the programmer writes down a list of instructions, each executed in proper sequence for performance of all of the desired subtasks and the overall task itself. This latter process is called *source coding* or simply *coding*.

Next, the programmer must translate each of the instructions into code understood by the computer--generally called machine code. Such a process is called assembling. This can be done manually or it can be handled by a computer using what is called a assembler.

To test the machine-code program, it must be loaded into the computer upon which it is designed to run along with the data to be manipulated. And more often than not (unless a program is simple and short) it will probably not run the first time. Or, it may run partially correct. Thus, a *debug* process comes next. The programmer endeavors to locate the point (i.e., instruction) in the program where a "bug" (incorrect instruction or instructions) exists. Once a programmer thinks he has found a bug, he makes changes in the program. That is, he deletes, adds, or modifies instructions until the program appears to run satisfactorily. Note that the word "appears" is used since the program may still have an additional bug or bugs which have not been located due to failure of the programmer to test all aspects of the program's operation. In fact, some programs are so large and complex that they are never completely debugged. Bugs are sometimes discovered by users months and even years later.

An overview of the functions of a computer and the programmer have thus been explained. So far, it has been stressed that the computer must have data, must have a program to control it, and must be connected to peripheral equipment.

With these basic concepts in mind, it is now appropriate to focus on the computer itself and how it is constructed internally.

1.2 BASIC COMPUTER ARCHITECTURE

This section deals with the building blocks of the computer or, for that matter, any digital system. Once one learns these basic blocks and their functions, then the exact manner in which they are interconnnected constitutes a principal factor in the uniqueness of a specific computer, whether main frame, mini, or micro. The approach taken is to review these blocks then consider a configuration that

can be called the "classic" stored-program digital computer. A simple hypothetical program will be tried out on this computer to ensure that the reader has a basic understanding of the function of each of the building blocks. It is then logical to proceed to explore many of the newer architectural enhancements (improvements) that are prevalent in micros and minis today.

It should be stressed at this point that an understanding of a computer's architecture is essential if optimum use is to be made of its machine- and assembly-language instructions.

Principal Building Blocks

Starting with the high-level block diagram of a computer, Figure 1-1 depicts the three major blocks in a stored-program digital computer:

- CPU (central processing unit)
- Memory
- I/O (input/output).

In addition to the three major blocks, there are data and control paths in the interconnections of these blocks. In some systems, the I/O is split into its two separate blocks--input and output. However, it is common to combine input and output into a single block as shown.

In general, the CPU is the portion of the computer where instructions are decoded and then executed. Data is processed in the CPU where arithmetic or logical types of operations are required in the execution of a program.

The memory portion contains read/write memory space for interim results or for the temporary storage of data to be put out to a peripheral device at some later time. Or, data often has to be brought in from a peripheral device and stored in memory while waiting to be processed. Also in memory is the program (sequence of instructions) required for data processing or manipulation.

The I/O portion does exactly what is indicated. It is the place in the system where external digital data is brought in, either serially (a single bit at a time), or in parallel (multiple bits simultaneously). Further, I/O is the point where data can be output to peripheral devices either serially or in parallel. In short, it is where the interface or connections are made between the computer and the external world.

Not shown in Figure 1-1 are the power supply and other essential elements which, though necessary, are not a part of the main functional configuration of the computer.

Moving now to each of the major blocks, the CPU is the locus of the brain, or intelligence, of a computer. It controls the

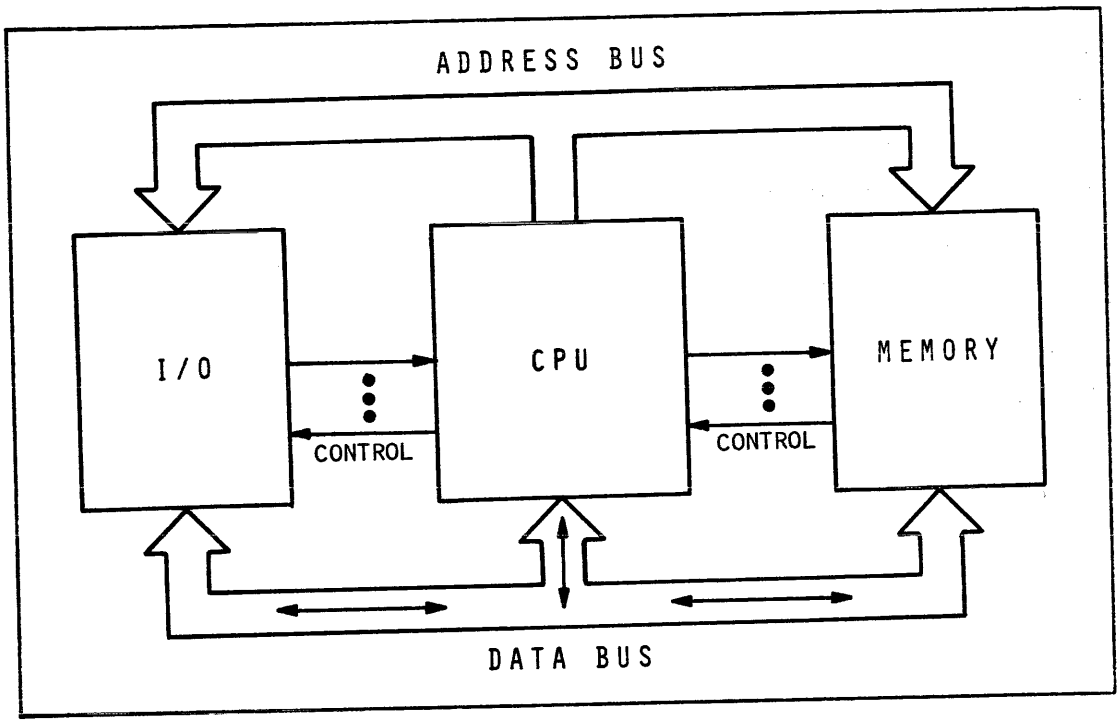


Figure 1-1. Block Diagram of a Computer

entire computer and executes each of the instructions one at a time until the entire program is completed. The following lists provide the principal functions of the CPU, the memory, and I/O sections. Afterward, further details are given on each section.

The CPU section:

- Fetches and decodes each instruction
- Performs arithmetic and logical operations as required by each instruction
- Performs address housekeeping
- Determines internal status conditions (flags) resulting from the CPU operation
- Transfers data to and from the memory and to and from the I/O sections
- Generates and receives handshake control signals to and from the memory and the I/O sections
- Provides all internal timing and control signals for the various registers and other components of the CPU itself.

The memory section:

- Stores the instructions
- Stores data from or provides data to the CPU

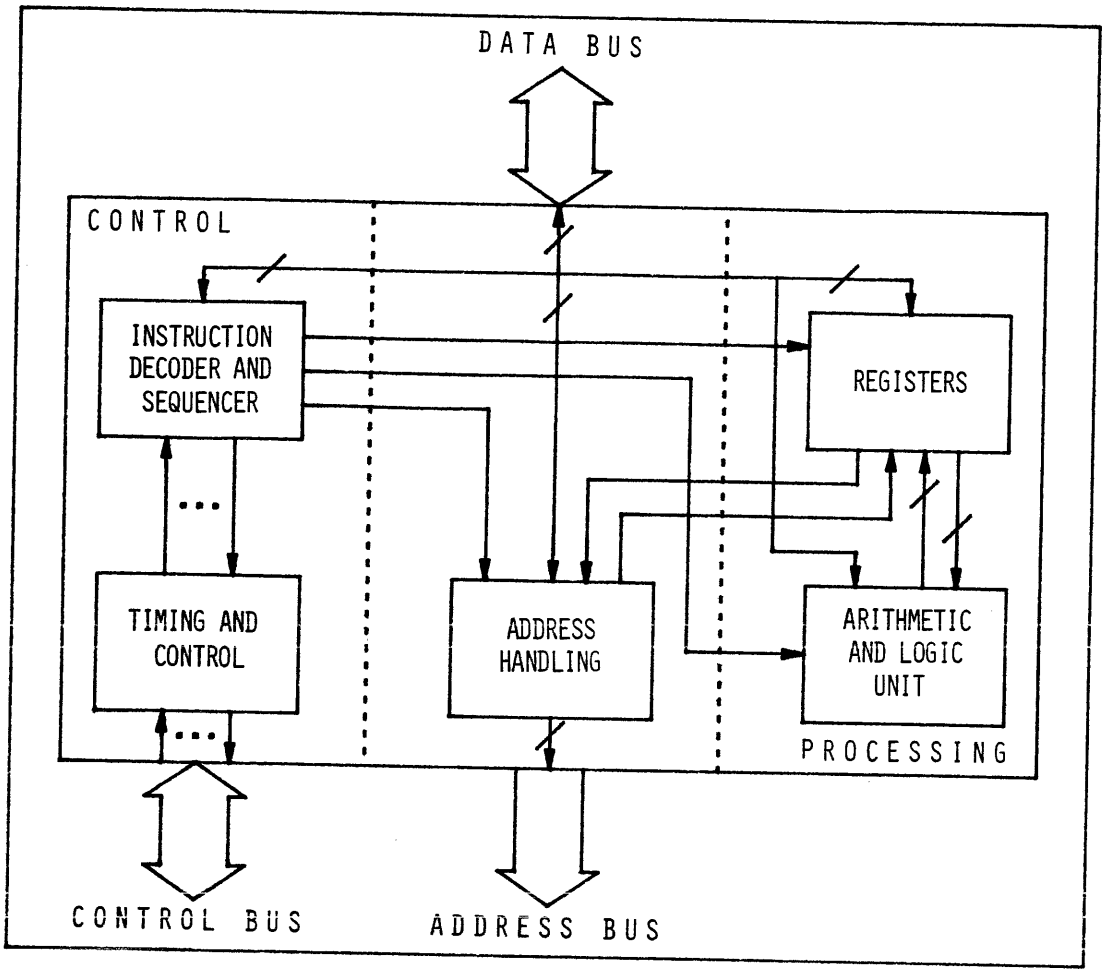


Figure 1-2. Typical CPU Structure

° Performs under direction of control signals from the CPU (or DMA controller, to be discussed later).

The I/O section:

° Provides the interface between peripheral devices and the CPU
° Transmits data and commands to peripherals from the CPU or memory
° Receives data from peripherals and passes it on to the CPU or memory
° Alerts the CPU when peripheral devices need service
° In general, performs under the direction of control signals from the CPU.

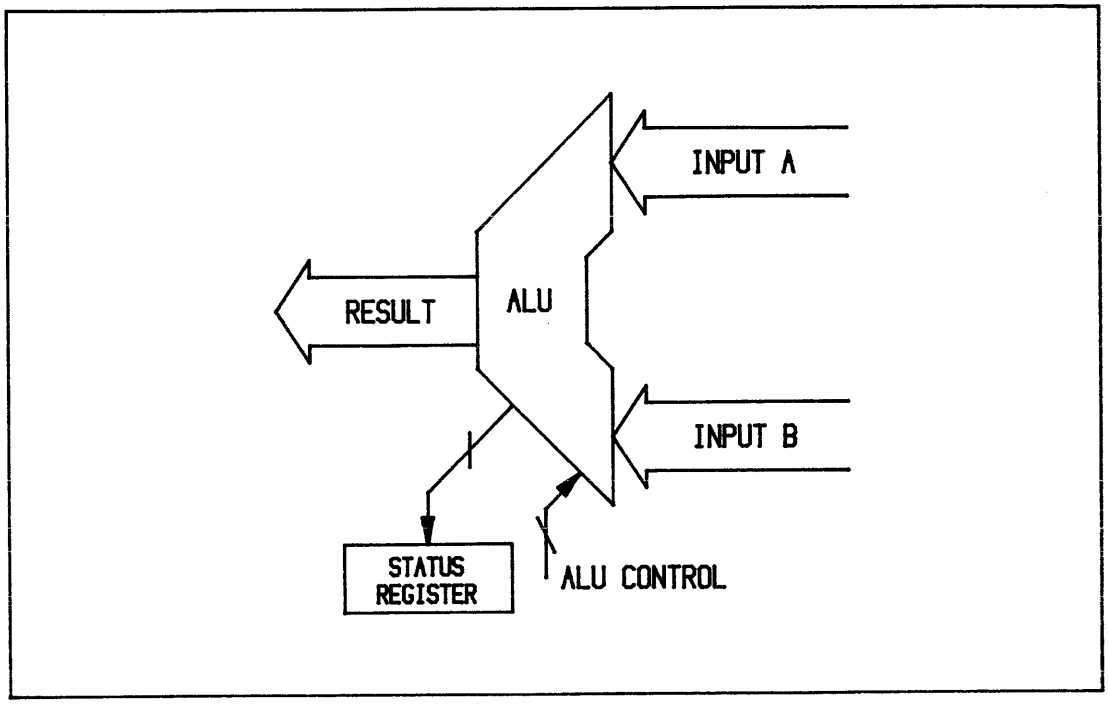


Figure 1-3. Arithmetic and Logic Unit

Components of the CPU

A detailed view of the CPU is provided in Figure 1-2. A slash mark on a line indicates two or more parallel lines. Being a very complex and critical part of the computer, the CPU and its component parts merit close attention. Consequently, each of the functional blocks in Figure 1-2 is discussed in detail.

Arithmetic and Logic Unit (ALU). Figure 1-3 is a diagram of the ALU showing two parallel inputs (A and B), one parallel output (RESULT), plus a status register. Notice that there is also a control input that selects one of a set of possible functions that the ALU can perform. Specifically, the ALU can be controlled to perform any one of the following arithmetic or logic operations.

Arithmetic Operations

- Add
- Shift/Rotate
- Compare
- Increment
- Decrement
- Negate
- Multiply
- Divide

Logic Operations

- AND
- OR
- EXCLUSIVE Or
- Complement (Invert)
- Clear
- Preset.

It should be noticed that there is no subtract function indicated. That is no problem because to subtract, the ALU merely forms the negative of the subtrahend and adds algebraically to achieve the desired result. Further details on each ALU operation is provided in subsequent chapters along with the role of the status bits resulting from these operations.

Control. As depicted in Figure 1-2, the control portion of the CPU consists of the instruction decoder and sequencer along with the timing and control circuits.

The control portion of CPU:

- Fetches the instruction from memory at the location pointed to by the program counter and moves it to the instruction register
- Decodes the instruction in the instruction register
- Executes the instruction by means of sequencing a series of control and timing signals to the other appropriate components of the CPU, the memory, and the I/O
- Performs all operations under control of the clock
- Controls all data transfers between various components, both external (memory and I/O) and internal
- In general, controls the entire computer by means of the timing and control circuits.

Address Handling. Quite often in a computer it is necessary to perform an addition on an initial address in order to provide the desired or effective address to either an instruction or a data word in memory. Consequently, computers sometimes utilize special circuits that assist in forming such addresses. Examples of this are given in detail later. However, at the present, modifying an address might consist of using, say, an index register (to be discussed later) and perhaps even using a special incrementer/adder circuit designed expressly for modifying a data pointer or a program counter. In many computers, this address handling is performed by the arithmetic logic unit in conjunction with the registers indicated in Figure 1-2.

Registers. Perhaps the variety of registers and their configurations constitutes one of the major distinguishing features of a given computer. A listing of a number of various types of registers is given below followed by some detailed information on the special function of each.

- Program counter (PC)
- Instruction register (IR)
- Memory address register (MAR)
- Memory data register (MDR)
- Accumulator or working register (ACC or WR)
- Status register (SR).

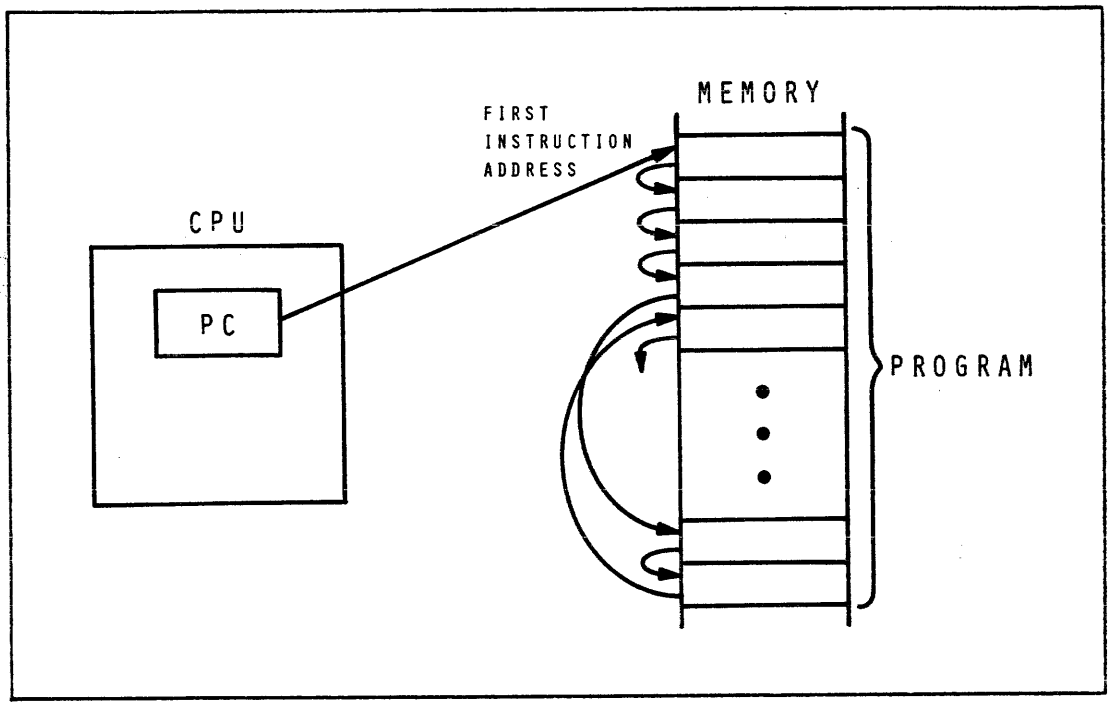


Figure 1-4. Program Counter

Program Counter (PC). This register is responsible for keeping track of the next instruction to be fetched from memory. Typically, it is 16 bits in length; however, it can be shorter, with the correspondingly smaller address space in memory. Highlights of the program-counter operation are as follows.

- It is automatically updated when each instruction is fetched so that it will be ready when the next fetch is needed (see Figure 1-4).
- Unless instructed otherwise, the CPU will keep on executing instructions sequentially.
- Transfer-of-control instructions alter the contents of the PC to cause program control to go (i.e., jump or branch) to a different point in the program.
- A special instruction may cause the CPU to stop incrementing the PC and, thus, stand in place.

Instruction Register (IR). As mentioned above, the program counter is always pointing to the next instruction to be fetched. Thus, when it is time to fetch the next instruction, the program counter contents are transferred to the memory address register along with appropriate control signals. In return, the instruction pointed to is fetched and transferred into the instruction register as shown in Figure 1-5. The IR holds the instruction while the CPU executes it. The decoding logic in the CPU analyzes the various

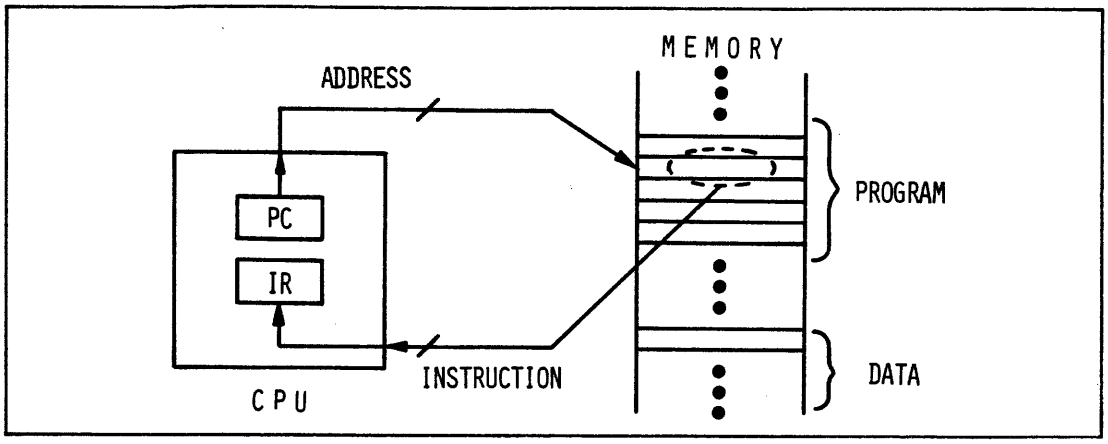


Figure 1-5. Instruction Register

fields in the instruction to determine what operation is to be performed.

Memory Address Register (MAR). This register (Figure 1-6) contains addresses for instructions, memory data, and I/O devices. The address to be placed in the memory-address register is controlled by the CPU at all times. For example, when an instruction is to be fetched, then, obviously, the contents of the program counter must be transferred to the MAR. Similarly, when an instruction references memory, the instruction will contain a memory address to be transferred to the MAR.

Memory Data Register (MDR). The principal function of the MDR is to buffer data going into and coming from the memory. It is used primarily to pass data between the central processing unit and memory, or, between the central processing unit and I/O devices. Thus, it is a bi-directional register. See Figure 1-7 for a block diagram. Sometimes a computer is designed with a set of two MDR's--one for input and one for output. Such an arrangement would not be bi-directional. In minicomputers and microcomputers, this latter type of design is becoming very rare.

Accumulator or Working Register (ACC or WR). This is a temporary working register of the CPU, accessible to the programmer. It is used in a variety of operations including the source or destination for data transfers, arithmetic and logic operations, and other special-purpose instructions. Typically the result of an ALU operation is placed in the accumulator as depicted in Figure 1-8. Generally the accumulator is connected to one of the input data ports of the ALU as shown (indicated by the A bus input). In the discussions that follow, when the term "register" is used without a qualifying adjective, it usually refers to a working register or accumulator.

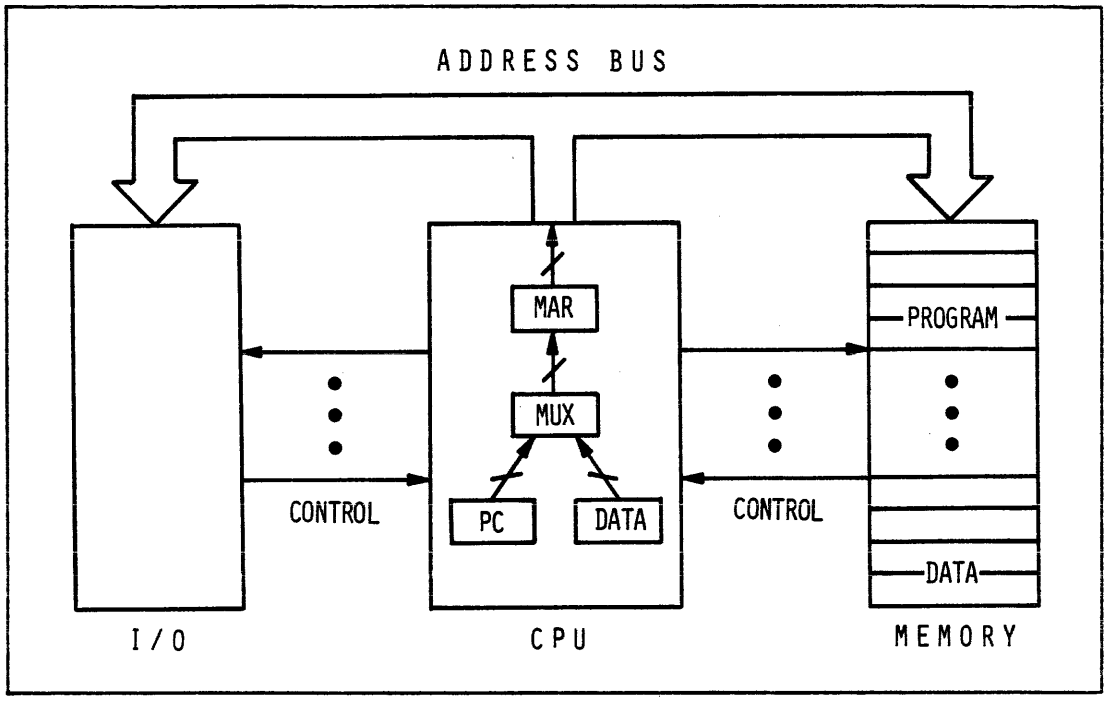


Figure 1-6. Memory Address Register

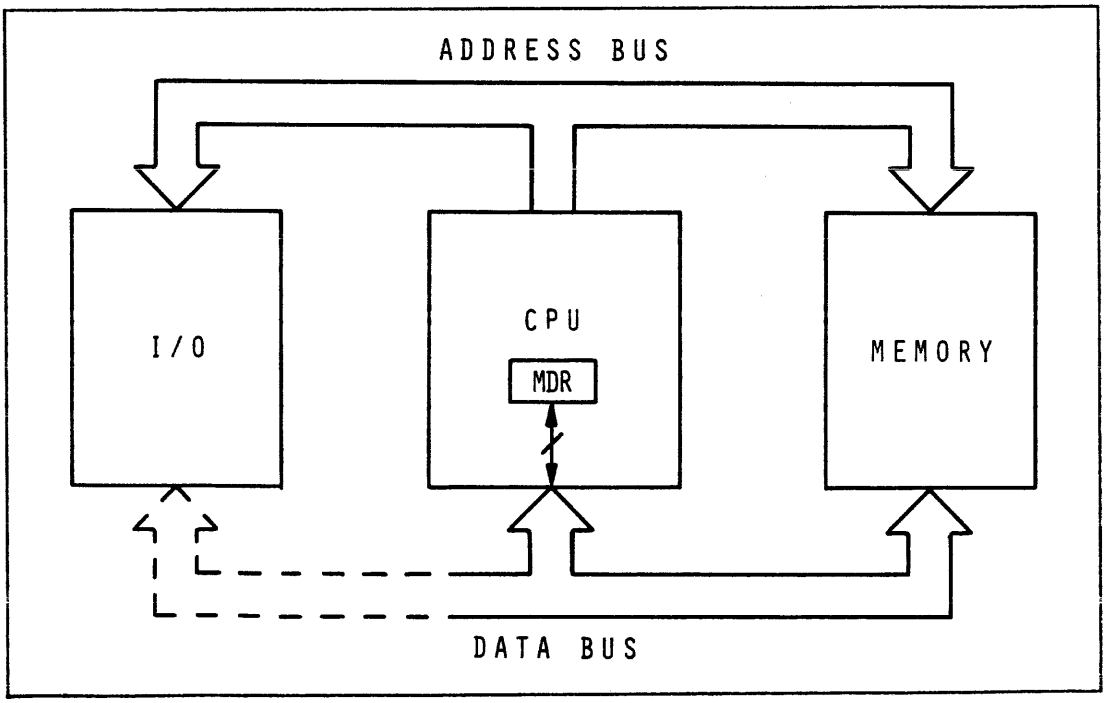


Figure 1-7. Memory Data Register

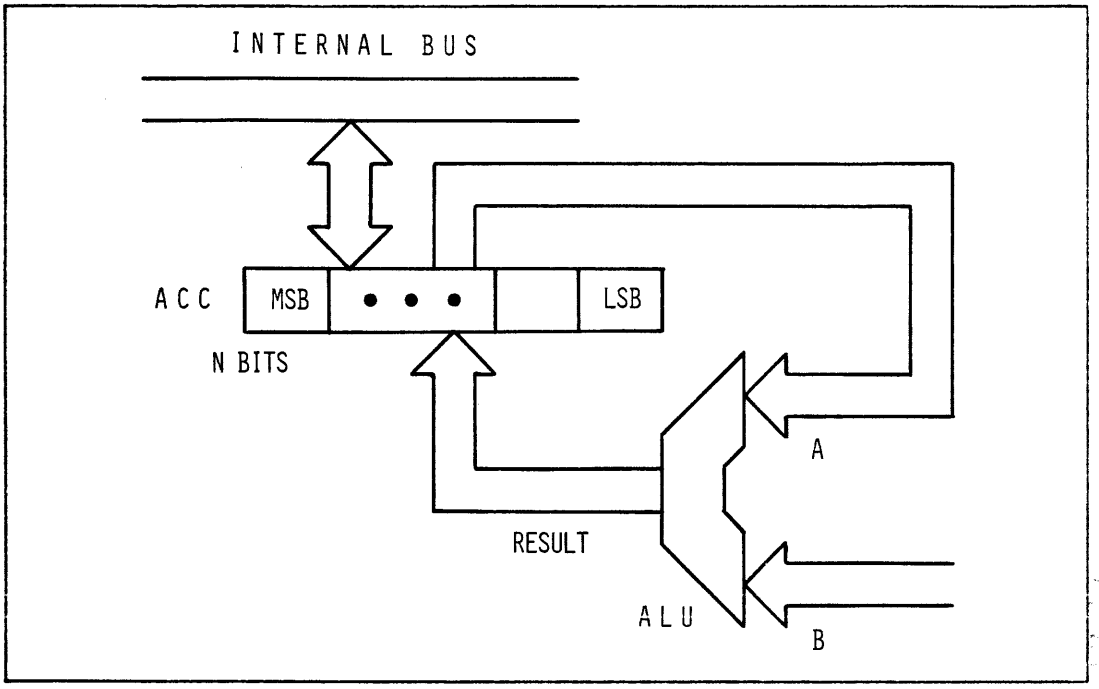


Figure 1-8. Accumulator or Working Register

In earlier CPU's, it was quite common to have only one accumulator and, as its name indicates, it would accumulate the result. However, as will be shown later, a single accumulator turns out to be a bottleneck in the execution of a program.

Status Register (SR). This register, shown in Figure 1-9, contains all of the status or flag bits that must be retained in the CPU after each arithmetic or logical operation is completed. In some machines this is called a program status word (PSW) or flag byte. Some of the more typical status flag bits are

- Arithmetic carry
- Arithmetic overflow
- Result of an operation being zero
- Result of comparing two operands: operands equal or one operand arithmetically greater (signed value), or logically greater than (unsigned value) the other
- Parity on the last result
- Sign of the last result
- Interrupt enable status.

One may well ask, "What is the purpose of the status register and each of the status bits?" The answer is that conditional jump or branch instructions make decisions by the condition (ONE or ZERO) of selected SR bits. This can cause the program to transfer

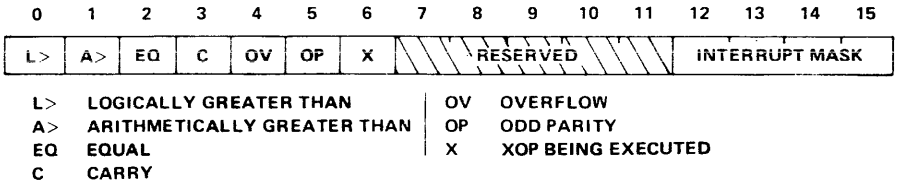


Figure 1-9. University Board Status Register

control to another area of the program based upon the results from a prior operation that set the SR bits. For example, after comparing two operands, the programmer might wish to use a transfer-of-control instruction such as "Jump if Equal." In such a situation, after finding the two operands to be equal, the machine automatically transfers control to another point in the program (not the next instruction as it would otherwise).

Buses. A bus is simply a common parallel path over which information is transferred, from any of several sources to any of several destinations. Referring back momentarily to Figure 1-2, it should be observed that all the interfaces to the CPU are buses. At the top is the data bus over which data going to and from memory and going to and from the I/O will flow. But, since this flow must be controlled, there must be control signals transmitted on the control bus to ensure that proper timing does occur. Another bus indicated on Figure 1-2 is the address bus. Addresses going to the memory and also to the I/O devices come out of this one-way bus.

The data bus itself is generally bi-directional, i.e., data flows both ways over the same lines under control of signals from the CPU. On the other hand, the control bus usually has a separate set of lines to handle input-sensing control signals and to handle the output of control signals generated by the CPU.

Memories

There would be no such thing as a stored-program digital computer if there were not some type of medium for storing the program. In fact this is so vital to computer technology that one can say that computer technology cannot itself advance without advances in the development of newer memories with certain desirable features.

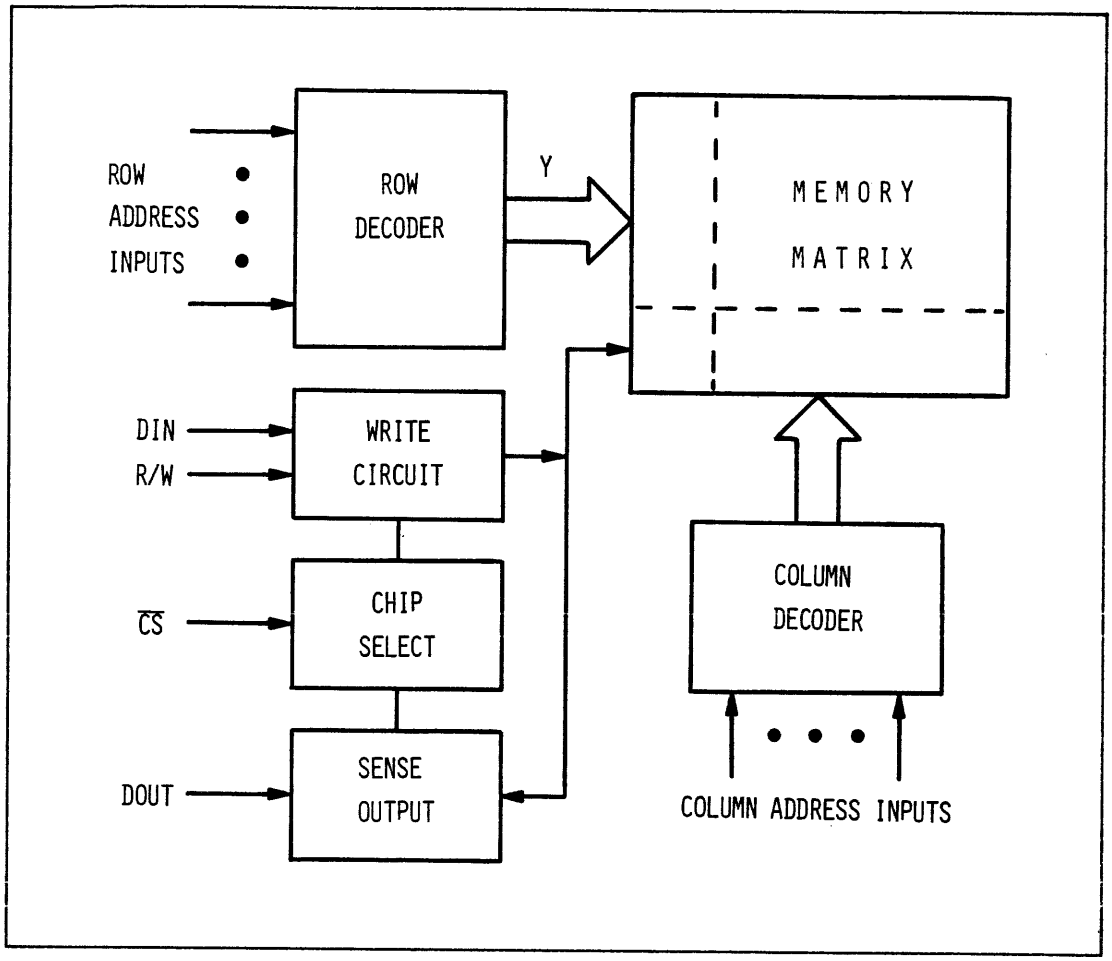


Figure 1-10. Block Diagram of Random Access Memory Chip

In general, one can say that the function of memory is that of a storage medium which holds a sequence of computer words. Each memory word may contain a program instruction or it may contain simply data, either a constant or a variable. Each memory word has a distinct location called a memory address, and the data stored at that address is stored as a set of electrical voltages that represent binary ONE's and ZERO's. The number of memory devices (generally called "chips") required by a given computer system will depend upon its total needs with regard to read/write and read-only memory. Details on the configurations of both types of these memories are given in Chapter 5.

There are two main types of memories: nonvolatile (permanent) and volatile (requires power in order to maintain the memory contents). Nonvolatile memories are further broken down into read/write and read-only categories, although most nonvolatile semiconductor memory devices are in the read-only category. There have

been many improvements and breakthroughs in semiconductor memory technology, some of which are reviewed in Chapter 5.

For a clearer understanding of memories and memory systems, Figure 1-10 shows a typical read/write semiconductor memory device. Such devices are generally referred to as RAM, random access memory. Referring to the diagram, a memory address from the CPU's address bus is provided to the address pins on the memory chip (or chips). This address can be considered to contain two fields as in an X-Y coordinate system. Once the X-Y coordinates settle in the memory chip they are decoded by the respective row and column decoder and thus select a given cell position. The cell position is often replicated within the chip to provide a parallel N-bit word at the selected X-Y address position. If the chip-select (CS-) input is active, then the RAM chip would be ready for either reading or writing depending upon the activity on the read/write (R/W) line shown on the diagram. Typically, the data coming in and going out of the chip are handled with a single bi-directional data I/O port consisting of one or more pins. In Figure 1-10, separate IN and OUT pins are shown.

Input/Output

The computer cannot process data without getting data from the outside (input). Likewise results or control to external devices cannot happen without output. Thus, an important measure of the versatility and utility of a computer is its ability to communicate with external devices or equipment.

One of the simplest configurations for handling I/O is to use a special input/output register. Whenever a data word is ready for output, it is transferred from memory or from the accumulator into this I/O register. Or, conversely, when an external peripheral device needs to input a word into the CPU, it merely puts the word in its output register and sets its status latch to indicate that data is ready to be transferred into the computer.

Typical I/O Configuration. Referring to Figure 1-11, it can be seen that a CPU not only requires the I/O register but also requires an address register. The latter can be expressly for I/O operations or the CPU can share the same address register as that used for memory operations. If the memory-address register is used and if the data appears on the data bus in conjunction with memory control signals, the type of I/O is called memory-mapped input/output. In this case, the I/O circuits look like memory circuits to the CPU, and are "mapped" into certain addresses (circuits respond when certain address line values are present). Alternately, of course, the I/O register can be on data lines separate from the memory data lines, but still share the address lines. Another scheme might be to share common address and data lines, but provide separate I/O control signals. In still another type of CPU, the I/O register shown on the diagram might also be used to carry the address in

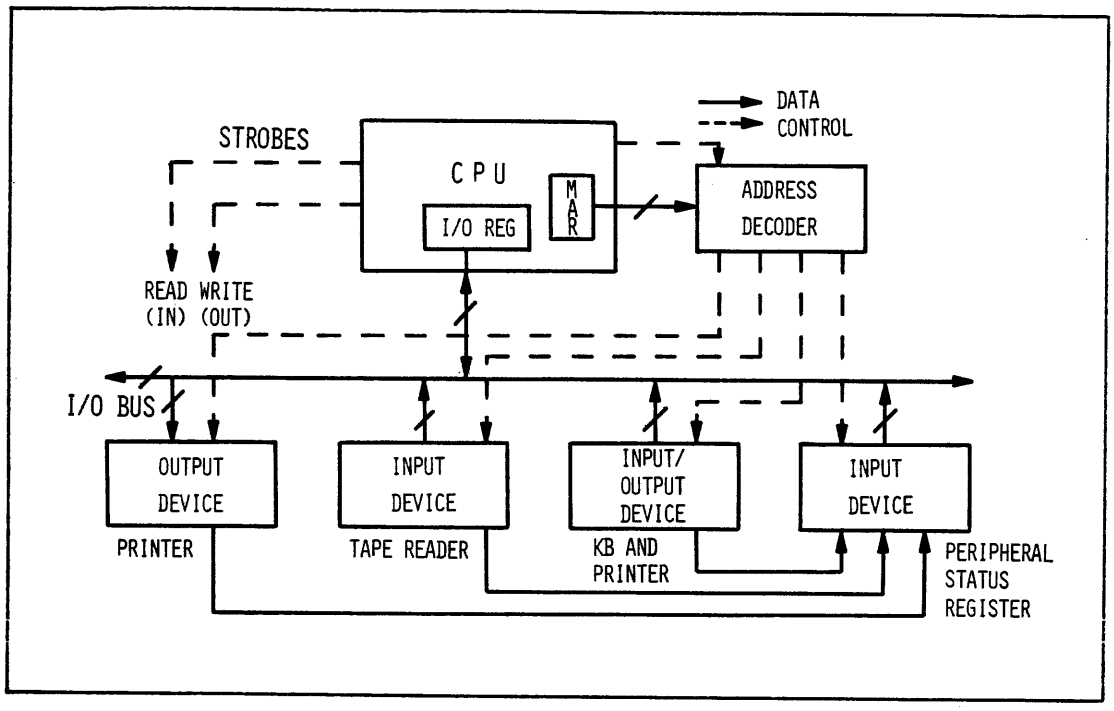


Figure 1-11. Typical I/O Configuration

a time-multiplexed mode. In such cases, the address would appear on the I/O register first followed by the data. Thus, the address would have to be latched externally to hold it for decoding. Subsequently, the data to be transferred would then appear on the I/O bus.

In addition to the I/O register and the address register, it should be noted that two strobes are a part of the I/O interfacing: read (IN) and write (OUT). Connected to the I/O bus in Figure 1-11 are one output device (printer), two input devices (tape-reader and peripheral status register), and one input/output device (keyboard/ printer terminal). Alternately, the two signals can be organized such that one indicates active I/O, and the other gives the direction, say, a ONE for a read (IN) or a ZERO for a write (OUT).

I/O Operation. The operation to be described is called program-controlled I/O. For the moment, assume that the CPU has just fetched an input instruction. The reason for the input instruction is to have the CPU check, say, a peripheral status register. The latter contains the status of various peripheral devices connected to the CPU. In short, the CPU is checking to determine if a given peripheral device is ready to receive data or if it has data to transmit to the CPU. This operation is called polling and, as can be surmised, it can be wasteful of CPU time if the data being input is coming from a keyboard activated by a human operator,

a relatively slow form of input compared to the speed of the CPU.

In some cases, keyboard data may be loaded in a terminal buffer as a sequence or block of characters. The time spent polling or waiting for the device to be ready is called software overhead. The temporary storage of characters in a buffer serves to reduce this overhead. When a device is ready another input instruction is executed. The input operation proceeds as follows.

- The input device address is placed on an address bus.
- An address strobe is sent out on the appropriate control line.
- The device places its data onto the data input bus.
- A data strobe is sent on the appropriate control line to acknowledge to the peripheral device that the data has been received by the CPU.

A similar sequence of events occurs for outputting data to a peripheral device. The principal exception is that the CPU would first place the device address on the address bus, then place on the data bus the word of data to be transferred. Strobes function in the same way as discussed above.

It should be stressed that the timing for this particular I/O operation is constant, but, in any case, it should be sufficient to make certain that the address decoder for the peripheral devices (shown in Figure 1-11) has sufficient time to settle down and to activate the addressed peripheral device.

To summarize the I/O operation described above, it can be said that three actions take place to perform an input/output operation:

- An address must be sent out to the peripheral device to be decoded.
- The data must be placed on the data bus going to or coming from the peripheral device.
- Suitable strobes or clock signals must be provided to ensure synchronous operations of all transfers.

As mentioned earlier, the particular configuration and operation of input/output described above is called program controlled I/O. Several significant enhancements or improvements in I/O operations are discussed in a later section. The principal weakness of the program-controlled I/O operation is that in some applications a great deal of time could be spent in polling and therefore not performing any useful work.

A problem common to most computers is that multiple devices need to be connected to the CPU. Therefore, some means of isolating each of the devices is necessary so that only the addressed device is active on the I/O bus. In earlier computers, this was handled with what is called open collector interface devices using what is termed collector logic. With time, improvements have been made in this area as discussed later in the chapter.

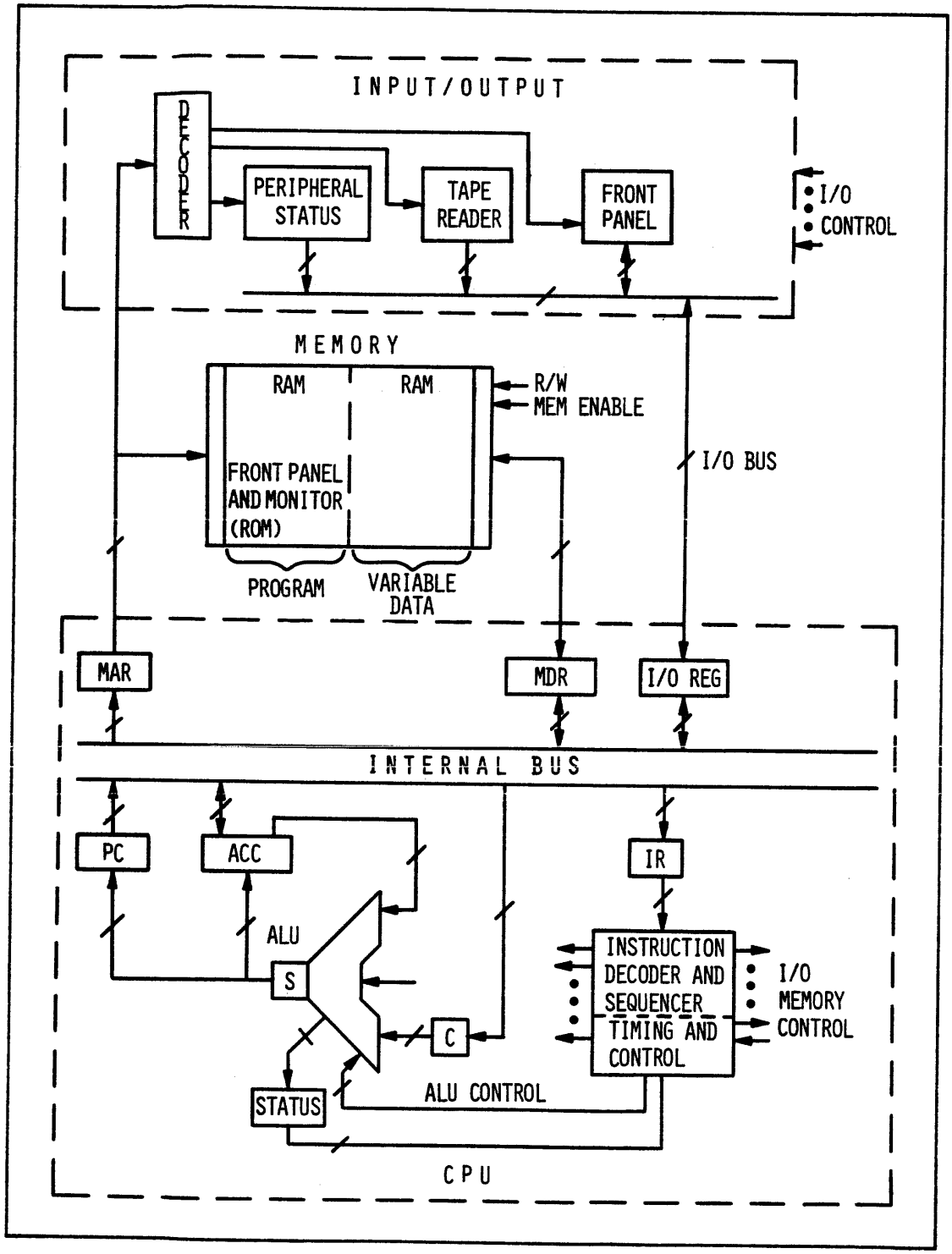


Figure 1-12. Computer Configuration Showing CPU, Memory, and I/O

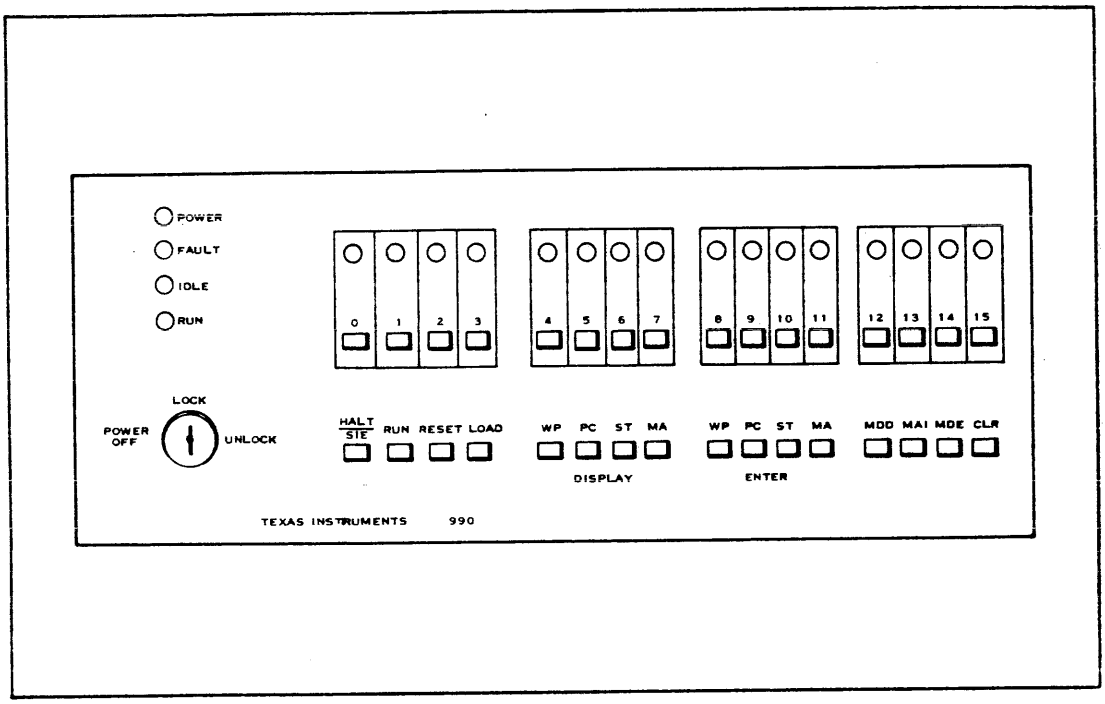


Figure 1-13. An Example of an Operator Control Panel

So that the reader will more fully understand the function of the various component parts of a computer, a program example is now given.

1.3 EXAMPLE OF COMPUTER OPERATION

To illustrate a computer's operation, Figure 1-12 is used to provide a view of the entire computer with CPU, memory, and input/output. There is sufficient detail to trace various data and program words throughout the system. To avoid starting at too elementary a level, it is assumed that already loaded into the memory are two programs in ROM, namely:

- Front panel service routine--a program which senses the positions of 16 switches used for inputting a 16-bit word (either address or data), and 16 display lights (LED's) used for displaying a result. In addition, there are other inputs such as pushbuttons that are used to initiate certain actions. See Figure 1-13 for an example of a typical front panel configuration.

- Monitor--a program that enables the user to communicate with the CPU. It allows him to input programs by means of a tape recorder/reproducer or keyboard and also to print out data on a terminal printer. In addition,

the monitor allows the user to inspect and modify the contents of the program counter, the status register, and the accumulator. All of this is achieved by means of simple commands from the keyboard followed by appropriate numerical data (generally hexadecimal format).

Having both a front-panel subroutine and a monitor now permits the user to make a choice on how he will load his program. He can load it through the tape reader using the monitor's built-in loader, or he could use the keyboard and simply modify memory. Further, he can load it a word at a time by means of the front-panel switches. If the program is short, it really will not make much difference which way it is loaded, because it takes very little time to load a short program by means of the front switches or the keyboard. A point to be stressed is that, in any case, the user must find some means to get his program into the memory of the system if he wants to execute it. If the program were longer and used frequently, it could be stored in one or more ROM chips which are merely plugged into the memory of the computer and, therefore, would be ready for execution.

Program Example

The program selected for this example is as follows.

> Add the two values stored in memory at locations X and Y, put the sum (result) out to the binary front-panel display and then branch to the monitor program. If an output carry occurs in the addition process, put out all ONE's to the binary front panel display, then branch to the monitor.

A flowchart and complete listing of the program example are given in Table 1-1.

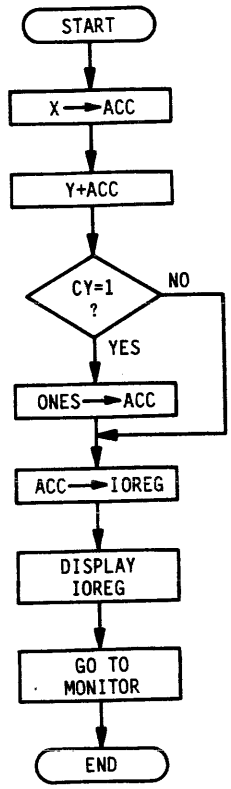
Prior to stepping through each instruction and data movement on Figure 1-12, it would be helpful to first review the binary formats of computer instructions generally and the purpose and function of each particular instruction in this program example.

Several examples of instruction formats are depicted in Figure 1-14. A single-word instruction is illustrated in Figure 1-14(a), indicating that half of the N-bit word is used as an operation code ("op code") field, and the remainder is used to contain an immediate operand or an address of an operand. When a CPU has several addressable registers or accumulators, then often the address of two of them can be put in this portion (field) of the instruction. A direct memory address using the entire second word of a two-word instruction is shown in Figure 1-14(b). An N-bit word thus addresses a total of 2^N words in memory. The format for a two-word instruction with the second word as an immediate operand is illustrated in Figure 1-14(c). Not shown, but common in more sophisticated instruction sets, is a three-word instruction consisting of an op code (first word) and <u>two</u> memory addresses (second and third words).

TABLE 1-1

LISTING AND FLOWCHART OF PROGRAM EXAMPLE

	LABEL	INSTRUCTION	COMMENT
1.	START	MOV X,ACC	MOVe the value at memory location X to ACCumulator.
2.		ADD Y,ACC	ADD the value at memory location Y to accumulator; if carry occurs, indicate in status register.
3.		JOC ERR	Jump On Carry to instruction labeled ERR, i.e., if carry occurs, jump.
4.	DISP	MOV ACC,IOREG	MOVe value in ACC to IOREG.
5.		OUT FP	OUTput contents of IOREG to address of Front Panel.
6.		BR MONITOR	BRanch unconditionally to monitor program.
7.	ERR	LDI ACC,ALLONES	LoaD Immediate ACC with all ONE's.
8.		JMP DISPLAY	JuMP unconditionally to instruction labeled DISPLAY.



With regard to the program example, the first instruction says to move the value at memory location X to the accumulator. Sometimes this is referred to as a "load" instruction. The second instruction says to add the value at memory location Y to the contents of the accumulator (which prior to the execution of instruction 2 contained the value that was moved to it from location X). The third instruction says to jump on carry, that is, take a jump to location labeled ERROR if an output carry occurs in the addition process executed in the prior instruction. This means that the CPU will have to test or sense the carry-status latch in the status register to determine whether or not a jump is to be taken. At the moment, assume that a carry did not occur. Instruction 4 requires that the contents of the accumulator be moved to the I/O register. Instruction 5 requires that the contents of the I/O register be output to the front panel--a peripheral device--the address of which is indicated with the abbreviation FP.

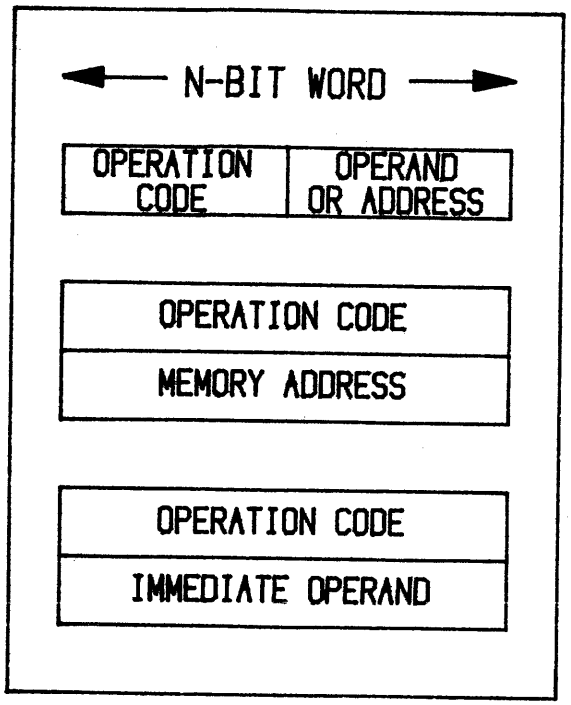


Figure 1-14. Typical Instruction Formats

The sixth instruction requires that the CPU make an unconditional branch (change in the contents of the program counter) to the starting address of the monitor program. Generally, when the monitor program is executed it indicates that it is working properly by issuing a "prompt" character to the printer or display device. The prompt character indicates to the operator that he may enter a monitor command, say, to inspect or modify memory, or, perhaps, to inspect or change the program counter, accumulator, or status register.

Going back to instruction 3, suppose that a carry was indeed generated in the addition process. The CPU, having sensed the carry on the status register, would then proceed to add a displacement value to the contents of the program counter, which would cause the computer to jump over or skip three instructions (instructions 4-6) and proceed directly to the instruction labled ERR. This instruction requires that the accumulator be loaded immediately with a value of all ONE's. In short, a binary word consisting of all ONE's is loaded into the accumulator by this instruction. Instruction 8 then causes the program to jump unconditionally to the instruction labled DISP whereupon the contents of the accumulator would be moved to the I/O register and then on out to the front-panel display before branching to the monitor. Thus, the operator would see at a glance that all of the light indicators on the front-panel display are active, clearly indicating that a carry condition did exist in the addition process.

Loading the Program

Since the program example is relatively short (about a dozen words), it will be loaded by means of the front-panel switches. The front-panel switches are first selected to input the address where the first instruction is to be located, say, memory location 100. With the address set up on the switches of the front panel, an appropriate pushbutton is depressed causing the address to be loaded into the CPU's memory address register. See circle 1 path on Figure 1-15. A switch on the front panel now selects data to be input at that address, and the first word of the first instruction is set up on the front-panel switches. The LOAD DATA pushbutton is depressed indicating that the data is to be transferred through

the I/O register and the memory-data register into the memory a location 100, indicated with a circle 2 on the diagram. At thi point the memory-address register is incremented, handled by a sep arate pushbutton on the front panel. The new incremented value move along the circle 1 path. The second word of the first instructio contains the address X = 120, for this example. The LOAD DAT pushbutton is depressed again causing the value 120 to be transferre through the I/O register and the memory-data register into th location at address 101, illustrated again in Figure 1-15 wit the circle 2 on the dotted line.

In a similar way, the second instruction, also consisting c two words, is loaded through the front-panel switches.

As in the first instruction, the initial word of the secor instruction contains the operation code of the instruction indicatir that the accumulator is to be the destination of this particul operation. The second word of the instruction contains the addres of location Y, namely 121. The remaining instructions are load one word at a time until instruction 8 is loaded, at which tir the loading process is complete.

Initialization of Data and Registers

In this example, values for X and Y at locations 120 and 12 respectively, are loaded into memory. The program counter th must be initialized to the first instruction of the program, locati 100, using the front-panel switches. Circle 2 and 3 paths on Figu 1-15 indicate the data movement path for this. Actually, under fron panel control, the value for the PC loaded through the front-pan switches is not immediately transferred to the PC but is held a temporary register until the RUN button is depressed. Both t accumulator and the status register do not require initializati for proper execution of this program. Therefore, the program now ready for execution.

Execution

The next step is to cause the program to be executed by mea of depressing a pushbutton on the front panel called RUN. Up this point, the computer has been running; however, it has be under full control of the front-panel subroutine--the subrouti that has allowed the operator to load in one at a time the instructio required for his program. In order to execute a user progra the front-panel pushbutton labeled RUN must be depressed, resulti in the program-counter value entered earlier being loaded into t program counter to cause the first instruction of the user's progr to be fetched.

Referring now to Figure 1-16, the contents of the progr counter are transferred to the memory-address register and th on the memory, shown with a circle 4. The word at location 1

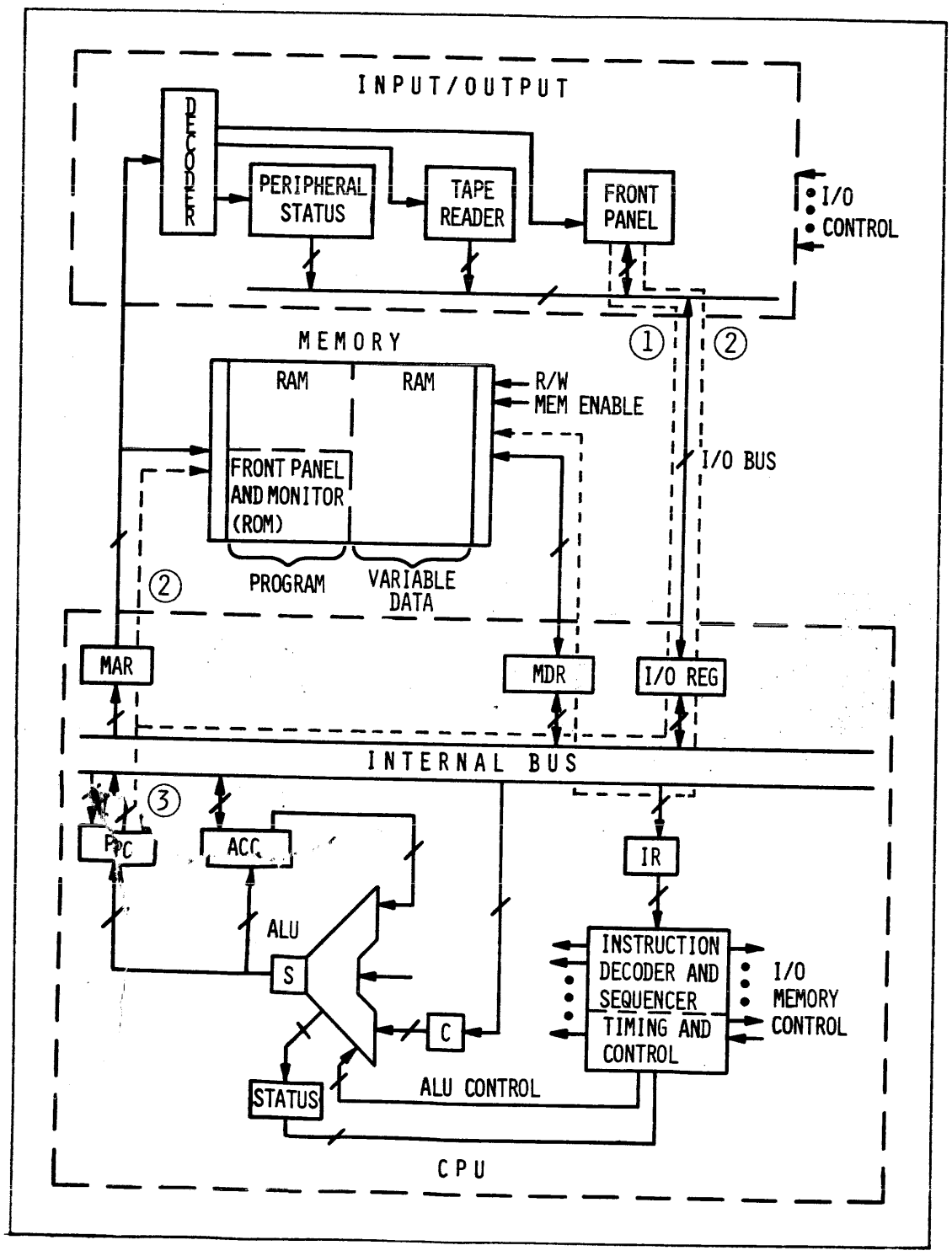


Figure 1-15. Paths for Loading the Program and Initializing the PC

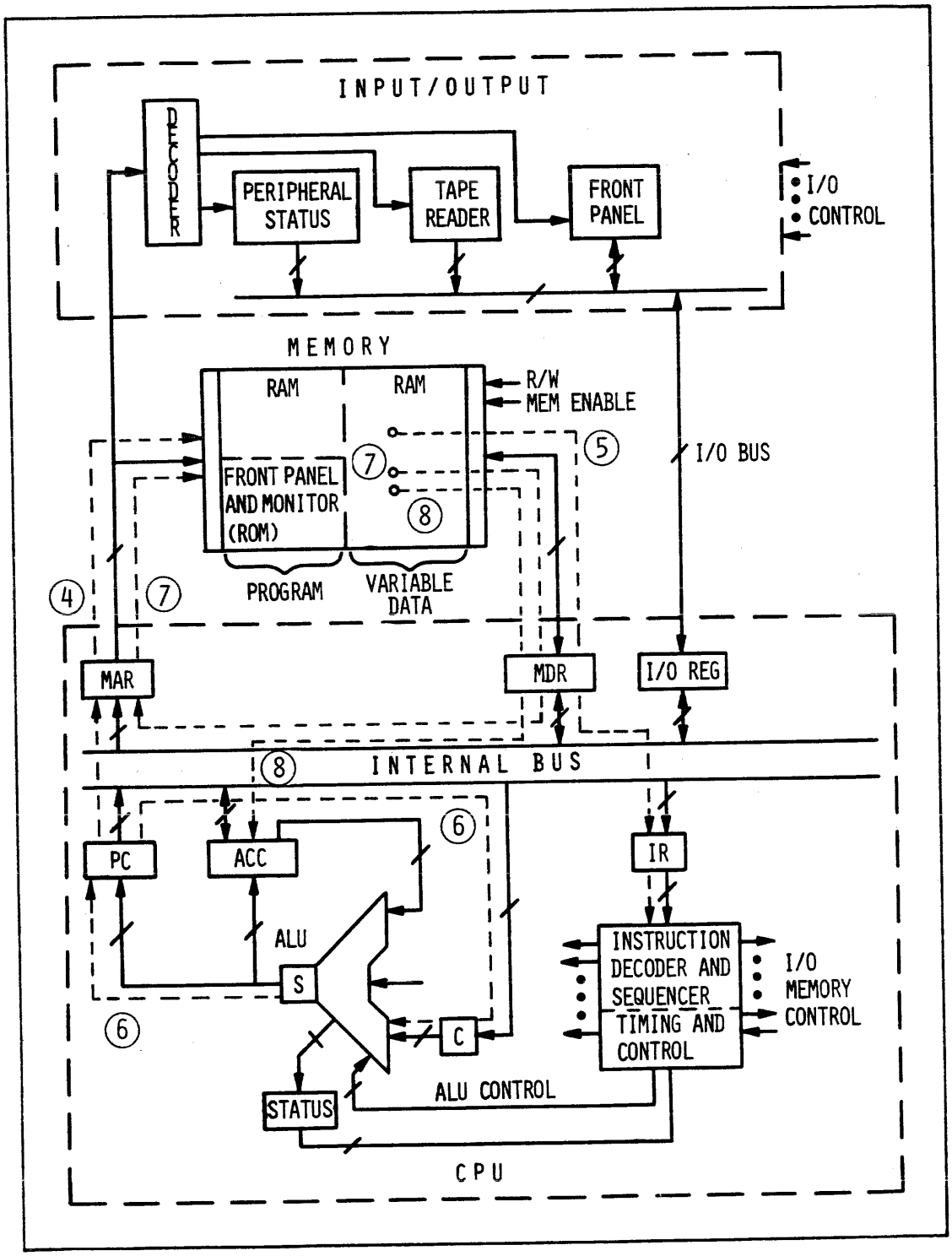


Figure 1-16. Instruction and Data Transfers Required for Execution of the First Instruction

(the first instruction) is now transferred out of memory into the memory-data register and then over to the instruction register as indicated by a circle 5. The instruction is decoded and requires that the second word of the instruction be fetched before the instruction can be executed. While the first instruction word was being transferred to the instruction register the program-counter contents were fed to the ALU to be incremented concurrently and the incremented value placed back in the program counter, circle 6 path. Thus the second word of the instruction, at address 101, is now fetched and brought not to the instruction register but to the memory-address register (MAR) as indicated by circle 7 on Figure 1-16.

This second word of the first instruction contains the address 120, which is what has been called location X. The value at location X is now moved from memory to the accumulator by means of the path indicated with circle 8.

Referring now to Figure 1-17, the third word is fetched in a similar way, labeled with circle 9. It contains the instruction for ADD. The second word of this instruction contains the address Y (121) which is fetched and transferred to the MAR (circle 10) in order that the contents of location of 121 may be transferred from the memory through the MDR to the B input or port of the ALU (circle 11). Notice also that the contents of the accumulator (containing the value at location 120) are now connected into the other port of the ALU (circle 12). Once these values are in place, the control unit issues the appropriate control signal to the ALU to cause the addition of the two values, one from the accumulator and one that has been placed in the MDR from memory. The result is now transferred to the accumulator (circle 13).

Assume for the moment that no carry occurred in the addition process. Then when instruction 3 is fetched, the control unit would sense the carry flip-flop in the status register and detect that no carry existed. Consequently, the next instruction would be fetched and cause the contents of the accumulator to be moved to the I/O register, circle 14 on Figure 1-17. When instruction 5 is fetched, the control unit causes the contents of the I/O register to be transferred to the address of the front panel. The address is contained in the second word of the OUT instruction. Therefore, before this instruction can be completed the address of the front panel must be transferred from memory (second word of the instruction) to the memory-address register (MAR) and then on out to the decoder in the input/output section of the diagram. See circle 15 path out of MAR. The decoder recognizes the address for the front panel and activates the front panel (circle 16) to receive the word being transferred from the I/O bus (circle 17). The final step in the instruction execution is a strobe generated by the control unit to cause the front panel to receive and hold (latch) the contents transferred to it from the I/O register.

Once this transfer of data to the front panel is made, the next fetch executes a branch to the monitor, causing the first instruction of the monitor program to be brought into the instruction

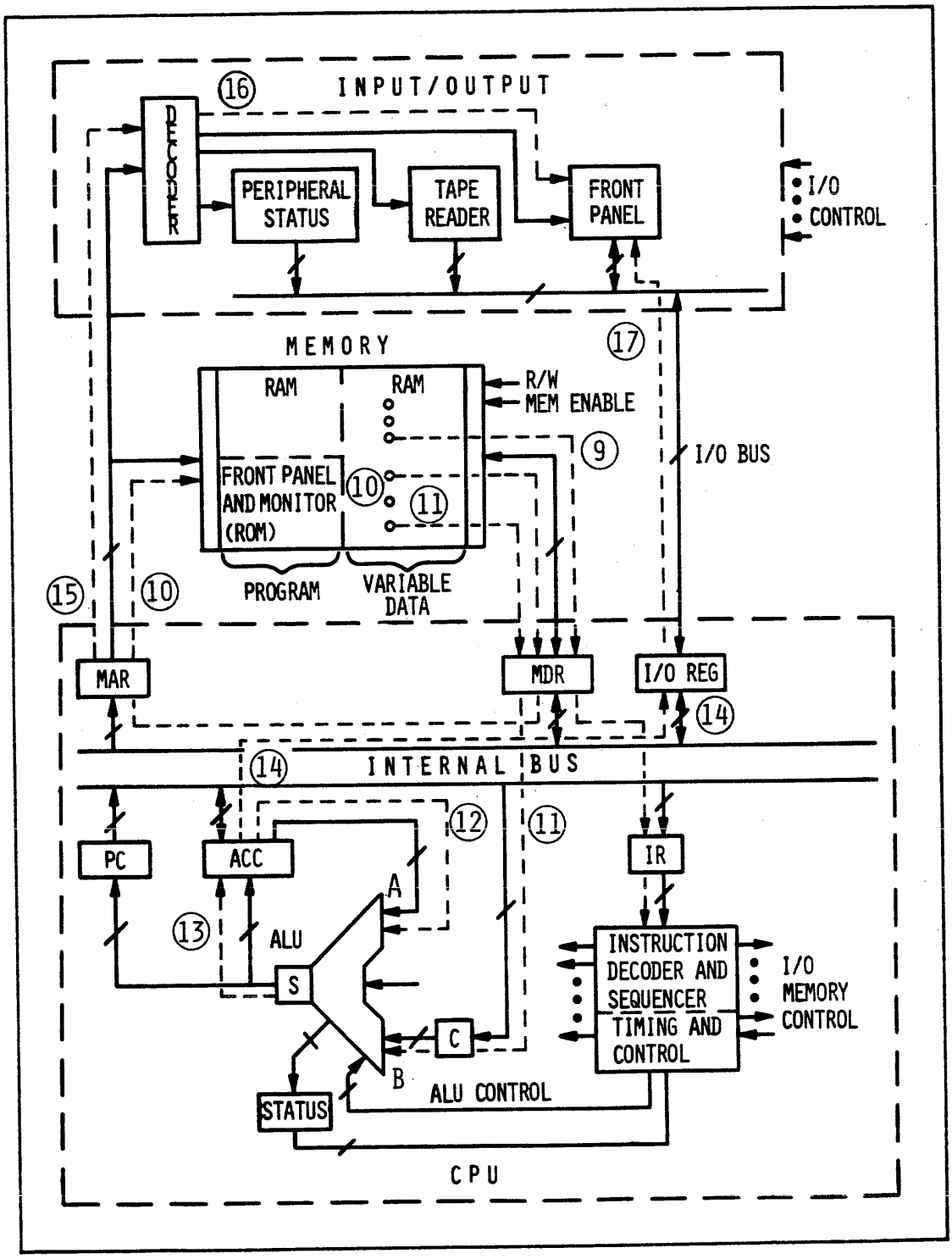


Figure 1-17. Data Transfers Required for Execution of Additional Instructions

register. At this point the user's program has been completed, and the CPU is now operating under control of instructions in the monitor program. Consequently, a prompt would be generated by the program and transferred through the I/O register to the printer shown on the diagram, Figure 1-17.

Going back to the program example for a moment, assume now that the addition process did result in a carry. In this case the control unit senses the carry and instead of fetching instruction 4, it fetches instruction 7. It achieves this by taking the displacement value (typically eight bits) from the JUMP ON CARRY instruction and adds it to the program-counter value utilizing the ALU, thereby modifying the PC to the required address of instruction 7. With instruction 7 (LDI), the control unit recognizes that the instruction contains a second word (all ONE's). Thus it fetches this word and brings it directly into the accumulator as required by the instruction. It then proceeds to fetch the next instruction, number 8 (JMP DISPLAY), which requires an unconditional jump to the location labeled DISPLAY. This means that the unconditional-jump instruction contains a negative displacement value. This negative value, when added to the present contents of the program counter (already advanced to the next location), results in the address of instruction 4. The program proceeds as discussed above to complete the execution of the program and branch to the monitor.

Instruction 6 (BR MONITOR), when fetched, requires that the second word of the instruction be transferred from memory directly into the program counter in order to cause an immediate change in the program counter to the starting point of the monitor program. This means, of course, that the old value of the program counter is destroyed and replaced by the address of the monitor's first instruction.

With this illustration of a program execution, it should be evident that many transfers and many control signals occur throughout a CPU in a precise and logical order to cause the program to be executed. Of course, the program itself must be designed so that each step is the logical next step for proper processing of the data to obtain the desired result.

If the operator desired to execute the program a second time, then he could, by means of the monitor program, go into the memory with a modify memory keyboard monitor command and modify the contents at locations X and Y to set the program up for new data to be added. Then the monitor can be used to modify the value in the program counter to prepare for execution of the user program. A special keyboard command is entered on the keyboard to cause the program to run. Thus, loading of data into the memory, initialization of the program counter, and execution of the program can all be handled conveniently by using the monitor keyboard commands.

1.4 ARCHITECTURAL ENHANCEMENTS

Because of the rapid, almost revolutionary, developments in semiconductor technology, substantial improvements have been made in the architectural design and configuration of computers--particularly microprocessors--in recent years. This section reviews some of the highlights of these developments to enable the reader to have a better perspective of the total spectrum of capabilities now available in the form of microprocessor and microcomputer chips. The architectural enhancements discussed here cover the CPU, memory, and input/output.

CPU

Principal enhancements in the CPU's architecture include multiple accumulators, some special types of registers, a stack, and microprogram control. Each of these enhancements is covered in some detail below.

Multiple Accumulators. Probably the single feature which has added more convenience for the programmer has been the inclusion of multiple accumulators or working registers. Such an arrangement may well imply the need for dual internal buses so that both a source and a destination register can be addressed in an instruction. As pointed out earlier, an accumulator or working register allows the programmer to hold the temporary results for use in some later process. With some CPU's, the multiple accumulators may also have functions other than as general-purpose accumulators. One example of a multiple register arrangement in a CPU is given in Figure 1-18.

This diagram shows how a single demultiplexer is used to take the result, say, from the ALU and transfer it in the selected register (R0,R1...,R15). On the other side of the registers are found a source multiplexer and a destination multiplexer. The source multiplexer selects one of the 16 registers for placement on the A bus while the other multiplexer selects one of the 16 registers for connection to the B bus. (Details on the operation of data multiplexers and demultiplexers are provided in The TTL Data Book for Design Engineers, Second Edition.)

Index Register. Often a CPU contains what is called an index register. This register is used to hold a displacement or offset value to, say, the beginning of a table of constants in memory. Then an instruction can reference any word in the table merely by adding this displacement or offset value to the address portion of the instruction. Thus, when the instruction is fetched, the address portion is transferred to one port of the ALU. To the other port is transferred the contents of the index register. Addition is then performed, resulting in what is called the target or effective address. The index register itself may be a completely separate register or it may be one or more of the working registers or

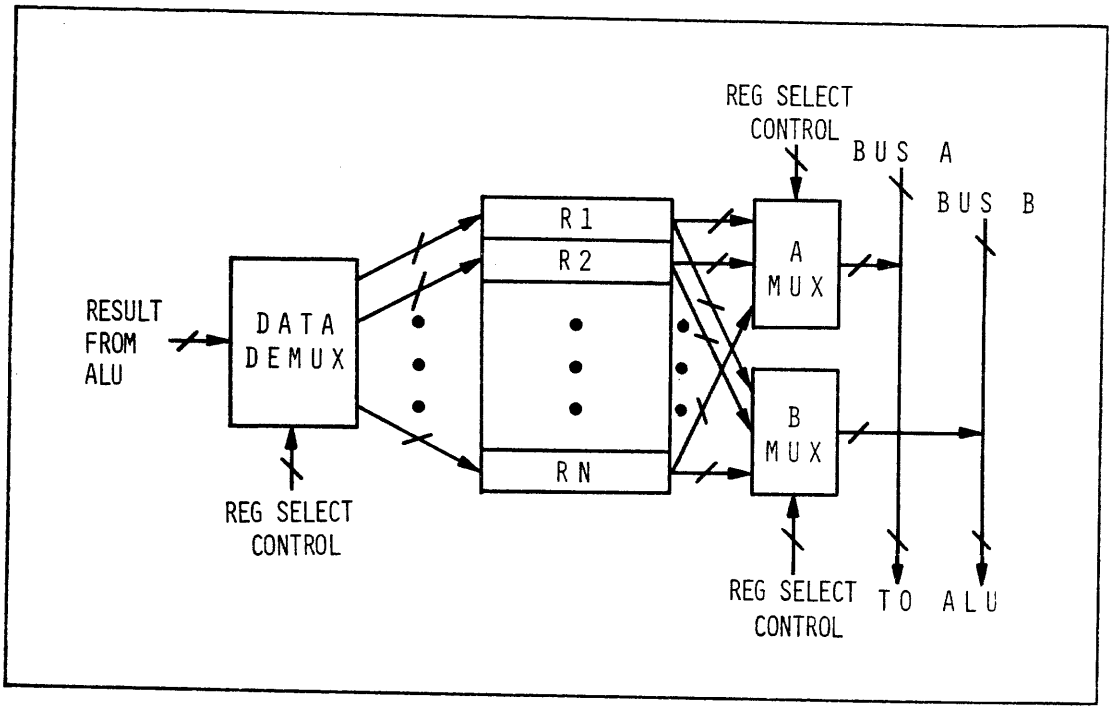


Figure 1-18. Multiple Accumulators or Working Registers

accumulators. Figure 1-19 depicts the process of forming an effective address by means of an index register.

It is stressed that this addition of an instruction address and the displacement in the index register to form the effective address does not normally change the value in the index register itself. By using such instructions as increment, decrement, add, and subtract, the programmer can readily modify the value in the index register and thereby systematically step through the table.

Workspace Pointer (WP). One of the most recent enhancements to CPU architecture is the workspace pointer. The workspace pointer is a register loaded under program control and used as a pointer (contains the address) to a block of, say, 16 workspace registers contained in RAM. This means that the accumulators or working registers are located in memory rather than in the CPU's hardware, as illustrated in Figure 1-20. The principal advantage of this concept is that the programmer can select more than one block of registers to solve a complex programming problem by merely reloading the workspace pointer whenever his program requires a "context switch" (e.g., interrupt). In short, when the programmer reaches a point in his program where he needs more registers to perform a subroutine, he merely uses a type of branch instruction which contains not only an address to the new portion of the memory where the subroutine is located but also a new value which is loaded

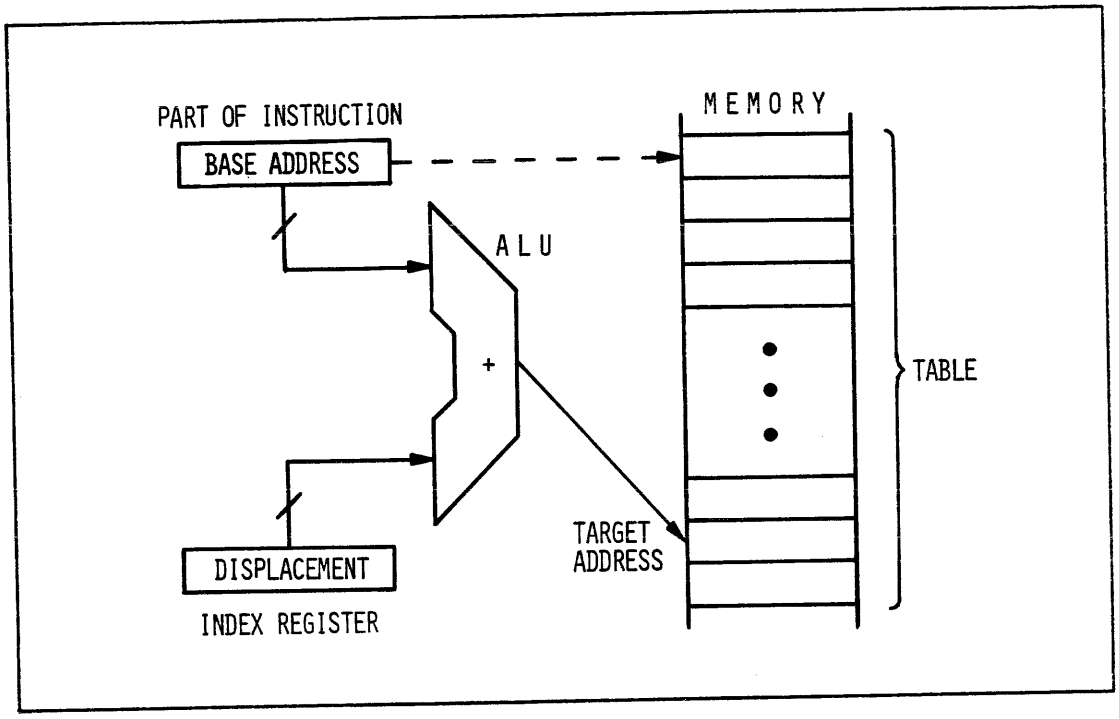


Figure 1-19. Index Register Operation

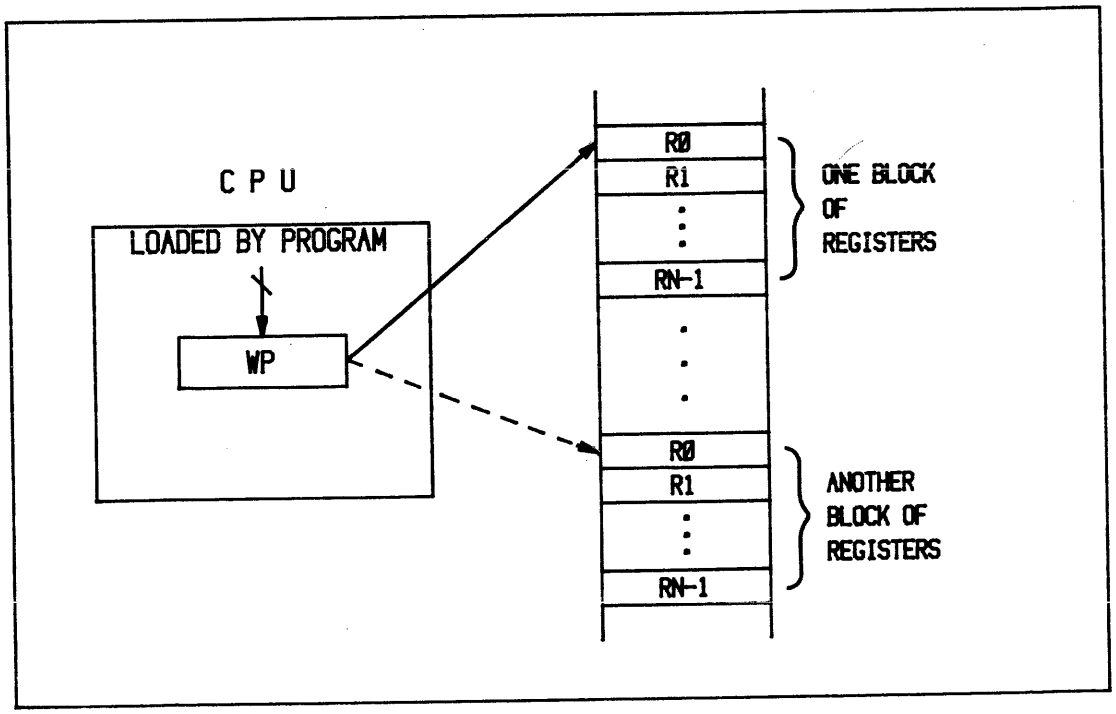


Figure 1-20. Workspace Register Concept

into the workspace pointer. This allows the programmer to use a separate block of registers exclusively for the subroutine. Naturally, the last program counter value and the prior value of the workspace pointer have to be saved in a context switch, but there is no requirement to save the prior block of registers since they are already stored in memory, thereby saving time. Exactly how the "context switch" is achieved is treated in Chapter 8. The single drawback of using a workspace pointer with registers in RAM is that it takes more time to access memory when processing instructions referring to workspace registers. Recent experience indicates that the advantages of the workspace pointer concept outweigh this single drawback, especially in real-time (interrupt-driven) applications.

Stack. A hardware stack is a special kind of memory which has sequential access in the following way. Words are pushed on to the the top of the stack in sequence and are pulled off the top of the stack in reverse order. Such a memory is called last-in-first-out (LIFO). Words are pushed one at a time onto the stack from a bus and pulled off the top of the stack back onto the bus as shown in Figure 1-21.

For housekeeping purposes, there is special logic connected to the bottom of the stack (OR gate) so that whenever a word moves into the bottom location, a stack full signal is generated and fed to the control portion of the CPU. A similar signal is sometimes used to indicate that the stack is empty.

The main purpose of the stack is to allow the programmer to save the contents of the registers (the CPU machine state) on the stack for temporary storage while the program transfers to another location for processing a subroutine or an interrupt. As discussed previously, a CPU using a workspace pointer achieves the same result quite conveniently by means of the "context switch." The addressing mode for a stack is particularly simple since there is only one address to the stack, namely, the "top." The stack's usefulness in nested subroutine operation can be significant.

Stack Pointer(SP). Some central processing units do not provide for either a stack or a workspace pointer in their hardware configuration, but instead use what is called a stack pointer. This is a register that is initialized by the programmer and points to the location of the first place in RAM memory designated by the programmer as stack. This is illustrated in Figure 1-22 wherein a number of locations are shown below the top location of stack. This is sometimes called a software stack and places the burden of housekeeping on the programmer to make certain that stack operations are always between the top and bottom locations. The burden is somewhat offset by the advantage that a larger amount of stack storage is available in memory which can accommodate a very deep stack before reaching "bottom."

To illustrate, assume that the stack pointer is initialized to memory location 1031 and that the bottom location has the address of 1000. This provides a total of 32 word locations for use by the programmer for all stack operations. The first word pushed onto the stack is stored at location 1031, after which the SP is automatically decremented to the address 1030. The second word is pushed onto the stack at location 1030 and again the SP is automtically decremented to the value 1029, etc. When a PULL (or POP) instruction is executed, the CPU automatically increments the stack pointer to the next higher address; then it transfers this word. This leaves that particular location available for a PUSH instruction if one should occur prior to a PULL instruction.

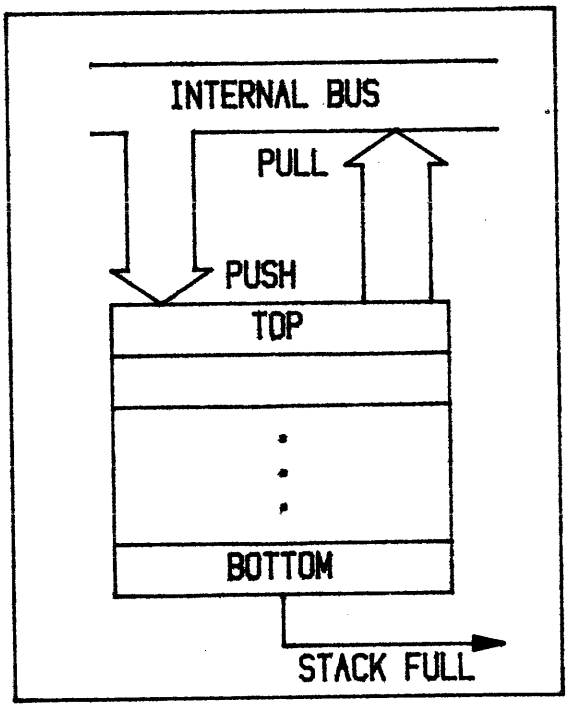


Figure 1-21. Stack

It should be clear that the operation of the stack pointer address is such that the software stack works on the same basis as the hardware stack; namely, lastin-first-out. The "housekeeping" chore of keeping all stack operations within the top and the bottom locations has to be assumed by the programmer. This is generally handled by checking the value of the stack pointer after each operation. Needless to say, if the value in the stack pointer ever goes outside the limits of the top and the bottom addresses, temporary data could be destroyed thereby causing erroneous program operation and results.

Microprogram Control. The use of microprogram control in a processor to decode and generate the various control signals within a CPU represents a significant improvement in the systematic design and operation of the control portion of the CPU. In one sense, the microprogram control portion of a CPU may be considered an inner computer. In brief, when an instruction is fetched, the instruction causes a sequence of micro instructions within the microprogrammed control unit to commence execution. Each micro instruction directs the central processing unit to perform one step in the execution of the instruction. Thus, a set of micro instructions constitutes what is called a macro or machine instruction. Macro instructions are, therefore, the machine instructions used by the programmer to develop a program. In the vast majority of cases, the manufacturer of the CPU defines the instruction set in what is called microcode; therefore, the instruction set is fixed by

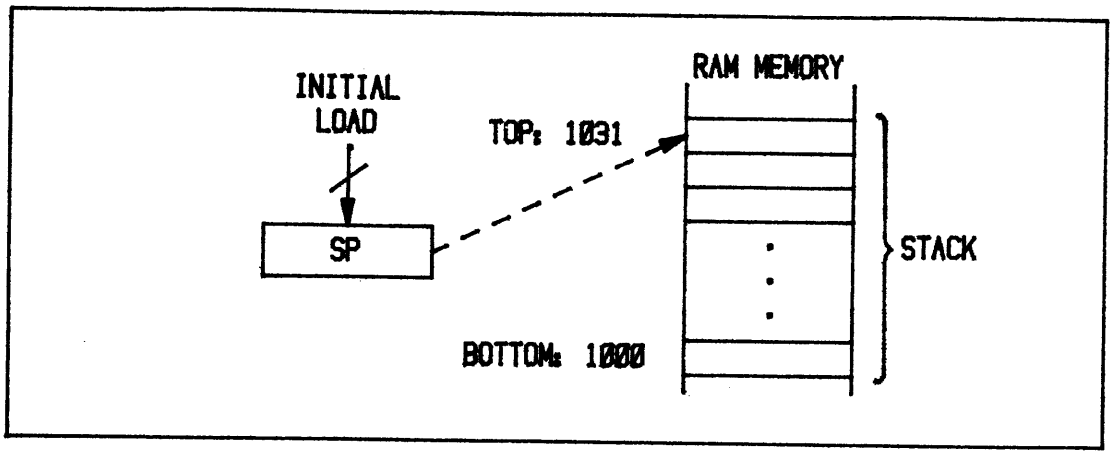


Figure 1-22. Stack Pointer

means of the internal mask that becomes a part of the standard processor design.

Although such a microcode was developed (microprogrammed) by the manufacturer, this does not mean that it is microprogrammable by a user. Generally, chips referred to as bit-slice microprocessors are microprogrammable. The latter constitute multiple-chip CPU's and are beyond the scope of this book; however, they should be considered when a very high speed, special purpose microcomputer is required.

Figure 1-23 depicts a configuration for a microprogrammed control unit.

Memory

In recent years there has been an upsurge and proliferation of semiconductor memory devices. In addition, new configurations of magnetic storage devices have been developed to provide bulk auxiliary storage of programs and data for rapid access. A review of the two major categories of nonvolatile and volatile storage devices available will provide some idea of the extent of the growth in this technology.

A list of various types of nonvolatile storage devices includes

Semiconductor devices

- ROM (Read Only Memory)
- PROM (Programmable Read Only Memory)
- EPROM (Eraseable Programmable Read Only Memory)
- EAROM (Electrically Alterable Read Only Memory)

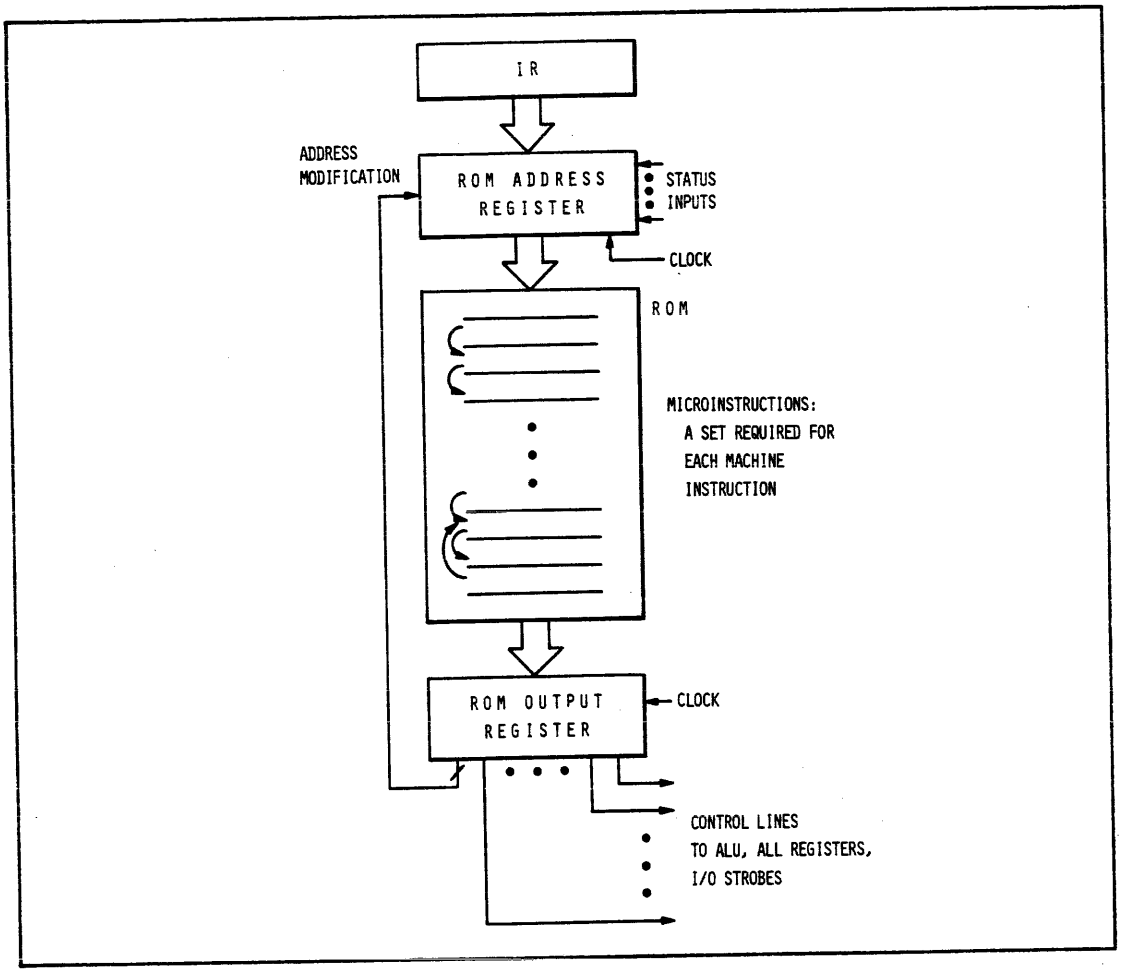


Figure 1-23. Microprogram Control Unit

Magnetic Devices

- Core (for years the principal type of computer main memory)
- Tape (reel-to-reel, cassette, and cartridge)
- Disk (rigid and floppy)
- Bubble (MBM, Magnetic Bubble Memory)
- Drum (rigid)

Nonmagnetic Devices

- Paper Tape
- Cards.

Similarly, a list of the various types of volatile memories, all of which are semiconductor, includes

- Static RAM (Random Access Memory)
- Dynamic RAM
- Shift Register (dynamic and static)
- CCD (Charge Coupled Device).

Some of the features, characteristics, and comparisons of the new semiconductor memory devices are reviewed below to provide some idea of the impact of this technology on engineering design and development as well as on production of products based on these devices.

ROM. Although a ROM memory also has random-access capability, for some reason the use of the word RAM is commonly applied only to read/write random-access memory. A read-only-memory is manufactured so that desired data will be read-out, and no other data can be written into the device. The data pattern is fixed at manufacturing time. The lowest cost approach to ROM requirements is met by using mask-programmed ROM's which means that a special processing mask of ONE's and ZERO's is used during device manufacture. Some of the advantages offered are

- Larger memory size per chip (number of words and size of the word) per chip
- Lowest cost in quantities
- Low power
- Fast access.

However, one must keep in mind disadvantages such as the relatively high cost of making the mask, and the need for absolutely correct data, because the mask is permanent and cannot be modified short of creating a whole new mask at the factory. Another factor to be kept in mind is the long lead time on initial delivery of the chips.

Some of the more common applications for mask programmed ROM's are

- Microprocessor program storage
- Lookup tables
- Code conversion
- Character generation and, in general
- All types of sequential ROM-driven controllers.

PROM. The invention of field-programmable semiconductor PROM's was a significant advance in semiconductor memories. These PROM's allow the user to create his own ROM by applying a series of electrical programming pulses to the device in a special piece of equipment called the PROM programmer. Generally, the programming process destroys a fusible link in a semiconductor cell to create the desired

logic state of a ZERO. If the fusible link is not destroyed the cell retains the logic ONE state.

Some of the clear advantages of the PROM over the ROM are

- Rapid turn-around during the development phase
- Low cost for a few copies
- When the project gets to the production stage there is usually a ROM counterpart which, in volume, can be purchased at a substantially lower cost.

Disadvantages to keep in mind are

- Difficult if not impossible to correct mistakes
- Programming equipment can be expensive.

Fortunately, many suppliers of PROM's have programming equipment that they make available to their customers.

EPROM. Invention of erasable PROM's was another breakthrough in memory technology. An EPROM can be programmed by a controlled series of electric impulses. If an error in the program is discovered, or if the program is discarded or transferred to mask ROM, the EPROM can be erased and reprogrammed. Even though EPROM's are relatively high in cost and often require expensive programming equipment, they still maintain significant advantages:

- Fast turn-around time
- Permanent storage
- Stored data can be changed as desired by erasing and reprogramming
- Good density
- Low power
- Fast access time
- Usually a pin-compatible ROM is available.

An ultraviolet light source is used to erase or clear the EPROM prior to reprogramming.

RAM. The emphasis here is on active semiconductor devices which require power to maintain the information being stored. In general, RAM is used to store intermediate or temporary results of the program as well as to store blocks of input and output data. In the latter function the RAM serves as an input and output buffer. There are two principal types of semiconductor RAM's: dynamic and static. Dynamic RAM's are generally of higher densities than static RAM's. The principle of operation of a dynamic RAM cell is that a parasitic capacity charge is stored, and in order to hold the charge, it must be regenerated periodically, i.e., refreshed. Generally, this refresh must be accomplished at least every two milliseconds. A failure to refresh will cause a loss of data even though power is still applied. Usually, dynamic RAM memory systems require

external hardware to periodically refresh the cells. In some cases, however, the microprocessor chip itself may have built-in circuity to handle this requirement.

Static RAM's use the multiple-gate transistor latch to store each bit in a memory cell; therefore, no refresh is required. Consequently, a minimum of external hardware is required, if any. Although static RAM's are less dense than dynamic RAM's, they will retain the data stored in them as long as proper power is supplied.

Input/Output

There have been considerable enhancements in the I/O capability of microprocessors in recent years. The improvements discussed in this section are

- Addressable single-bit I/O
- Vectored, prioritized, maskable interrupts
- Direct memory access (DMA)
- Special LSI chips for I/O timing and control
- Three-state logic devices

As pointed out earlier, the input/output capability of a processor represents one of its principal capabilities because controlling external devices and processing external input data are what processors are designed to do.

Single-Bit I/O. Ever since the birth of the first computer it has been possible to transfer data into and out of the processor in parallel by means of the I/O register or the data bus. However, the capability to address and to put out a single data bit or to read a single data bit into the processor has been relatively restricted to just a few of the available processors. When this feature has been available, these single I/O bits have been restricted in number to approximately a half dozen or less. More recently, processors have been introduced which allow the programmer to address any one of up to 4096 individual I/O bits. Although a system that requires that many I/O bits would be rare, it is not unusual in certain types of control applications to require several hundred individual input (sense) bits and several hundred output (discrete control) bits.

The two principal ways that single-bit I/O is achieved are illustrated in Figures 1-24 and 1-25. Whenever this capability is built in (Figure 1-24), the number of control flags (outputs) must be restricted as well as the number of sense inputs, since pin limitations force this restriction on the designer of the device. As mentioned earlier, using external hardware devices such as addressable latches and multiplexer chips makes it convenient to add on exactly the number required for a given application. One such configuration is shown in Figure 1-25. Further details on this single-

bit I/O architectural enhancement shed in Chapter 6.

The major advantage of single-bit I/O is the ease of handling discrete input and output. A large percentage of control applications involve sensing the position of switches, pushbuttons, and other single-bit information. After sensing, the processor makes a decision to control certain discrete output peripherals such as solenoids, relays, display LED's, etc.

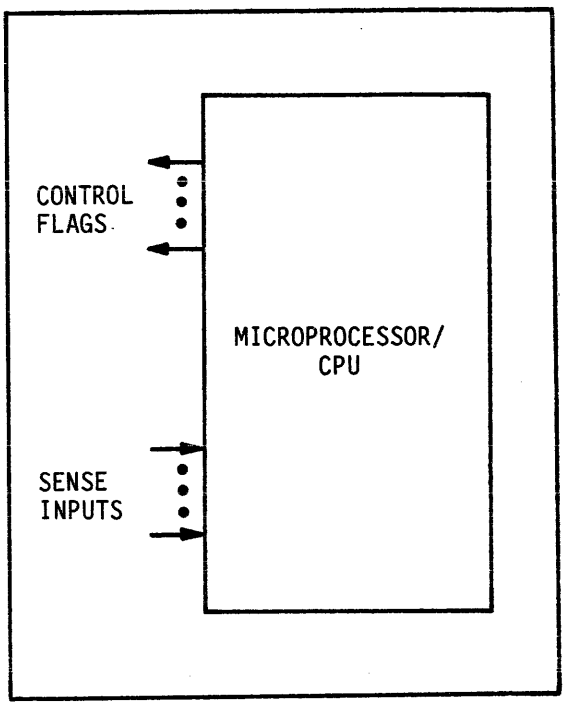


Figure 1-24. Single Bit I/O: Built-In Capability

Interrupts. The use of computers for real-time applications involving control and processing created a need for a new capability that was not required in nonreal-time applications. The word real-time implies the need for immediate service of one type or another in situations where the events occurring are time-dependent. Some typical real-time applications are

- Fire-detection system
- Limit switch in a machine controlled by a processor
- High-speed terminal interfaced to a processor.

Several examples of nonreal-time processing which require no interrupt capability are

- Processing of payroll checks
- Scientific calculations
- Preparation of management information reports.

The word "immediate" above is used in the time frame of the electronics involved, typically microseconds or milliseconds. In a relatively simple system there may be only one interrupting device, such as the fire-detection system mentioned above. In this case the interrupt device could be connected into the CPU as shown in Figure 1-26(a). This is an example of a CPU with a single interrupt line which is nonvectored. Whenever the interrupt device needs service, it activates the interrupt request line. Such servicing entails setting aside the current program and the CPU machine state before addressing the interrupt device to determine the nature of the interrupt (requiring input or output, etc.). In the case of the fire detector, the type of service would be quite explicit;

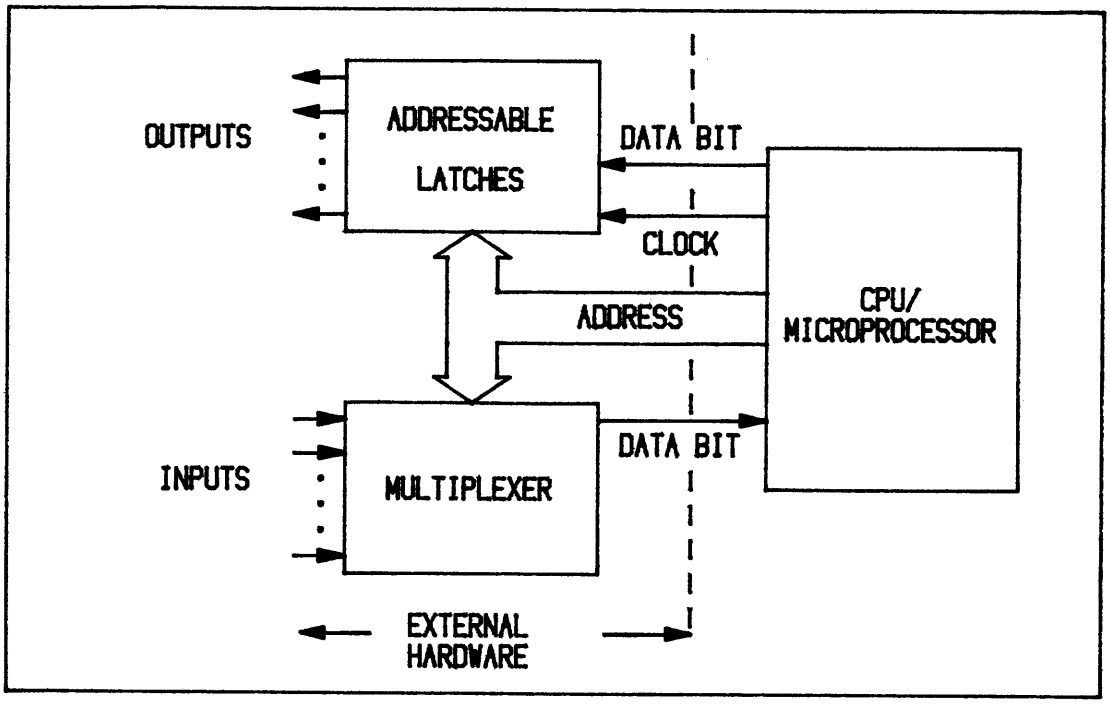


Figure 1-25. Single Bit I/O: Using External Hardware Devices

namely, to activate a sprinkler system and/or set off an alarm. Now suppose there are multiple interrupt devices such as multiple limit switches on a machine. The interrupt request line coming out of each of the interrupt devices is OR'ed with appropriate logic and tied into the single interrupt-request line going into the CPU. Thus, when any of the interrupt devices becomes active, a single request goes to the interrupt line. The CPU must resolve which of the devices requested service. It can handle this by means of polling. The requirement for the CPU to poll various devices to determine which one is active, however, can be eliminated by using a technique called vectored interrupt.

One type of vectored interrupt is indicated in Figure 1-26(b). Here, multiple-interrupt devices are shown connected directly to multilevel-interrupt request lines on the CPU. Built into the CPU is a priority encoder (PE). In the case of a single interrupt being active, the priority encoder merely supplies the corresponding code (vector) to the CPU so that the CPU is able to immediately identify and service that device without polling. In the case of several devices being active simultaneously, the priority encoder automatically provides the CPU with the code of the highest priority interrupt device. In the illustration, four interrupt devices are shown connected to the the multilevel-interrupt request line. The number of levels has to be restricted because, again, of pin limitations on the device itself.

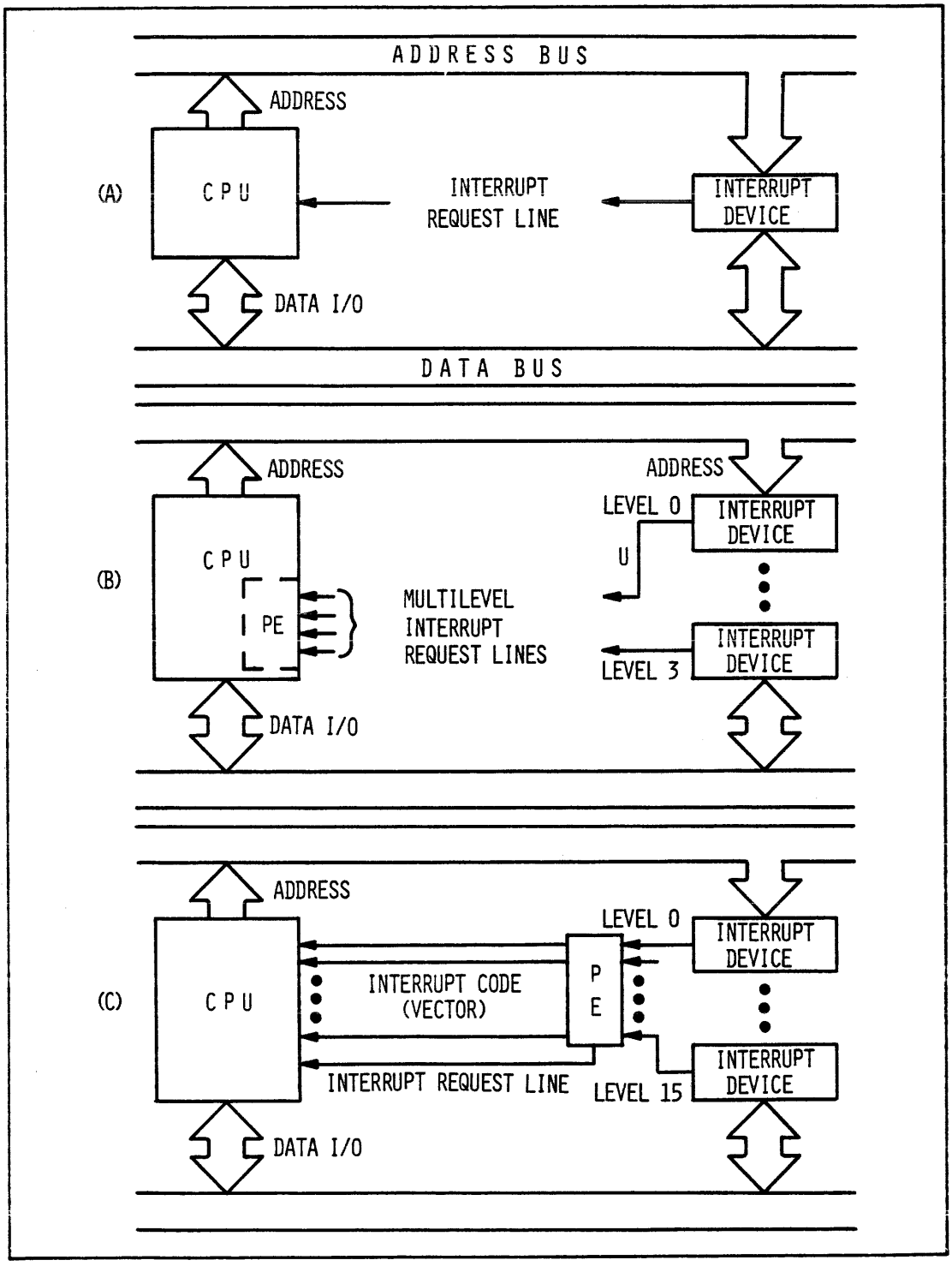


Figure 1-26. Three Types of Interrupt Capability

To get around the problem of pin limitations designers developed still another technique for handling multiple-interrupt devices as shown in Figure 1-26(c). This diagram depicts 16 interrupt devices connected into a block labled PE (priority encoder), which performs the same function as in the prior configuration. The priority encoder chip (or chips) automatically provides a binary encoded vector, four bits in this case, of the highest active priority interrupt device.

As one can see there has been a progression in interrupt capability moving from a single-interrupt request line to multilevel interrupt request lines, and then finally to provision for an interrupt code vector with all of the priority encoding logic being done externally. It should be noted in Figure 1-26(c) that even though there is only one interrupt request line the interrupt code vector consists of, typically, three or four lines to provide for eight to 16 priority-interrupt levels.

Whenever a CPU has interrupt capability it is also essential that it has some means of masking out such interrupt requests due to high-priority processing that it may be performing at the time. In the simplest case, an interrupt-enable latch is used. This latch, when set to one, allows interrupt requests to be fed into the CPU and serviced as has been discussed. However, when the interrupt-enable latch is cleared, the CPU will not recognize any interrupt requests active on the interrupt request line.

In the case of multiple-level interrupt request lines and systems using an interrupt code or vector, the CPU can have built into it a method of individually masking out a given interrupt level or masking out all interrupt requests having a lower priority than a predetermined level. This feature permits the programmer to determine at any given time in his program which particular levels of interrupt he will allow to be enabled and which would not be enabled. Such masking of interrupt levels is essential for orderly processing of complex real-time interrupt requests.

Direct Memory Access. Up to this point all the input/output discussed can be categorized as program controlled. It should be evident also that polling techniques for control of input/output are initiated under program control except in the case of those initiated by interrupt request. Direct memory access (DMA) is different from other forms of I/O in that it is completely independent from program control because it is initiated by the DMA device controller, and all transfers are made under its control.

In brief, the DMA device is allowed to get direct access to the memory so that the data being transferred to or from memory does not pass intermediately through the CPU as in the case of ordinary CPU-controlled I/O. A typical sequence of operations given with reference to Figure 1-27 follows.

- The DMA device (disk in this case) makes a request via the HOLD line for use of the three CPU buses.
- The CPU ceases execution when it has completed the current instruction and then issues a HOLD ACKNOWLEDGE signal to the DMA controller, while simultaneously releasing control of the address, data, and control buses.
- The machine state of the CPU is frozen during the sequence of operations to follow.
- The DMA controller sends the proper control signals (read/write, memory enable, etc.) directly to the memory and commences to place an address on the address bus and data on the data bus (to transfer it to memory), or prepares to receive data from the data bus as required.
- Such transfers can consist of only a single word or multiple words (block) as required in the system.
- When the transfer is completed, the DMA controller deactivates the HOLD line which causes the CPU to deactivate its HOLD ACKNOWLEDGE, thereby returning control to the CPU.
- Program execution continues with the instruction immediately following the last instruction executed by the CPU.

Special I/O Chips. To complement the rapid advancement made in microprocessor chips, manufacturers have developed special I/O chips which aid immeasurably in handling various input/output functions normally associated with processors of various types.

One of the most common chips is that called a UART, an acronym for Universal-Asynchronous-Receiver-Transmitter. This type of chip permits the designer to off-load from the CPU the task of timing and formatting asynchronous serial data that would ordinarily be transferred to or received from a terminal. Asynchronous means that each character code format contains its own start and stop bits utilized by the receiving device to achieve character decoding synchronization on a charater-by-character basis. Such an activity can consume a fair amount of initial software development time and also CPU execution time, the latter of which could be used to perform other tasks. Typical of the terminals that can be interfaced to a microprocessor using the UART chip are the Teletype Model 33 and the TI Silent 700 series. In addition, there are numerous other terminals that utilize the same 8-bit data character format, referred to as ASCII code (see Chapter 2).

Similar to the UART is the USART, the latter an acronym for Universal-Synchronous-Asynchronous-Receiver-Transmitter. This chip can do everything that the UART can do plus handle synchronous digital communication devices by means of additional timing logic built into the chip and controlled by software under programmer control. Synchronous means that the bits constituting each character are transmitted at a fixed clock rate without the need for start

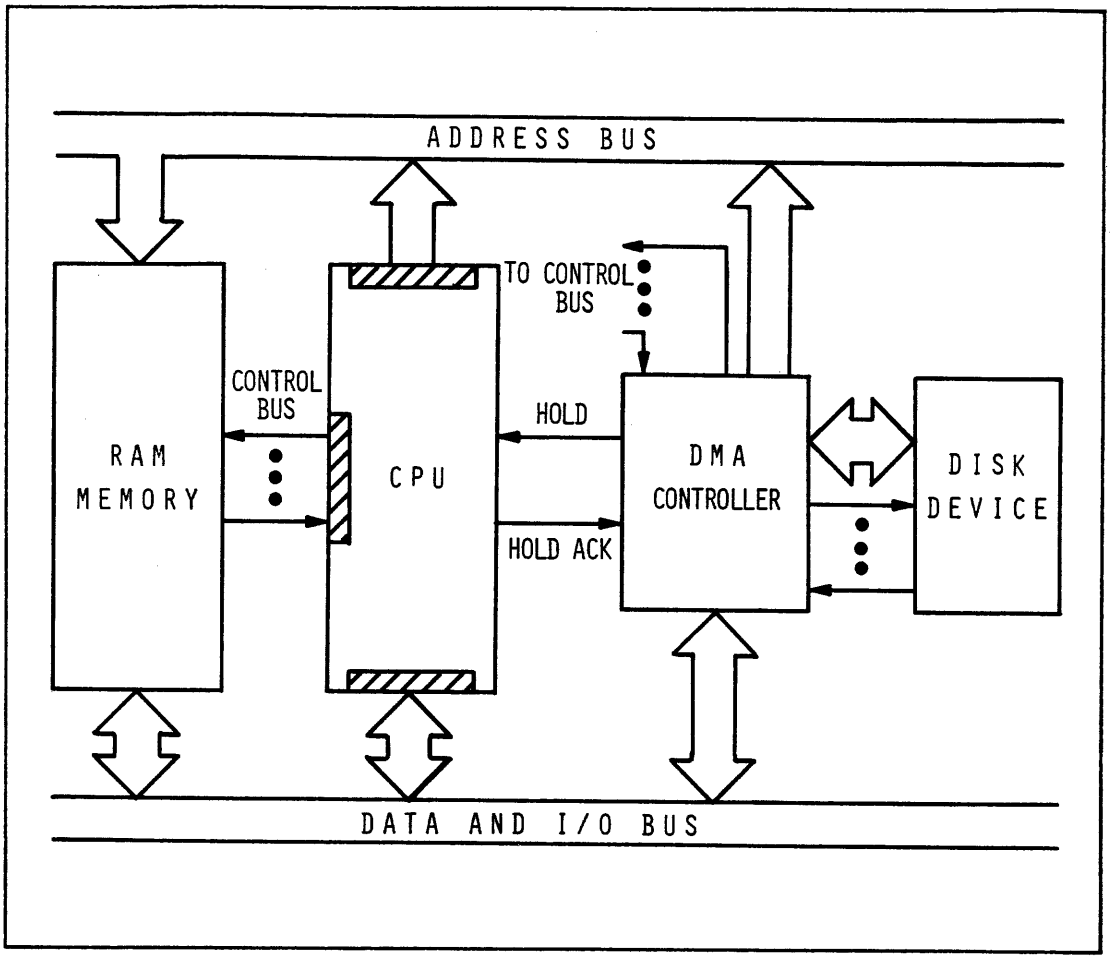


Figure 1-27. DMA Capability

and stop bits attached to each character code. The need for individual character synchronization is obviated by the use of a block- or frame-synchronization code at the beginning of the block of data.

Two additional LSI chips for I/O use are the PIO and DMA controller chips. The former is for Programmable-Input-Output. Such chips make it rather easy for the designer to design a bi-directional I/O port interface on his microprocessor. In addition, the PIO chip generally permits several pins to be programmed as interrupt inputs--generally multiple-level interrupts.

The DMA controller chip is designed to handle the special controller function discussed earlier (see Figure 1-27).

In general, all of these chips make the job of the designer much easier and reduce considerably the software housekeeping and overhead tasks that the CPU would have to perform if these specialized

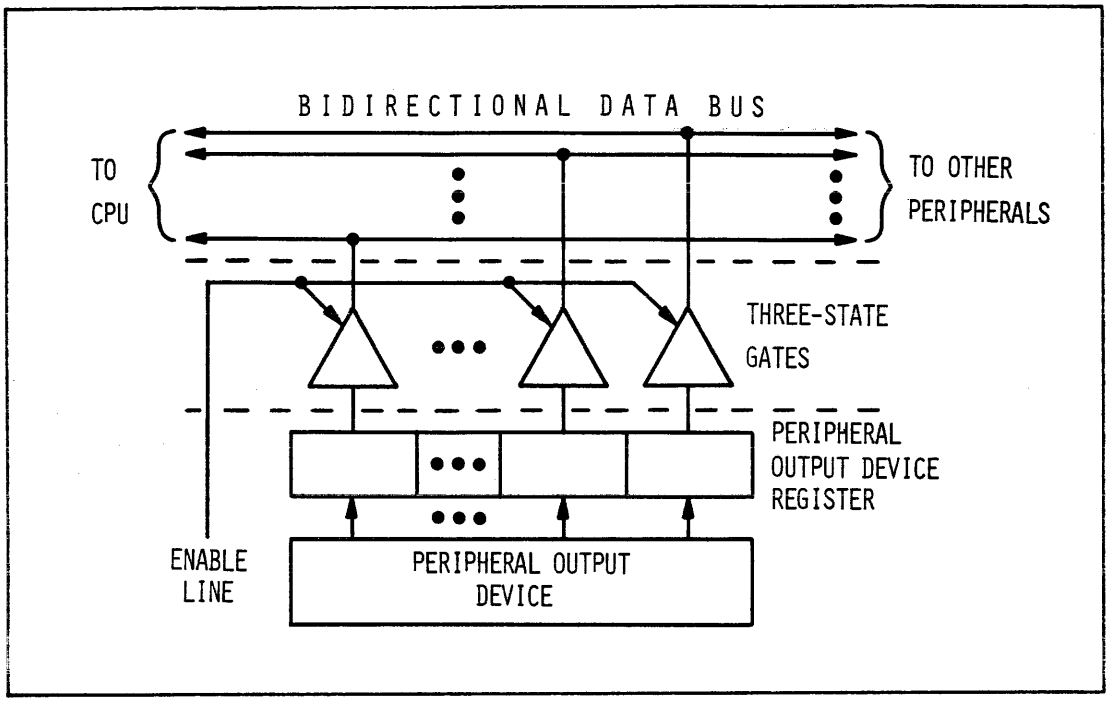


Figure 1-28. Example of Three-State Logic Devices

chips were not available. A more detailed discussion of several of these chips is given in Chapter 7 and in Appendix D.

Three-State Logic Devices. One of the continuing problems since the beginning of computers has been the need to attach multiple peripheral I/O devices onto a common data bus. The first breakthrough was that of utilizing collector logic. By using open collector devices, the designer could, under certain limitations, connect multiple devices onto a common bus as long as only one device was actively putting data on the bus at a time. Some limitations of this approach led to the development of what is called three-state logic.

In a three-state logic device, there are the regular states for ONE and ZERO plus an additional third state called the high-impedance state. An example of three-state logic gates tying a peripheral output-device register onto a data bus is given in Figure 1-28. Only when the enable line is active does the data in the peripheral output-device register appear on the data bus. When the enable is inactive, the output of the typical three-state gate is in a very high-impedance state--for all practical purposes disconnected from the data bus. There is more on this technique in Chapter 7.

1.5 TMS 9980A MICROPROCESSOR

All of the material in prior sections has been designed to introduce to the reader the concept of a stored-program digital computer, basic computer architecture and components, and the recent architectural enhancements in computers generally and microprocessors and microcomputers in particular. It is now appropriate to focus attention on the TMS 9980A microprocessor since it is the principal teaching vehicle used in this textbook.

Throughout this manual, "TMS 9980A" refers to the MP 9529 microprocessor, a version of the TMS 9980A specially selected for the TM 990/189.

General Description

In a 40-pin DIP package, the TMS 9980A single-chip microprocessor is instruction-set compatible with the 990 and 9900 family of minicomputers and microprocessors manufactured by Texas Instruments. The TMS 9980A has a 16-bit central-processing unit (CPU) but has a convenient 8-bit data bus, plus an on-chip clock. In general, the TMS 9980A features have been designed to minimize the system cost for smaller system applications as opposed to the TMS 9900 microprocessor which is in a 64-pin package and has a 16-bit data bus. The instruction set of the TMS 9980A is the same as that of the TMS 9900 and offers the full capability of a minicomputer in instruction power. One of the outstanding features of the 9900 family is the memory-to-memory architecture which permits multiple register files to be resident in memory, resulting in faster response to interrupts and, in general, improved programming flexibility. In addition to the chip itself, there is a compatible set of MOS and TTL memory and logic-function circuits that can be used with the TMS 9980A in various applications.

Additional features and characteristics of the TMS 9980A are

- Up to 16,384 bytes of addressable memory
- Six prioritized interrupts
- DMA I/O capability in addition to memory-mapped I/O
- Single-bit I/O by means of an internal Communications Register Unit (CRU).

It is stressed that the 16-bit word size of the TMS 9980A is one of the reasons for its powerful instruction set. At the same time, however, its data bus is only 8-bits wide, which means that to fetch a single-word instruction requires two memory read cycles. Here the trade-off of a longer time to read an instruction word versus the more cost-effective but smaller 8-bit data bus turned out to be attractive in view of reduced overall system cost using available memory and other associated MOS chips.

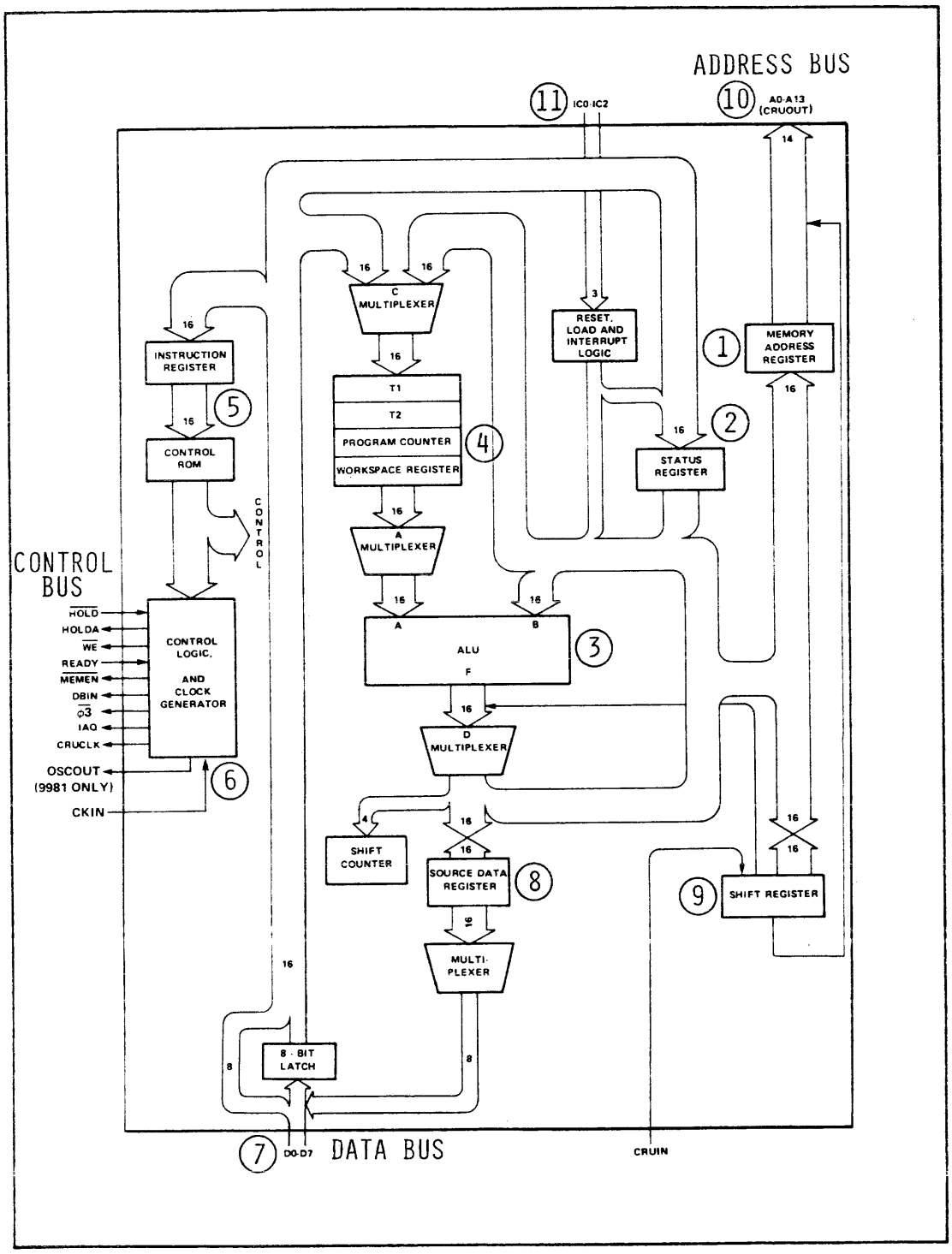


Figure 1-29. TMS 9980A Microprocessor Architecture

Architecture

Keeping in mind general architectural characteristics and some of the more recent enhancements, it is suitable now to examine the specific architecture of the TMS 9980A. The discussions in the next few paragraphs will be centered around Figure 1-29, a block diagram of the TMS 9980A.

In the upper right-hand corner of the diagram is found the memory address register (circle 1). It is shown that the output of the memory address register is restricted to 14 bits, thereby limiting the address capability to 16,384 bytes.

Shifting slightly down and to the left is the status register (circle 2). It contains the most recent status bits resulting from the prior ALU operation such as carry, overflow, and logical and arithmetic comparison bits (to be discussed later).

Moving further down and to the left is the ALU in the center of the diagram (circle 3). It has two ports going into it, the left one being controlled by multiplexer A. Above this multiplexer is the program counter (circle 4). Thus it is clear that the program counter is incremented or is modified by means of processing it through the ALU.

Moving over to the left side of the diagram and toward the top is the instruction register (circle 5). The instruction register holds the current instruction which is used to address the micro-program-control ROM shown below it. The block below the control ROM is that of the control logic and clock generator (circle 6).

The lines on the control bus that are going into and out of the control-logic block are each discussed below:

- HOLD -- used by the DMA controller to indicate to the CPU that a DMA transfer is requested.
- HOLDA -- the handshake signal used by the CPU to indicate to the DMA controller that the CPU is acknowledging the HOLD request.
- WE -- the write enable signal indicating that the CPU will be writing to memory or to a memory-mapped-output peripheral device.
- READY -- a handshake signal used by memory and input peripheral devices (memory-mapped) to indicate READY status.
- MEMEN -- memory enable which the CPU must activate in order to either read or write to the memory or to memory-mapped I/O devices.
- DBIN -- data bus in signal which, when active, indicates that the CPU will be reading in data on its data bus.
- Ø3 -- phase three of the clock generator.
- IAQ -- a signal indicating instruction acquisition fetch of the first word of the instruction.
- CRUCLK -- a signal for the CRU clock used to strobe

data out serially from the CRUOUT line.

° CKIN -

The data bus is located at the bottom of the diagram (circle 7). It is bi-directional and has an 8-bit latch which holds the first byte of a word fetched from memory. When the second byte arrives, then the two bytes are put together to form the 16-bit word used by the CPU. Similarly, data going out onto the data bus comes from the source data register (circle 8). Normally, the source data register would contain the result of the last ALU operation. To the right (circle 9) is the shift register used for CRU* (serial in, serial out) operations. Precise details on this operation are discussed in Chapter 6. Suffice it to say, special instructions are used to shift data in a bit at a time on the CRUIN line (bottom of the diagram), and similarly to shift it out on the CRUOUT line (upper right portion of the diagram).

The address bus (circle 10) consists of 14 pins, as mentioned earlier. Located to the left are the interrupt code lines (circle 11, IC0-IC2). When an external device desires to interrupt the CPU it will place its device interrupt code on these three lines. Additional details are furnished in Chapter 8.

It should be noted that there is no accumulator on the microprocessor chip itself. Below the program counter, however, there is a workspace pointer which is loaded with the address of register 0 (abbreviated by R0). One of the main characteristics of the entire 990/9900 family is that of workspace registers being located in memory space. The workspace pointer is used to point to register 0 in a block of 16 general-purpose registers. Additional blocks of 16 registers can be defined by changing the value in the workspace pointer.

Word/Byte Formats

All of the memory locations in the TMS 9980A memory space are addressable as 8-bit bytes. A word is defined as 16 bits (or two consecutive bytes) in memory starting with an even address. The most-significant half (8 bits) of a 16-bit word is located at the even address and the least-significant half of the word resides at the subsequent odd address. Since the TMS 9980A has both word and byte instructions, any byte at an even or odd address can be addressed by the different address modes inherent in the instruction set. Figure 1-30 depicts the typical word and byte formats utilized in the TMS 9980A. It should be noticed that the most-significant bit (MSB) of a word is labled as bit 0 while the least-significant bit (LSB) is labled with bit 15. Similarly, the MSB is bit 0 in a byte while the LSB is labled bit 7.

*CRU = Communications Register Unit

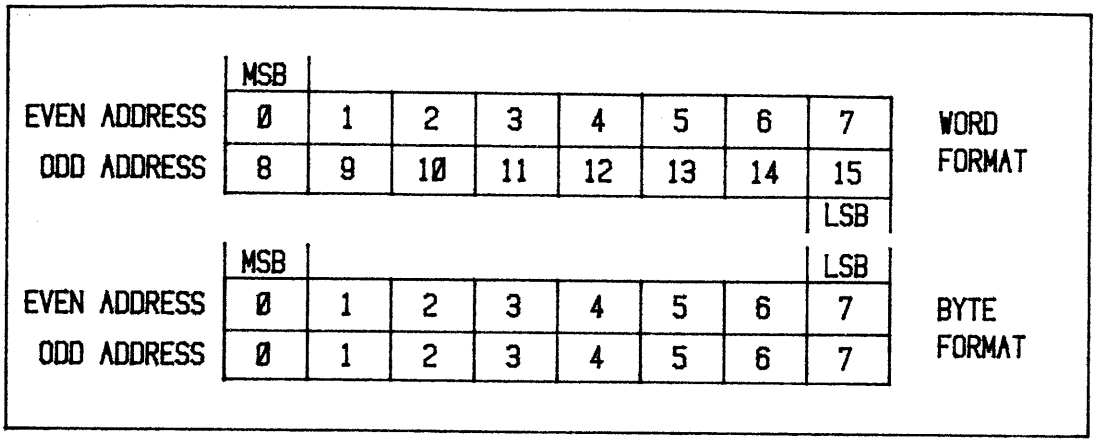


Figure 1-30. Word and Byte Formats (MSB=Sign Bit)

Memory Map

A memory map addressable by the TMS 9980A is shown in Figure 1-31. The first two words (addresses 0000 through 0003) are used for the RESET interrupt vector. In brief, whenever the CPU is reset, the CPU program-counter register is loaded with data bytes from locations two and three while the workspace pointer is loaded with the word in byte locations zero and one. As shown on the diagram, the interrupt vectors for levels one through four occupy memory addresses four through 13 (hexadecimal). Finally, the load-signal vector is located at the bottom of the diagram at the highest four bytes of memory.

On the TM 990/189 University Board, special on-board firmware (software programmed into PROM's) is contained at locations 0000_{16} to $07FF_{16}$. This special firmware contains the monitor which is used by the operator to enter machine code to inspect and modify memory and to perform other operations on the CPU in preparation for running or debugging a program. The symbolic assembler is used to assemble programs a line at a time as discussed in Chapter 4.

Workspace Registers

As mentioned earlier, the workspace pointer on the TMS 9980A CPU points to R0 of a block of 16 workspace registers. This is sometimes called a workspace-register file. Each workspace register can hold data or an address and can function as an operand register, an accumulater, an address register, or an index register (R0 cannot be used as an index register). The CPU can address any register in the workspace merely by adding a register displacement value from an instruction to the contents of the workspace pointer. The resulting value is then used to address the memory with a read or write request. In addition, a program can use as many of these

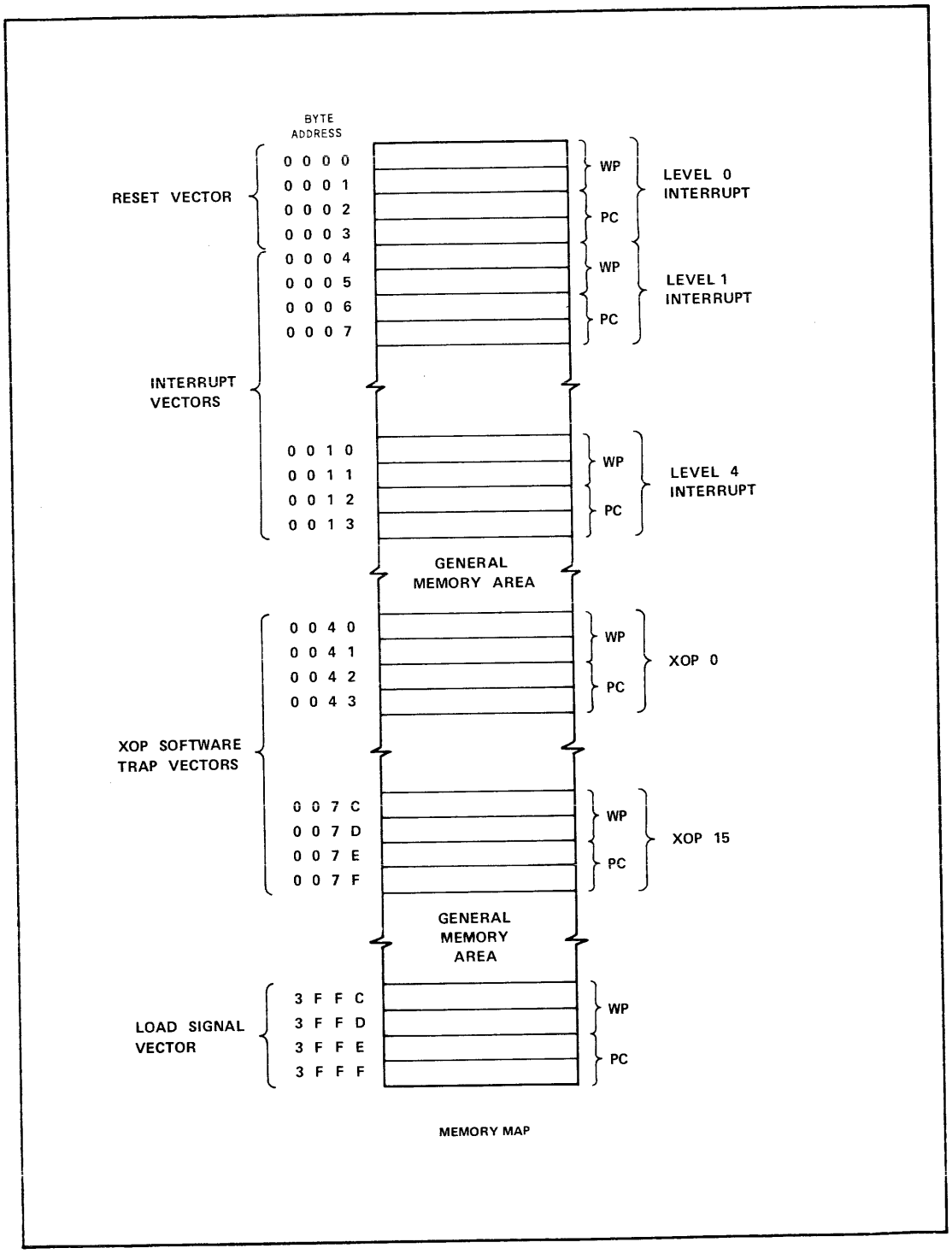


Figure 1-31. Memory Map

16-register blocks as required. In fact, the number of registers available to a program is limited only by the amount of memory.

The workspace-register file concept is particularly useful during operations that require a context switch, which is a change from one program environment to another. An interrupt or branch to a subroutine are examples of a context switch. Such an operation is depicted in Figure 1-32 where the program counter is shown to be pointing to the main program being executed out of the program A portion of the memory while the workspace pointer (WP) is pointing to register 0 of workspace A. When, say, an interrupt occurs, then the interrupt vector for the respective interrupt level is used to change the program counter to the value required for program B, (shown with a dotted line) while the workspace-pointer portion of the interrupt vector is used to change the workspace pointer to point to workspace B (shown also with a dotted line). At the same time, the original value of the program counter and workspace pointer are stored along with the status-register contents--all for program A--in the upper three registers of workspace B (R13, R14, and R15). In this manner, the programmer is not required to push onto stack or otherwise store the program counter and accumulators or registers in order to service the interrupt or to make a subroutine call. Thus, by exchanging the program counter, workspace pointer, and status register, the TMS 9980A is able to accomplish a complete context switch with six store cycles and six fetch cycles to memory. This is a considerable time saving over the more conventional approach used in the majority of other microprocessors to handle interrupt servicing and calls to subroutines. The context switch operation is described more fully in Chapter 8.

> Note: The status register and status bits are discussed in detail in Chapter 2 and later on in subsequent chapters dealing with the conditional jump instructions.

TM 990/189 Microcomputer Board

To acquaint the reader with the microcomputer board used as a teaching vehicle, reference is now made to Figures 1-33 and 1-34, a photograph and a layout of the TM 990/189 (University Board). This single-board microcomputer system was developed for use as a learning aid in the instruction of microcomputer fundamentals, machine- and assembly-language programming, and microcomputer applications and interfacing.

In addition to the TMS 9980A microprocessor, already described, this board contains a number of other component parts. These parts are described briefly in the following paragraphs.

Keyboard Display. This is the main on-board I/O peripheral which operates as an ASCII terminal. The keyboard with associated circuitry and monitor software is capable of generating the 87 most used ASCII characters using a 45-key pad. A 10-character display (7-segment

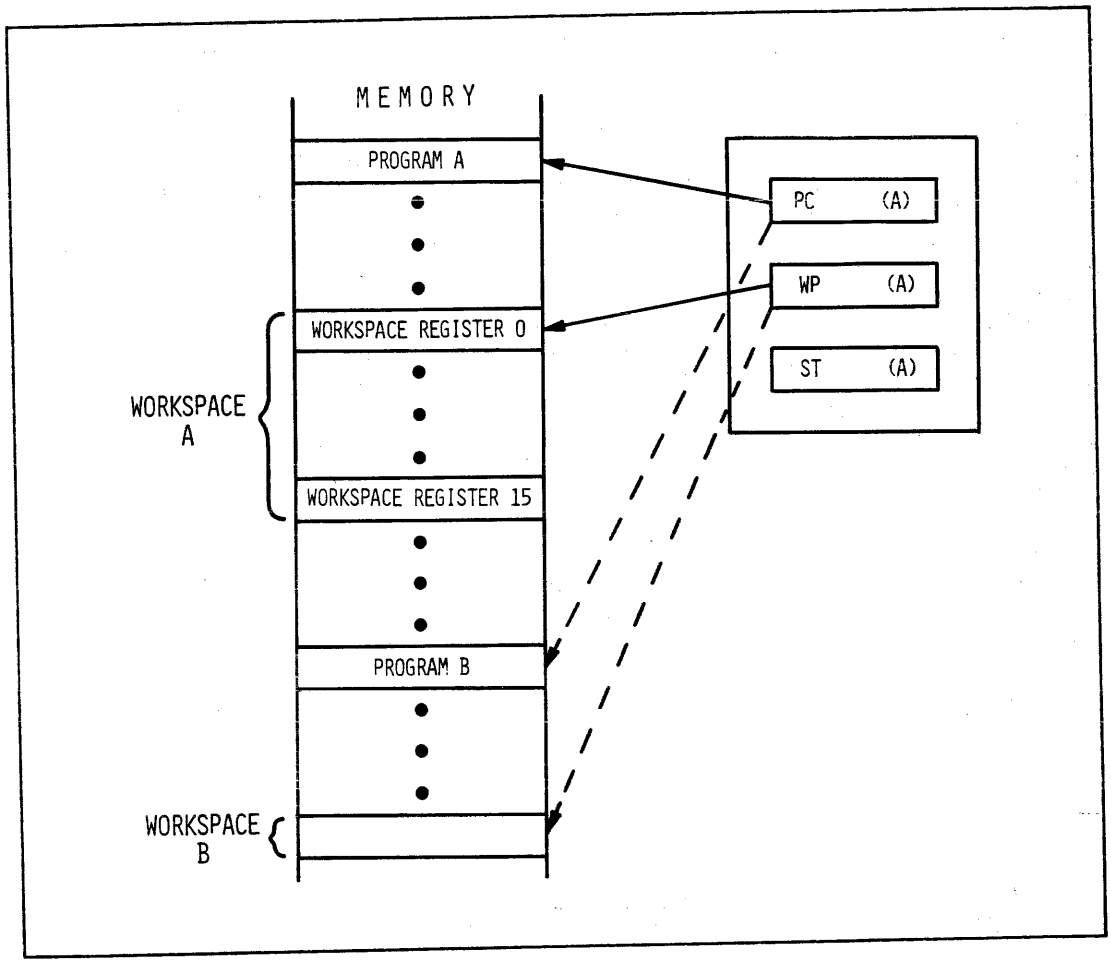


Figure 1-32. Context Switch

LED) is capable of displaying any nine contiguous characters of a maximum 64-character line (as if a typewriter-like terminal was used).

LED Display. Four LED display indicators are on board for general single-bit CRU output display and monitoring purposes under user program control. In addition, three other LED indicators are dedicated for monitoring specific functions under monitor firmware control.

Piezoelectric Speaker. A piezoelectric audio-output peripheral is available under user program control for generation of single tone sounds, music, key click, beep, etc.

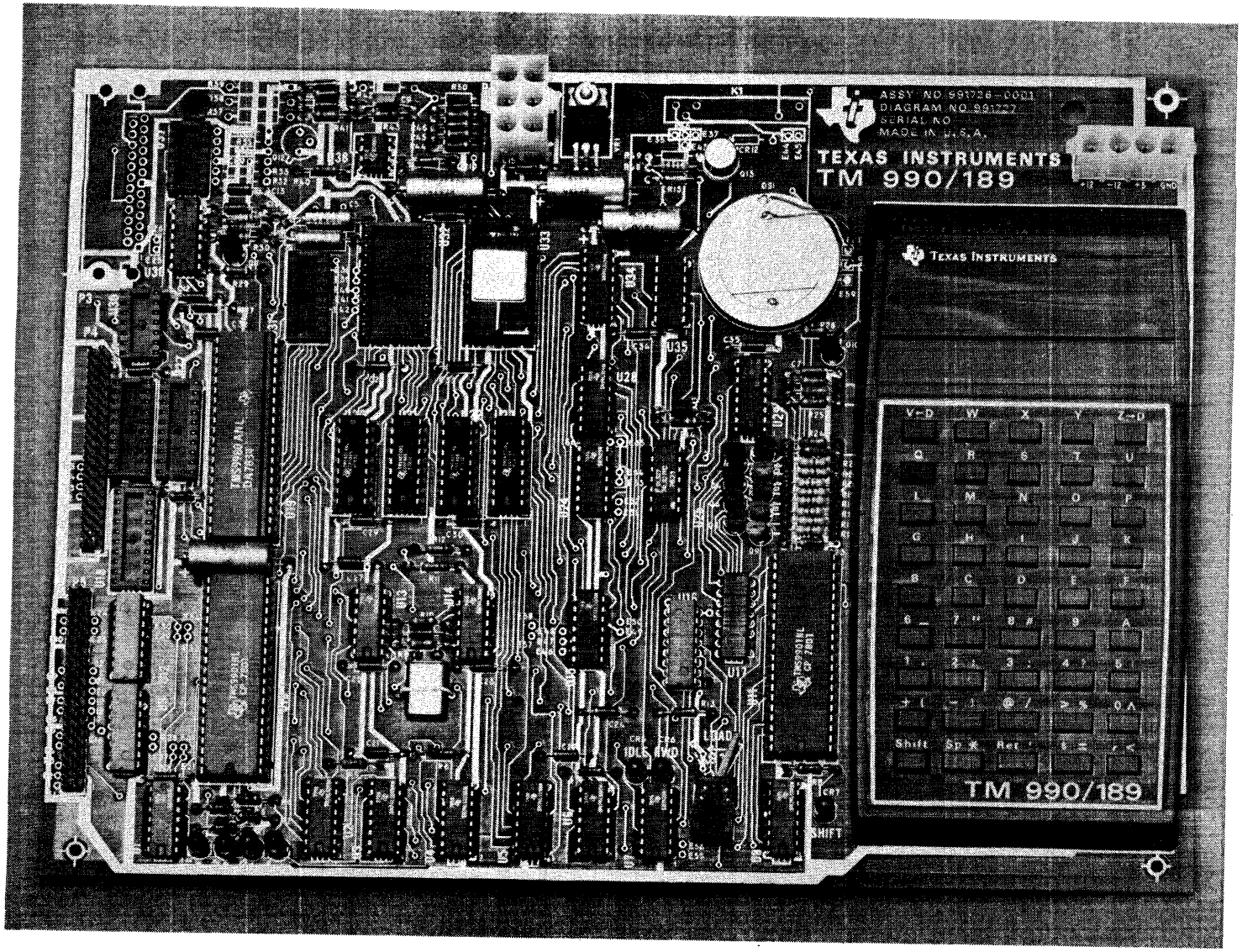


Figure 1-33. Photograph of TM 990/189 University Board

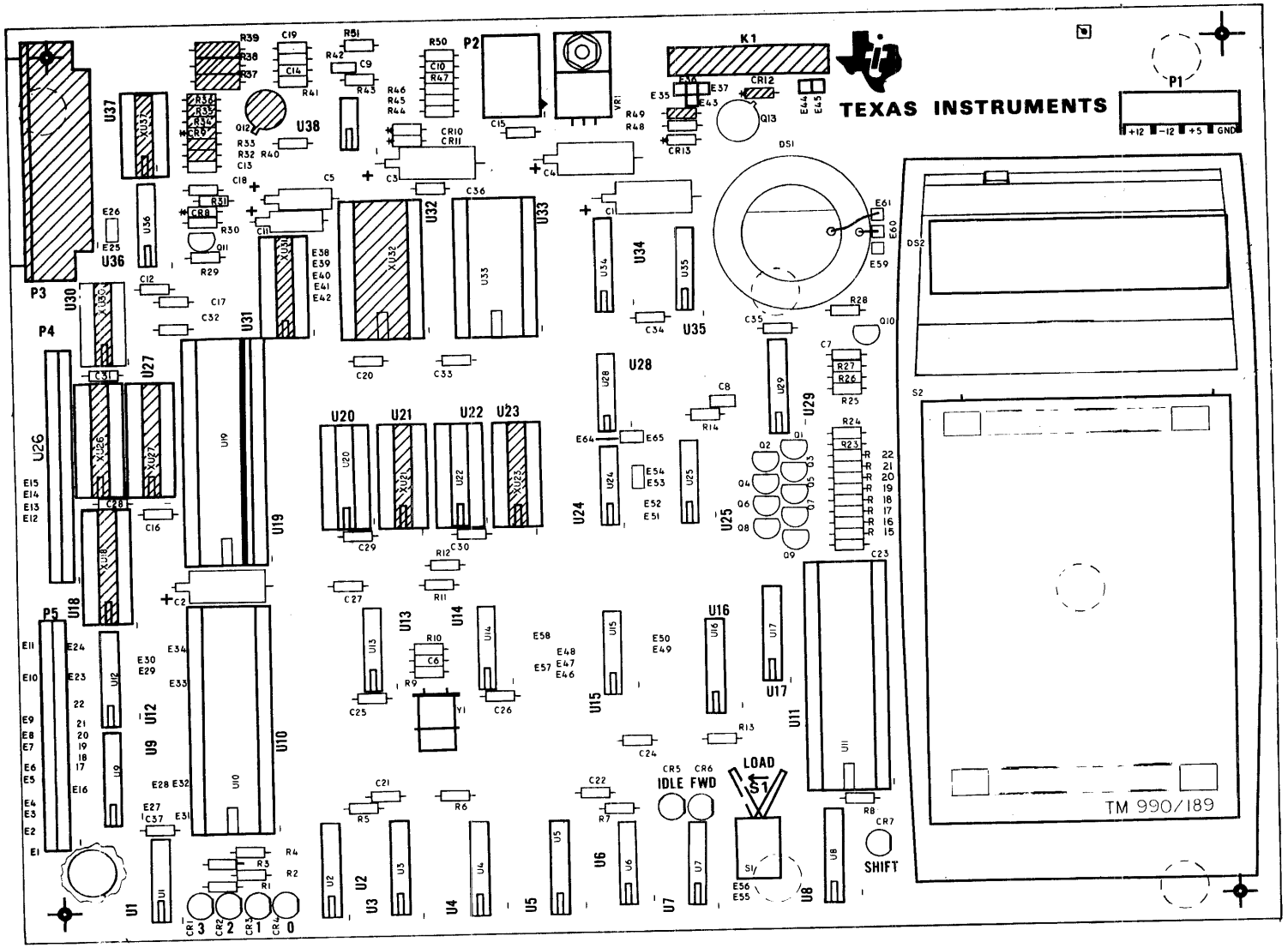


Figure 1-34. Layout of Components on the TM 990/189 University Board

Audio Cassette Interface. A standard audio cassette recorder/reproducer can be connected to the microcomputer for dumping programs and data from memory in a format compatible with the TM 990/302 Software Development Board. Similarly, programs and data stored on cassette can be loaded into memory. Both of these operations are handled by means of monitor keyboard commands. Space and circuitry is available on the board for deck-control relay mounting and operation.

EIA Interface. A standard EIA RS-232 interface can be added at the user's option if an external EIA terminal is desired in place of the on-board microterminal.

Bus Connector. A 40-pin connector is provided for attaching external cards or devices to the microcomputer address, data, and control buses. This can be used for memory or parallel I/O expansion.

I/O Expansion Connector. A 40-pin connector is provided to permit expansion of the CRU I/O capability.

Memory. The board contains 4K bytes of ROM/EPROM, which holds the UNIBUG monitor and symbolic assembler as firmware. There is expansion space for 2K additional bytes of ROM/EPROM for user-supplied application firmware. The board has 1K bytes of RAM, expandable by the user to 2K bytes.

Extended discussions of these component parts and the operation of the TM 990/189 microcomputer board are covered in later chapters as follows.

Chapter 2: On-board terminal and UNIBUG monitor operation
Chapter 4: Symbolic assembler
Chapter 5: Memories
Chapter 6: CRU I/O port and audio cassette interface
Chapter 7: TMS 9901 and TMS 9902 peripheral components
Chapter 8: UNIBUG user-accessible utilities.

1.6 SUMMARY

This chapter has introduced the reader to the field of computers in general and the rapidly emerging field of microprocessors and microcomputers in particular. The discussion has been limited to selected topics which are designed to set the stage for an in-depth coverage of subjects in subsequent chapters.

The principal building blocks of computers were discussed, namely, the components of the CPU, memory, and input/output. To more fully understand computer operation, a program example was used to illustrate the step-by-step process of executing a sequence of instructions to achieve a given task.

A number of architectural enhancements available in recent years were also covered. The importance of multiple accumulators, the index register, the concept of the workspace pointer and workspace register file, the stack and microprogram control were stressed. Some of the features and characteristics of ROM, PROM, EPROM, and RAM memory chips were reviewed. With regard to input/ output, the single-bit I/O concept along with multiple level, vectored and prioritized interrupts, direct memory access, special I/O chips, and three-state logic devices were also discussed.

Finally, the TMS 9980A microprocessor was described along with the TM 990/189 microcomputer board...the latter being the teaching vehicle used throughout the book for examples and exercises.

CHAPTER 2

ARITHMETIC, LOGIC, AND THE ALU

2.1 INTRODUCTION

In this chapter the reader is reacquainted or, perhaps introduced to the various number systems pertinent to the field of microprocessors. Other number bases are explained by comparing them with the more familiar decimal, or base-ten, system. This chapter can enable a person to become proficient in the translation and manipulation of decimal-, binary-, octal-, and hexadecimal-based numbers. The special number codes such as BCD (binary coded decimal) and ASCII (American Standard Code for Information Interchange) are also presented.

The reader will use the foundation gained in number systems and other digital codes as a basis for understanding the concept of an ALU (arithmetic logic unit). The ALU circuitry can be called the "brain" of the microprocessor. It can perform the four basic arithmetic functions (addition, subtraction, multiplication, and division) as well as Boolean logic operations. The general flow of binary input data, processing, and the output from the ALU will be covered in this chapter.

The last portion of the chapter is devoted to the description and operation of the University Board terminal and UNIBUG monitor. Wherever possible, the terminal's usage is related to the previously discussed material.

2.2 NUMBER SYSTEMS

This section describes the most common number systems or codes encountered when dealing with processors. By understanding the foundations of the decimal number system and the concept of positional notation, the reader will be in a position to understand the other number systems and codes including binary, octal, hexadecimal, BCD, and ASCII.

Decimal Number System

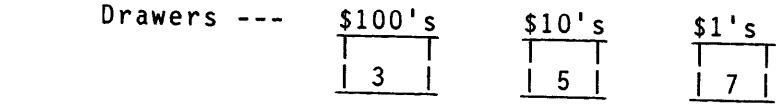
Decimal digits are usually the first numeric values that are encountered. There are ten digits in the decimal number system; therefore, it can be said the decimal number system has a base or radix of ten. The digits zero through nine are used to represent

the ten values in the system. The maximum-value digit, nine, is one less than ten, the base. Similarly, binary (base 2), octal (base 8), and hexadecimal (base 16) systems have as their maximum-value digits 1, 7, and 15, respectively. It is shown later that the value 15 is indeed a single digit in the hexadecimal system. Whenever the number base is not implicitly known, a subscript (equal to the base) commonly follows the right-most digit of the number. For example, 365_{10} indicates a base 10 (decimal) number.

The decimal number system illustrates the importance of a digit's position within a number. Although terms like units, tens, hundreds, and thousands are used, positional notation is the underlying concept. The table below illustrates the relationship between those terms and the powers of base ten.

$10^0 = 1$ = units (any base with zero exponent is one)
$10^1 = 10$ = tens
$10^2 = 100$ = hundreds
$10^3 = 1{,}000$ = thousands
$10^4 = 10{,}000$ = ten-thousands
$10^5 = 100{,}000$ = hundred-thousands.

Positional notation can be explained using the cash register drawer analogy. Suppose an item in a store costs $357 and the clerk is handed a certain combination of bills. After the transaction, a previously empty drawer might appear as below.



Although the price tag used the more familiar shorthand notation, 357, the cost can be written in positional notation:

[3x(hundreds)] + [5x(tens)] + [7x(ones)] or
using the power of ten table,
$[3x(10^2)] + [5x(10^1)] + [7x(10^0)]$ or
once again, by evaluation,
[300] + [50] + [7] = 357.

It can be seen that the digit's position within a number determines its importance or weight. In this example, three (3) is called the most-significant digit (MSD) and seven (7) is the least-significant digit (LSD).

Binary Number System

Early mechanical calculating machines used decade (ten state) gears to perform arithmetic operations. Digital computers, including the microprocessor, use high-speed logic switching circuits. Like a light switch, two basic states prevail in its logic circuitry:

on and off. There is a number system which has only two digits or states: the binary system. This number system contains the digits zero and one. Zero can be used to represent off and one to represent on.

Each digit in the binary number system is called a bit. The term "bit" is compounded from "binary" and "digit." When working with a microprocessor, it is often necessary to refer to the number of bits in a binary number. Three of the more common bit-group names are nibbles, bytes, and words. A nibble consists of four bits of data (half a byte) and has a maximum decimal value of 15. A byte contains eight bits of data and has a maximum decimal value of 255. A word can be any number of bits, since it is usually a characteristic of the computer being discussed. For instance, an eight-bit microprocessor's word size is a byte. A minicomputer typically has a word size of 16 bits.

As with the decimal system, the digits (bits) in the binary system have positional importance within the number. Except for the different base, the binary positional-notation scheme is the same as that of decimal numbers. The first nine positive entries in the binary system are listed in Table 2-1.

Table 2-1. Powers of Two

2^0	$= 1_{10}$	$= 1_2$
2^1	$= 2_{10}$	$= 10_2$
2^2	$= 4_{10}$	$= 100_2$
2^3	$= 8_{10}$	$= 1000_2$
2^4	$= 16_{10}$	$= 10000_2$
2^5	$= 32_{10}$	$= 100000_2$
2^6	$= 64_{10}$	$= 1000000_2$
2^7	$= 128_{10}$	$= 10000000_2$
2^8	$= 256_{10}$	$= 100000000_2$

In binary notation, a nibble of alternate ONE's and ZERO's is written as 1010_2. This same binary number can be expanded into positional-notation form as follows.

From the 'powers of two' table,
$[1 \times (2^3)] + [0 \times (2^2)] + [1 \times (2^1)] + [0 \times (2^0)]$, or
using decimal evaluation,
$[1 \times (8)] + [0 \times (4)] + [1 \times (2)] + [0 \times (1)]$, or
$[8] + [0] + [2] + [0] = 10_{10}$.
Hence, $1010_2 = 10_{10}$.

When the binary number (1010_2) is evaluated in positional notation, it has a value equal to decimal ten. The same approach can be taken to find the equivalent decimal value for any other base number. In the previous example, the most-significant bit's (MSB) weight is two cubed (2^3) or eight. The least-significant bit's (LSB) weight is (2^0) or one.

In reverse fashion, a whole decimal value can be converted to a binary number through a series of divisions by the base two. Each time the base is divided into the decimal number, the remainder is recorded and the new quotient is used for a subsequent division by the base. This process continues until the dividend is zero. The first remainder represents the least-significant bit and the last remainder represents the most-significant bit.

The procedure, or "algorithm", for the conversion is given below.

B	D	
2	10	R
2	5	0
2	2	1
2	1	0
2	0	1

LSD

MSD

B = Base
D = Dividend
R = Remainder

Hence, $10_{10} = 1010_2$.

Octal Number System

Octal based numbers are also used in the computer field. This system's base or radix is eight and contains the digits zero through seven. These digits are the same as those in the decimal system except, of course, their weights are based on eight rather than ten. The first five positive entries in the powers of eight table are listed in Table 2-2.

Table 2-2. Powers of Eight

$$8^0 = 1_{10} = 1_8$$
$$8^1 = 8_{10} = 10_8$$
$$8^2 = 64_{10} = 100_8$$
$$8^3 = 512_{10} = 1000_8$$
$$8^4 = 4096_{10} = 10000_8$$

As an example, an octal number such as 704_8 can be written in shorthand notation. The same octal number can be expanded into positional-notation form:

From the powers of eight table,
$[7 \times (8^2)] + [0 \times (8^1)] + [4 \times (8^0)]$ or
using decimal evaluation,
$[7 \times (64)] + [0 \times (8)] + [4 \times (1)] =$
$[448] + [0] + [4] = 452_{10}$.
Hence, $704_8 = 452_{10}$.

Conversion of a decimal number to octal can also be accomplished with the algorithm previously discussed:

B	D	
8	452	R
8	56	4
8	7	0
8	0	7

LSD

MSD

B = Base
D = Dividend
R = Remainder

Hence, $452_{10} = 704_8$.

Conversions from octal to binary numbers are quite simple because an octal digit can be represented by three bits. For example,

$$704_8 = (111)_2\ (000)_2\ (100)_2$$
$$= 111000100_2.$$

Conversely, a binary number such as 110101111 can be regrouped into an octal number equivalent:

$$110101111_2 = (110)_2\ (101)_2\ (111)_2 = 657_8.$$

Hence, $657_8 = 110101111_2$.

It should also be apparent from this example why the octal scheme is sometimes used for binary representation. Three octal digits are easier to remember and to work with than a nine-digit binary number. The octal number system is used by some computer manufacturers to represent instruction codes and other binary data.

Hexadecimal Number System

Hexadecimal is perhaps the most common number system used with microprocessors and computers in general. The system's base or radix is 16. It contains the values zero through 15. The digits 0 through 9 have the same value as those in the decimal system except that their positional weights are based on powers of 16. The values 10 through 15 are represented by the alphabetic characters A through F. The relationship between hexadecimal and decimal numbers is shown in Table 2-3.

NOTE: The " > " sign indicates hexadecimal (e.g., $>F = 15_{10}$).

Table 2-3. Hexadecimal and Decimal Equivalent

$0_{16} = 0_{10}$	$8_{16} = 8_{10}$
$1_{16} = 1_{10}$	$9_{16} = 9_{10}$
$2_{16} = 2_{10}$	$A_{16} = 10_{10}$
$3_{16} = 3_{10}$	$B_{16} = 11_{10}$
$4_{16} = 4_{10}$	$C_{16} = 12_{10}$
$5_{16} = 5_{10}$	$D_{16} = 13_{10}$
$6_{16} = 6_{10}$	$E_{16} = 14_{10}$
$7_{16} = 7_{10}$	$F_{16} = 15_{10}$

The first four entries in the powers of 16 table are shown in Table 2-4.

Table 2-4. Powers of Sixteen

$$16^0 = 1_{10} = 1_{16}$$
$$16^1 = 16_{10} = 10_{16}$$
$$16^2 = 256_{10} = 100_{16}$$
$$16^3 = 4096_{10} = 1000_{16}$$

In shorthand notation, an example hexadecimal number is written as $1BC_{16}$. However, the same hexadecimal number can be expanded in positional notation as follows.

From the 'powers of 16' table,
$[1 \times (16^2)] + [B_{16} \times (16^1)] + [C_{16} \times (16^0)]$ or

using decimal evaluation,
$[1 \times (256)] + [11 \times (16)] + [12 \times (1)] =$
$[256] + [176] + [12] = 444_{10}$.

Hence, $1BC_{16} = 444_{10}$.

Applying the same algorithm to convert a decimal number to a hexadecimal value produces the following.

B	D	R
16	444	
16	27	$12=C_{16}$
16	1	$11=B_{16}$
16	0	1

LSD
|
MSD

B = Base
D = Dividend
R = Remainder

Hence, $444_{10} = 1BC_{16}$.

Conversions from hexadecimal to binary numbers are simple because 16 combinational values can be represented by a nibble or four bits. This relationship enables each hexadecimal digit to be expressed as a four-bit binary grouping. The example below demonstrates how quickly the hexadecimal value F17 can be converted to its binary equivalent.

$$F17_{16} = (1111)_2 \ (0001)_2 \ (0111)_2.$$

Note that leading zeroes are used to fill the four-bit requirement when hexadecimal digits are less than eight.

Conversely, a binary number such as 1001011 can be regrouped, beginning with the least-significant bit and preceding toward the most-significant bit, into the equivalent hex number 4B:

$$1001011_2 = (0100)_2 \ (1011)_2,$$
$$= 4B_{16}.$$

Thus, the hexadecimal system affords the user yet another convenient alternative to strings of binary digits.

Fractional Numbers

In order to simplify the presentation on base conversions, only integer numbers in the various systems are explained. However, just as fractions exist in the decimal system (e.g., 3.125), they also exist in the other number systems as well.

For instance, the positional notation for 3.125_{10} can be written:

$$[3 \times (10^0)] + [1 \times (10^{-1})] + [(2 \times 10^{-2})] + [5 \times (10^{-3})] =$$
$$[3] + [1/10^1] + [2/10^2] + [5/10^3] =$$
$$[3.000] + [.100] + [.020] + [.005] = 3.125.$$

Converting this fractional decimal number to another base requires two steps. First, the integer portion, 3, can be converted using the algorithm already discussed. Secondly, the fractional portion, .125, can be converted by successively multiplying the fraction by the given base. As integers result in the products, they are recorded.

The fraction's conversion from decimal to binary is demonstrated below.

B	DF	P	I
2	.125	0.250	0
2	.250	0.500	0
2	.500	1.000	1

MSD

LSD

B = Base
DF = Decimal Fraction
P = Product
I = Integer

Hence, $3.125_{10} = 11.001_2$.

The multiplication process of base times a fractional number continues until the product contains all zeroes to the right of the decimal (indicating an exact binary equivalent) or until the desired precision is obtained.

Converting a different base fractional or mixed number to a decimal value is similar to the process used earlier. As an example, the binary fraction 1011.0101_2 will be converted into its decimal equivalent.

Taking the bits to the left of the binary point and converting:

$$[1 \times (2^3)] + [0 \times (2^2)] + |1X(2^1)|$$
$$+ [1 \times (2^0)] =$$
$$[8] + [0] + [2] + [1] = 11_{10}$$

Taking the bits to the right of the binary point and converting:

$$.[0 \times (2^{-1})] + [1 \times (2^{-2})] + [0 \times (2^{-3})] + [1 \times (2^{-4})] =$$
$$.[0 \times (1/2)] + [1 \times (1/4)] + [0 \times (1/8)] + [1 \times (1/16)] =$$
$$.[0] + .[25] + [0] + .[0625] = .3125_{10}.$$

Hence, $1011.0101_2 = 11.3125_{10}$.

Table 2-5 below shows a comparison between several fractional values in the hexadecimal, binary, and decimal number systems.

Table 2-5. Fraction Comparison Table

Hexadecimal	Binary	Decimal
0.01	0.00000001	0.00390625
0.02	0.0000001	0.0078125
0.03	0.00000011	0.01171875
0.04	0.000001	0.015625
0.05	0.00000101	0.01953125
0.06	0.0000011	0.0234375
0.07	0.00000111	0.02734375
0.08	0.00001	0.03125
0.09	0.00001001	0.03515625
0.0A	0.0000101	0.0390625
0.0B	0.00001011	0.04296875
0.0C	0.000011	0.046875
0.0D	0.00001101	0.05078125
0.0E	0.0000111	0.0546875
0.0F	0.00001111	0.05859375
0.1	0.0001	0.0625

Note that just as with decimal numbers, a period separates the whole (or integer portion) from the fractional part. These periods are called hexadecimal point, binary point, and decimal point, depending upon the number system in which they appear.

Binary Coded Decimal

Since many digital applications use decimal digits, the need to convert binary codes to decimal conveniently gave rise to the "binary-coded decimal" (BCD) system. The 8-4-2-1 weights for binary digits lend themselves to easy transition into the decimal digits zero through nine.

The decimal value 789 may be converted to BCD in the following manner.

$$7_{10} = [0 \times (8)] + [1 \times (4)] + [1 \times (2)] + [1 \times (1)] = 0111_2$$
$$8_{10} = [1 \times (8)] + [0 \times (4)] + [0 \times (2)] + [0 \times (1)] = 1000_2$$
$$9_{10} = [1 \times (8)] + [0 \times (4)] + [0 \times (2)] + [1 \times (1)] = 1001_2.$$

Hence, $789_{BCD} = (0111)_2 \quad (1000)_2 \quad (1001)_2$.

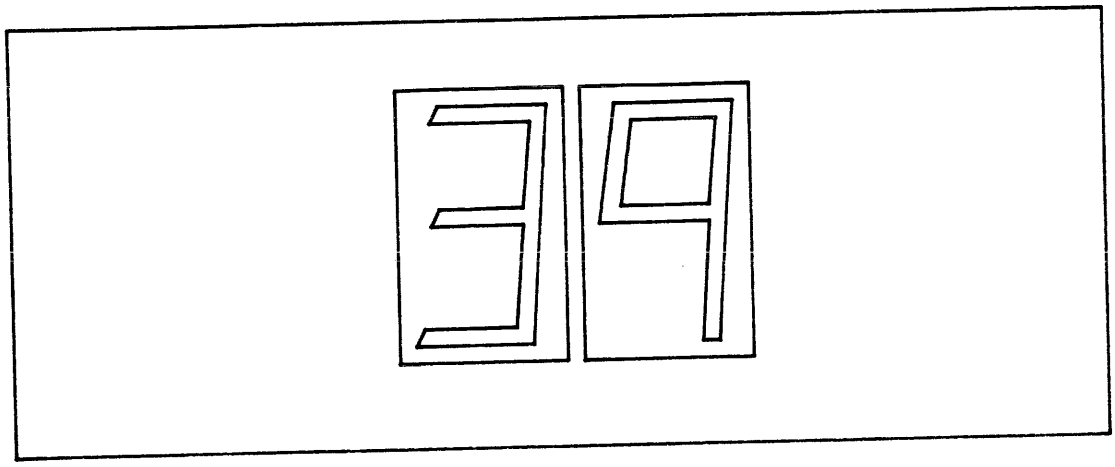


Figure 2-1. Two Digit Decimal Readout

The binary-coded value 10001100011 can be converted to decimal by grouping each four bits from the least-significant to the most-significant bit as follows.

$$(0100)_2 \quad (0110)_2 \quad (0011)_2 = 463_{BCD}.$$

As a practical illustration, suppose an instrument panel requires two decimal digits for readout. One approach would be to mount two numeric displays as illustrated in Figure 2-1. An alternative approach might be to mount two vertical rows of lamps as shown in Figure 2-2. Note that in both instances the decimal value 39 is displayed.

A byte of data in binary code equals a maximum decimal value of 255. A byte of BCD data is less efficient since it can represent a maximum decimal value of only 99. Table 2-6 below shows further relationships.

Table 2-6. Relationship of Decimal, BCD, and Binary Values

Decimal	BCD	Binary
0	0000 0000	0000
1	0000 0001	0001
2	0000 0010	0010
3	0000 0011	0011
4	0000 0100	0100
5	0000 0101	0101
6	0000 0110	0110
7	0000 0111	0111
8	0000 1000	1000
9	0000 1001	1001
10	0001 0000	1010
11	0001 0001	1011
12	0001 0010	1100
13	0001 0011	1101
14	0001 0100	1110
15	0001 0101	1111

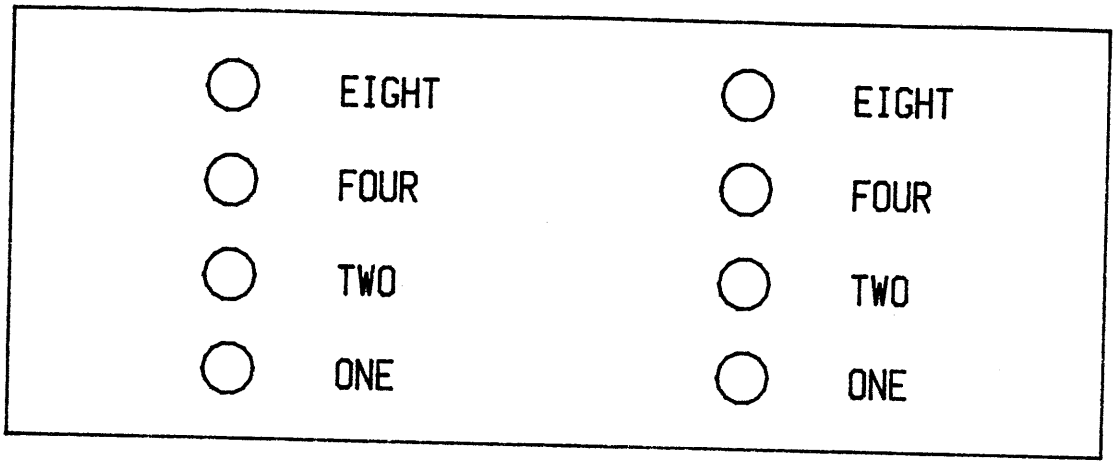


Figure 2-2. Two Digit BCD Readout

Table 2-6 also illustrates the simplicity with which decimal numbers can be converted to BCD. However, the binary-to-BCD conversion is not as direct. A binary number must first be converted to decimal and then translated into BCD, as in the next example.

$$110.01_2 = 6.25_{10} = (0110.0010\ 0101)_{BCD}.$$

ASCII Code

There are other forms of binary code which represent alphabetic characters and numbers. Three of the most common alphanumeric codes are ASCII, BAUDOT, and EBCDIC. However, only the ASCII (American Standard Code for Information Interchange) code will be described here. The ASCII code is widely used in microprocessor systems to communicate with peripherals and other microprocessors systems.

Within the ASCII code system, there exists the six-bit (2^6) and seven-bit (2^7) codes. The six-bit code contains 64 characters encompassing the upper-case alphabet, the decimal numbers 0 through 9 and other special characters. The seven-bit code contains 128 characters, including the lower-case alphabet and additional special characters. The special characters are used for punctuation and control. The seven-bit ASCII code is shown in Table 2-7. Control character abbreviations are listed and explained following the table.

Table 2-7. The 7-Bit American Standard Code for Information Interchange

*ASCII CHARACTER CODE

CHARACTER	BINARY CODE	HEXADECIMAL CODE	CHARACTER	BINARY CODE	HEXADECIMAL CODE
Space	010 0000	20	P	101 0000	50
!	010 0001	21	Q	101 0001	51
" (dbl. quote)	010 0010	22	R	101 0010	52
#	010 0011	23	S	101 0011	53
$	010 0100	24	T	101 0100	54
%	010 0101	25	U	101 0101	55
&	010 0110	26	V	101 0110	56
' (sgl. quote)	010 0111	27	W	101 0111	57
(	010 1000	28	X	101 1000	58
)	010 1001	29	Y	101 1001	59
* (asterisk)	010 1010	2A	Z	101 1010	5A
+	010 1011	2B	[	101 1011	5B
, (comma)	010 1100	2C	\	101 1100	5C
- (minus)	010 1101	2D	]	101 1101	5D
. (period)	010 1110	2E	^	101 1110	5E
/	010 1111	2F	_ (underline)	101 1111	5F
0	011 0000	30	`	110 0000	60
1	011 0001	31	a	110 0001	61
2	011 0010	32	b	110 0010	62
3	011 0011	33	c	110 0011	63
4	011 0100	34	d	110 0100	64
5	011 0101	35	e	110 0101	65
6	011 0110	36	f	110 0110	66
7	011 0111	37	g	110 0111	67
8	011 1000	38	h	110 1000	68
9	011 1001	39	i	110 1001	69
:	011 1010	3A	j	110 1010	6A
;	011 1011	3B	k	110 1011	6B
<	011 1100	3C	l	110 1100	6C
=	011 1101	3D	m	110 1101	6D
>	011 1110	3E	n	110 1110	6E
?	011 1111	3F	o	110 1111	6F
@	100 0000	40	p	111 0000	70
A	100 0001	41	q	111 0001	71
B	100 0010	42	r	111 0010	72
C	100 0011	43	s	111 0011	73
D	100 0100	44	t	111 0100	74
E	100 0101	45	u	111 0101	75
F	100 0110	46	v	111 0110	76
G	100 0111	47	w	111 0111	77
H	100 1000	48	x	111 1000	78
I	100 1001	49	y	111 1001	79
J	100 1010	4A	z	111 1010	7A
K	100 1011	4B	{	111 1011	7B
L	100 1100	4C	|	111 1100	7C
M	100 1101	4D	}	111 1101	7D
N	100 1110	4E	~	111 1110	7E
O	100 1111	4F			

*American Standards Institute Publication X3.4-1968

*ASCII CONTROL CODES

CONTROL	BINARY CODE	HEXADECIMAL CODE
NUL – Null	000 0000	00
SOH – Start of heading	000 0001	01
STX – Start of text	000 0010	02
ETX – End of text	000 0011	03
EOT – End of transmission	000 0100	04
ENQ – Enquiry	000 0101	05
ACK – Acknowledge	000 0110	06
BEL – Bell	000 0111	07
BS – Backspace	000 1000	08
HT – Horizontal tabulation	000 1001	09
LF – Line feed	000 1010	0A
VT – Vertical tab	000 1011	0B
FF – Form feed	000 1100	0C
CR – Carriage return	000 1101	0D
SO – Shift out	000 1110	0E
SI – Shift in	000 1111	0F
DLE – Data link escape	001 0000	10
DC1 – Device control 1	001 0001	11
DC2 – Device control 2	001 0010	12
DC3 – Device control 3	001 0011	13
DC4 – Device control 4 (stop)	001 0100	14
NAK – Negative acknowledge	001 0101	15
SYN – Synchronous idle	001 0110	16
ETB – End of transmission block	001 0111	17
CAN – Cancel	001 1000	18
EM – End of medium	001 1001	19
SUB – Substitute	001 1010	1A
ESC – Escape	001 1011	1B
FS – File separator	001 1100	1C
GS – Group separator	001 1101	1D
RS – Record separator	001 1110	1E
US – Unit separator	001 1111	1F
DEL – Delete, rubout	111 1111	7F

*American Standards Institute Publication X3.4-1968

In the seven-bit ASCII code, the most-significant or eighth bit is often used as a parity or check bit to determine the validity of a character's transmission. The value of this bit is set according to whether even or odd parity is desired. Even parity means that the number of ONE bits, including the parity bit, is even. Odd parity means that the number of ONE bits, including the parity bit, is odd. The following example uses the character "U" to illustrate parity.

Odd parity:
$11010101_2 = D5_{16}$ Number of ONE bits = 5, an odd number
Even parity:
$01010101_2 = 55_{16}$ Number of ONE bits = 4, an even number.

2.3 ARITHMETIC LOGIC UNIT

As mentioned in the previous chapter, the arithmetic logic unit (ALU) is a complex network of digital circuitry designed to execute arithmetic and logic operations as specified in the microprocessor's instruction set. A complete study of the ALU is not within the scope of this chapter; however, a cross section of some of the more typical instruction operations will be presented.

Description

The ALU's relationship to other computer system components is shown in Figure 2-3. Briefly, the control unit fetches instructions sequentially from memory, decodes, and executes them. Depending upon the instruction, data may be input from registers, memory, or an external device and transferred to the ALU for arithmetic or logical processing. Subsequently, the control unit outputs the resulting new data to memory or an external device.

Adders

In its simplest form, the arithmetic portion of the ALU can be thought of as a bank of binary adder circuits. The binary digital adders are usually called half-adders or full-adders. A half-adder performs addition upon two binary digits without taking into account a possible carry from a preceeding stage. Referring to Figure 2-4, the A and B inputs represent the two individual bits to be added by a half-adder unit. The S output is the least-significant bit of their sum. The C_{OUT} (carry) output is set to one if the result of the addition exceeds one. Table 2-8 shows the operation of a half-adder for the four possible input combinations. It should be noted here that the sum output by a half-adder produces a logical function known as Exclusive-OR (EX-OR). The symbolic logic equation for EX-OR is $A \veebar B = S$.

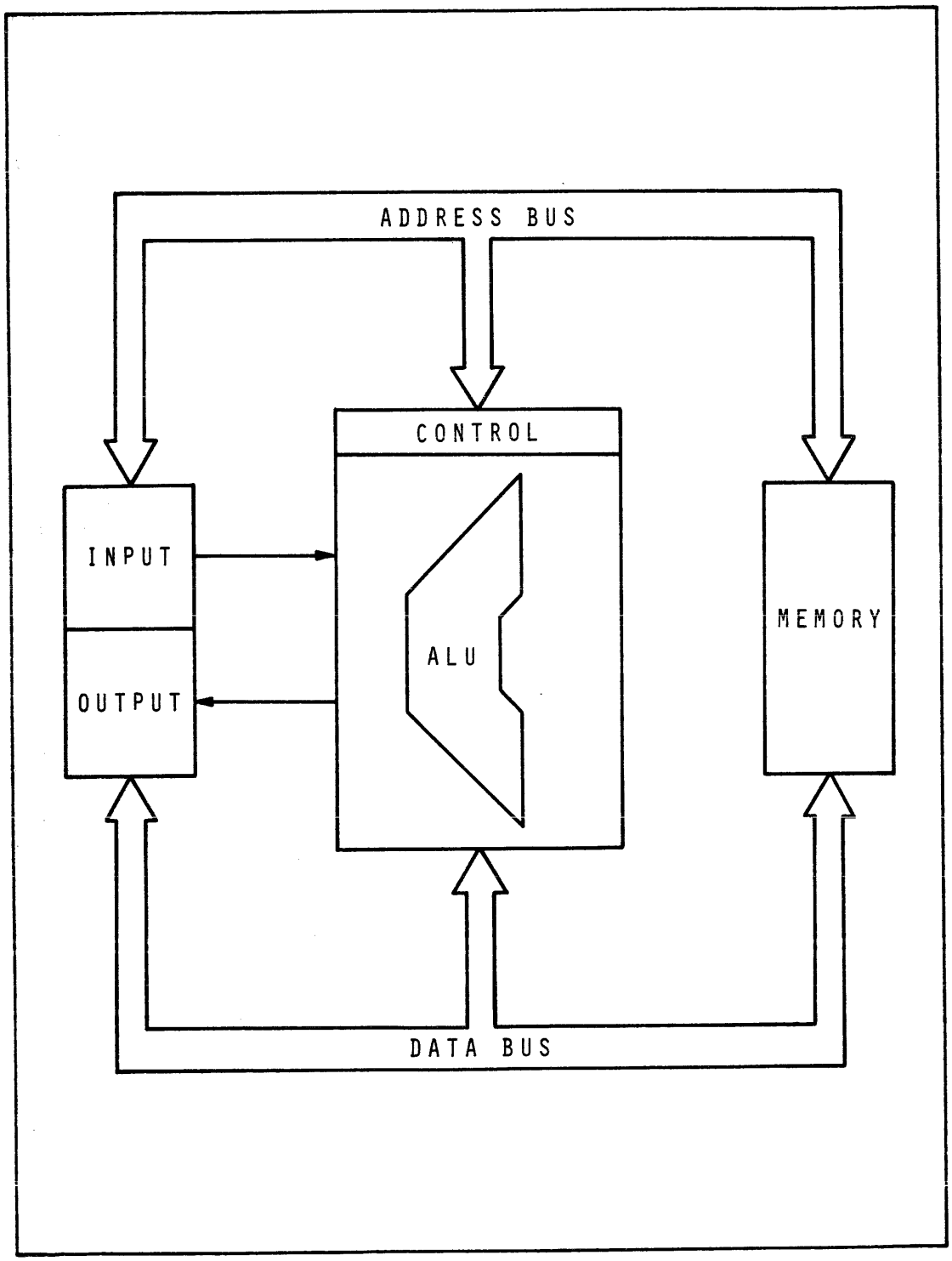


Figure 2-3. Simplified Microprocessor System .

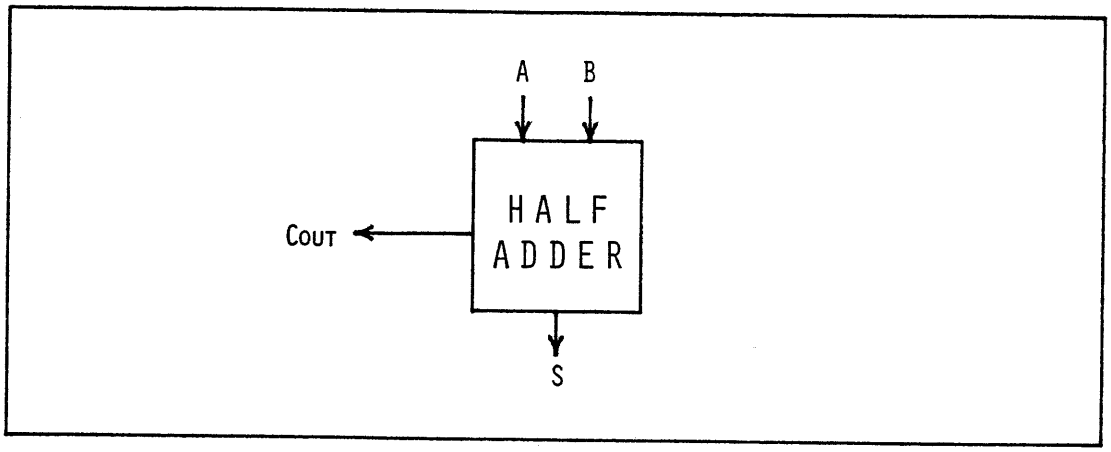


Figure 2-4. Half Adder

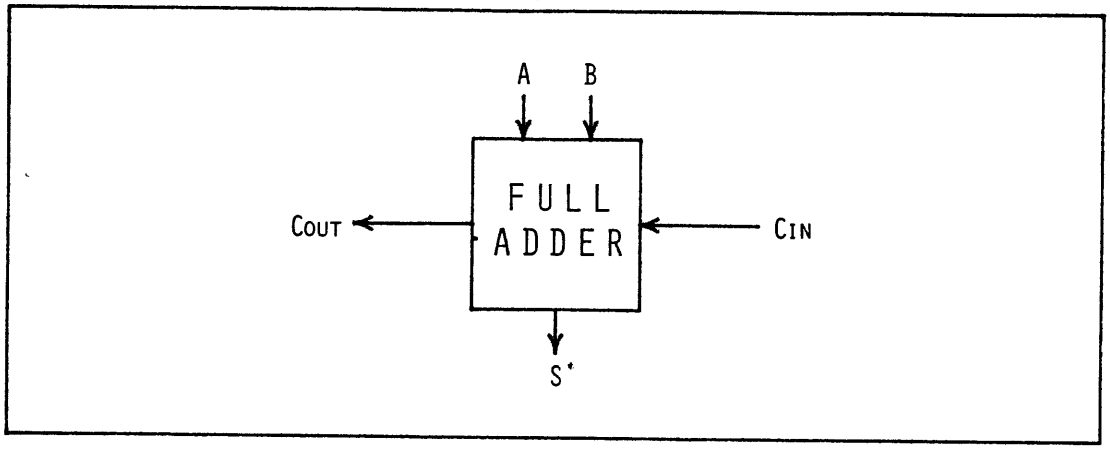


Figure 2-5. Full Adder

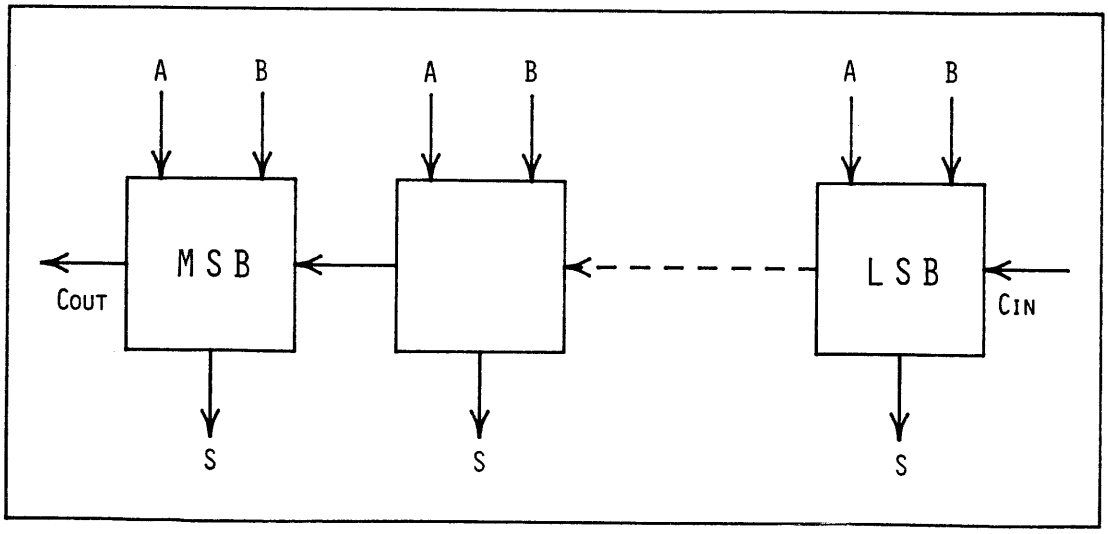


Figure 2-6. Parallel Adder Bank

The full-adder unit is needed to perform the arithmetic functions inside the ALU. Referring to Figure 2-5, it can be seen that the full-adder is much like the half-adder since both produce two outputs, sum and carry. Notice, however, that in addition to the A and B inputs, the full-adder has another input called the carry-in (C_{in}). This input enables the adder to take into account the carry output by a preceeding stage. Table 2-9 shows the operation of the full-adder for all possible input combinations.

Table 2-8. Truth Table for Half Adder

A	+	B	=	S	C
Ø		Ø		Ø	Ø
Ø		1		1	Ø
1		Ø		1	Ø
1		1		Ø	1

The adder examples presented thus far have shown additions only upon a single bit of binary data. In most microprocessors, the adders are parallel and number at least as many bits as are present in the system's word size. This collective aspect of the ALU is illustrated in Figure 2-6. The Texas Instruments' TMS 9980A is a parallel multi-bit microprocessor, capable of both 16- and 8-bit operations. A simplified version of the TMS 9980A's ALU is illustrated in Figure 2-7. The ALU's inputs (A and B) along with the output (RESULT) each represent a word of data. The carry-out (C_{out}) is output from the most-significant bit's adder. The carry-in (C_{in}) which is input at the least-significant bit's adder may be derived from the previous ALU operation's carry output. Selection of the data word size (either 16 bits or 8 bits) and the particular operation to be performed by the ALU upon the data at the two inputs is determined by the controller circuit via the control (CTL) input lines. The overflow (OV) output line and the complement (COMP) input lines are discussed in the following section.

ALU Operation

For a more thorough understanding of the various ALU functions several examples are shown. These examples are presented in the form of a problem and a solution in both binary and hexadecimal.

Half-Adder. The next example illustrates the function performed by a parallel bank of half-adders. Note that the carry from each bit's addition is ignored, thus creating an EX-OR result from the two binary values.

Table 2-9. Truth Table

A	+	B	+	C_{in}	=	S	C_{out}
0		0		0		0	0
0		0		1		1	0
0		1		0		1	0
0		1		1		0	1
1		0		0		1	0
1		0		1		0	1
1		1		0		0	1
1		1		1		1	1

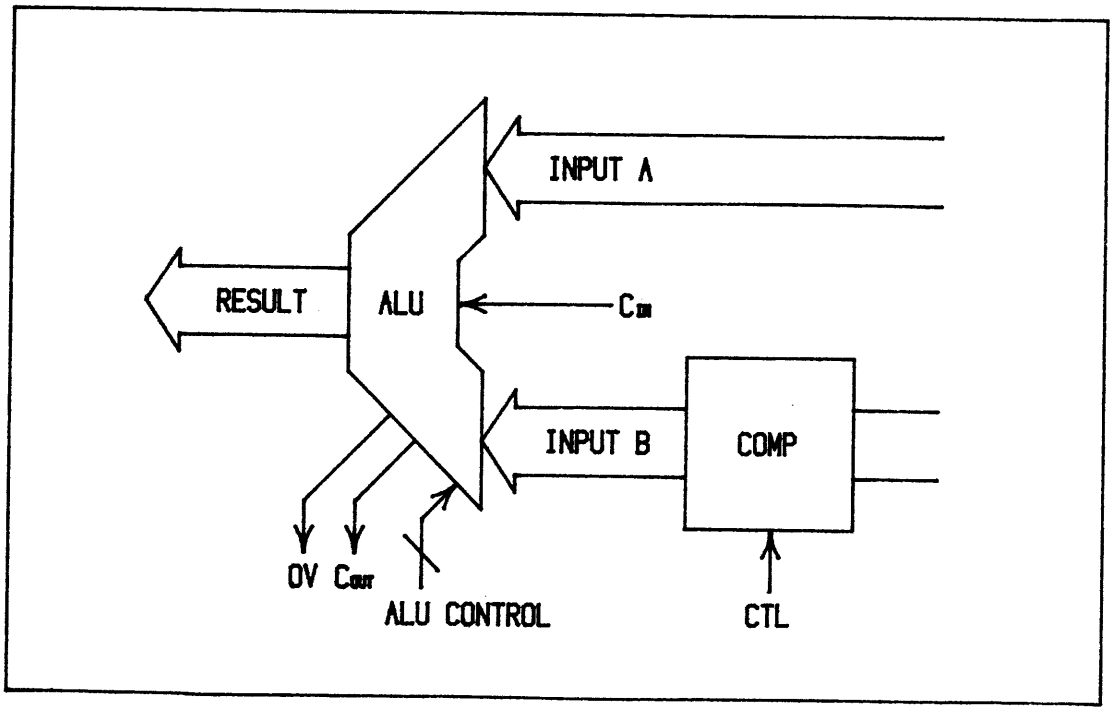


Figure 2-7. Simplified Version of the TMS 9980A ALU

EXAMPLE: Half-add (EX-OR)

PROBLEM: 0011011111101111_2 EX-ORed with 1101010101000001_2.

SOLUTION: A = $(0011)_2$ $(0111)_2$ $(1110)_2$ $(1111)_2$ = $37EF_{16}$,
B = $(1101)_2$ $(0101)_2$ $(0100)_2$ $(0001)_2$ = $D541_{16}$.

	msb															lsb
BIT	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
A	0	0	1	1	0	1	1	1	1	1	1	0	1	1	1	1
B	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0	1
S	1	1	1	0	0	0	1	0	1	0	1	0	1	1	1	0

ANSWER: S = $(1110)_2$ $(0010)_2$ $(1010)_2$ $(1110)_2$ = $E2AE_{16}$.

EXERCISE: 1010101111001101_2 EX-ORed with 0011011101000101_2.

ANSWER: $>9C88_{16}$

Full Adder. The following example utilizes the carry bit from each stage of addition to the next. A full addition is performed using the two binary values.

EXAMPLE: Full-add (ADDITION).

PROBLEM: 1111101011001110_2 + 0001101011011011_2.

SOLUTION: A = $(1111)_2$ $(1010)_2$ $(1100)_2$ $(1110)_2$ = $FACE_{16}$.
B = $(0001)_2$ $(1010)_2$ $(1101)_2$ $(1011)_2$ = $1ADB_{16}$.

	msb															lsb
BIT	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Cin	1	1	1	1	0	1	0	1	1	0	1	1	1	1	0	
A	1	1	1	1	1	0	1	0	1	1	0	0	1	1	1	0
B	0	0	0	1	1	0	1	0	1	1	0	1	1	0	1	1
S	0	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1
Cout	1	1	1	1	1	0	1	0	1	1	0	1	1	1	1	0

ANSWER: S = $(0001)_2$ $(0101)_2$ $(1010)_2$ $(1001)_2$ = $15A9_{16}$.

The example above was solved by singularly adding each bit of the two 16-bit words. Since this method of addition is time consuming, an alternate and faster method uses hexadecimal arithmetic as shown below.

PROBLEM: $FACE_{16} + 1ADB_{16}$.

SOLUTION:

$$\begin{array}{rl} & *C\text{<---}1 \quad 1 \quad 1 \\ (FACE)_{16} = & (15 \quad 10 \quad 12 \quad 14)_{10} \\ (1ADB)_{16} = & (1 \quad 10 \quad 13 \quad 11)_{10} \\ \hline & 1 \quad (17 \quad 21 \quad 26 \quad 25)_{10} \\ & (-16 \quad -16 \quad -16 \quad -16)_{10} \\ \hline & 1 \quad (1 \quad 5 \quad 10 \quad 9)_{10} = 15A9_{16}. \end{array}$$

Solve the following problem using binary and hexadecimal methods of addition.

EXERCISE: $0101110000110110_2 + 0010110110001011_2$.

ANSWER: 1000100111000001_2 or $89C1_{16}$.

Complementing. Referring to the ALU diagram in Figure 2-7, note the COMP block through which input word B must pass. The controller (CTL) can cause input word B to be preconditioned prior to its addition with input word A. This preconditioning forces each bit in word B to be inverted. The process of inverting each bit, (ZERO to ONE, ONE to ZERO) is called complementing. The result is called the ONE's complement of the original number. The process of forming the ONE's complement of a number is shown in the following example.

EXAMPLE: B--->B (1's COMP).

PROBLEM: ONE's complement 1011010101111111_2.

SOLUTION: $B = (1011)_2\ (0101)_2\ (0111)_2\ (1111)_2 = B57F_{16}$.

BIT	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
B	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	1
B(1 COMP)	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0

ANSWER: $B(1's\ COMP) = (0100)_2\ (1010)_2\ (1000)_2\ (0000)_2 = 4A80_{16}$.

EXERCISE: ONE's complement 0111111001101000_2.

ANSWER: 1000000110010111_2 or 8197_{16}.

*C <---1 means "a carry of one." When restricted to a 16-bit answer, the carry (17th bit) is lost.

A shorter method for determining ONE's complement is to use 15 complement. This procedure is illustrated in the following example using the same values from the previous example.

EXAMPLE: B--->B (15's COMP).

PROBLEM: 15 complement $B57F_{16}$.

SOLUTION:

$$\begin{array}{rl} (FFFF)_{16} = & (\ 15 \quad 15 \quad 15 \quad 15)_{10} \\ (B57F)_{16} = & \underline{(-11 \quad -5 \quad -7 \quad -15)_{10}} \\ & (\ 4 \quad 10 \quad 8 \quad 0)_{10} = 4A80_{16}. \end{array}$$

EXERCISE: 15 complement 8197_{16}.

ANSWER: $7E68_{16}$.

ONE's Complement. This action by the ALU is also known as the logical function called NOT. To execute an instruction which inverts or complements a word, the controller might present zero at input A and issue a command which complements the word at input B. An addition can then be performed as usual causing the complemented B value to be placed into the ALU's output word (RESULT).

TWO's Complement. This action by the ALU is performed by full-adder units which add one to the least-significant bit of the resulting ONE's complement of a number. The next example illustrates the TWO's complement process.

EXAMPLE: B--->B (2's COMP).

PROBLEM: TWO's complement 0110001110100000_2.

SOLUTION: B = $(0110)_2$ $(0011)_2$ $(1010)_2$ $(0000)_2$ = $63A0_{16}$.

BIT	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
B	0	1	1	0	0	0	1	1	1	0	1	0	0	0	0	0
B(1's COMP)	1	0	0	1	1	1	0	0	0	1	0	1	1	1	1	1
Add 1																1
B (2's COMP)	1	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0

B (2's COMP) = $(1001)_2$ $(1100)_2$ $(0110)_2$ $(0000)_2$ = $9C60_{16}$.

EXERCISE: TWO's complement 1000010100011001_2 or 8519_{16}.

ANSWER: 0111 1010 1110 0111_2 or $7AE7_{16}$.

A shorter method for determining TWO's complement using hexadecimal numbers can be accomplished by using the following rules.

1) Copy down any least-significant zeros of hexadecimal number,
2) Subtract the least-significant nonzero digit from 16,
3) Subtract all higher significant hex digits from 15.

Using these rules, the 16's complement is accomplished in the following manner.

PROBLEM: 16 complement $63A0_{16}$.

SOLUTION:

$$\begin{array}{rrrr} (15 & 15 & 16 & 0)_{10} \\ \underline{(-6} & \underline{-3} & \underline{-10} & \underline{0)_{10}} \\ (\ 9 & 12 & 6 & 0)_{10} \end{array} = 9C60_{16}.$$

EXERCISE: Sixteen's complement $2DB_{16}$.

ANSWER: $D25_{16}$.

Base Complementing. The term applied to the previously presented examples is called "base complementing." This process is important because it enables the microprocessor to subtract two values using addition. The principle can be seen from the following decimal subtraction problem. Suppose 43 is subtracted from 75. The result, of course, is 32. The same problem can also be solved by taking the TEN's complement of 43, adding it to 75 and considering only the two least-significant digits of the result:

$$\begin{array}{rrr} & (9 & 10)_{10} \\ & \underline{(4} & \underline{3)_{10}} \\ 43\text{'s (10 COMP)} = & 5 & 7_{10} \\ \text{adding,} & 7 & 5_{10} \\ \underline{\not{1}} & 3 & 2_{10} \end{array} = 32_{10}.$$

Although subtraction was used to obtain the TEN's complement, recall that the microprocessor simply complements the bits of the word, increments by one and then adds to obtain the TWO's complement.

$$\begin{array}{rrr} \text{Word B} = 43_{10} = 2B_{16} = & & 0010\ 1011_2 \\ & (1\text{'s COMP}) = & 1101\ 0100_2 \\ & & \underline{\qquad\qquad +1} \\ & (2\text{'s COMP}) = & 1101\ 0101_2 \\ \text{Word A} = 75_{10} = 4B_{16} & = & \underline{0100\ 1011_2} \\ \text{RESULT } \underline{\not{1}} & & 0010\ 0000_2 \end{array}$$

$(0010)_2\ (0000)_2 = 20_{16} = 32_{10}$.

*$\not{1}$ means "ignore carry."

Signed and Unsigned Numbers

For a microprocessor to fulfill the requirements of some applications, signed numbers must be employed. Positive numbers have a zero as their most-significant bit. By forming the TWO's complement of a number, a number of equal absolute value, but having an opposite sign, is produced. With negative numbers, the most-significant bit is set to ONE. Conversely, a positive number can be formed by TWO's complementing its negative equivalent.

For example, a byte value of positive one is represented by 00000001_2. A negative one is the TWO's complement of positive one or 11111111_2. As with decimal values, adding a positive one to a negative one yields zero.

$$\begin{array}{l} 0000\ 0001_2 \\ \underline{1111\ 1111_2} \\ 0000\ 0000_2. \end{array}$$

carry 1!

Number Range. The range, R, of absolute numbers for a computer the word size of which is n bits can be expressed as

$$0 \leq R \leq 2^n - 1.$$

Therefore, a byte's absolute decimal-number range can be computed as zero for the minimum and 255 ($2^8 - 1$) for the maximum.

The range of signed numbers for a computer with a word size of n bits can be expressed as

$$-2^{n-1} \leq R \leq 2^{n-1} - 1.$$

Therefore, a byte's signed decimal-number range (n=8) can be computed as -128 (-2^{8-1}) for the minimum and 127 ($2^{8-1} - 1$) for the maximum.

All the signed numbers for which n=4 are given below.

Table 2-10. Signed Number With n=4

$$\begin{array}{l} 0111_2 = +7 \\ 0110_2 = +6 \\ 0101_2 = +5 \\ 0100_2 = +4 \\ 0011_2 = +3 \\ 0010_2 = +2 \\ 0001_2 = +1 \\ 0000_2 = \ \ 0 \\ 1111_2 = -1 \\ 1110_2 = -2 \\ 1101_2 = -3 \\ 1100_2 = -4 \\ 1011_2 = -5 \\ 1010_2 = -6 \\ 1001_2 = -7 \\ 1000_2 = -8 \end{array}$$

Overflow. Referring to the ALU diagram in Figure 2-7, overflow (OV) is a status-line output by the unit. Overflow is a condition which exists whenever two signed numbers are added and the result is not within the signed-number range defined by $-2^{n-1} \leq R \leq 2^{n-1} - 1$. There are several rules which enable one to determine when an overflow condition will occur. The rules concerning overflow are as follows.

1. It applies only to signed-number addition.
2. When the two operands have opposite signs, overflow is impossible.
3. When the two operands have the same sign, overflow is possible and occurs whenever the sign of the sum is opposite that of the operands.

NOTE: Rules 2 and 3 apply after any TWO's complement operation has been performed on the subtrahend prior to the addition.

As stated in Rule 2, overflow cannot occur whenever two numbers with different signs are added. In an effort to produce overflow, extreme values are used in the next two examples.

Maximum positive value $127_{10} = 0111\ 1111_2$
Maximum negative value $\underline{-1_{10}} = \underline{1111\ 1111_2}$
$+126_{10}$ $\quad 0111\ 1110_2$

Minimum nonzero positive value $1_{10} = 0000\ 0001_2$
Minimum negative value $\underline{-128_{10}} = \underline{1000\ 0000_2}$
$-127_{10} = 1000\ 0001_2$.

There was no overflow, since both results lie within the range -128 to 127.

As stated in Rule 3, overflow is possible whenever two numbers with like signs are added. A sum with a different sign (i.e., incorrect sum) is an indication of overflow. The next four examples represent some possible combinations.

$+\ 64_{10} = 0100\ 0000_2$
$\underline{+\ 64_{10}} = \underline{0100\ 0000_2}$
$+128_{10} = 1000\ 0000 = -128_{10}$ (overflow)

$+\ 64_{10} = 0100\ 0000_2$
$\underline{+\ 63_{10}} = \underline{0011\ 1111_2}$
$+127_{10} \quad 0111\ 1111_2 = +127_{10}$ (no overflow)

$-\ 64_{10} = 1100\ 0000_2$
$\underline{-\ 64_{10}} = \underline{1100\ 0000_2}$
$-128_{10} = 1000\ 0000_2 = -128_{10}$ (no overflow)

$-\ 64_{10} = 1100\ 0000_2$
$\underline{-\ 65_{10}} = \underline{1011\ 1111_2}$
$-129_{10} = 0111\ 1111_2 = +127_{10}$ (overflow).

Carry and Overflow. Referring to Figure 2-7, carry (C_{out}) and overflow (OV) are two status output lines from the ALU. Examples using a bit width of three will be used here to illustrate the relationship between carry and overflow in signed numbers.

Signed Binary Number Set	Decimal Equivalents
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

A number greater than 3 or less than -4 is not within the range of the three-bit signed-binary-number set. Should any two numbers in the set be added, it is possible for the result to be outside the range (overflow state) and therefore invalid. The next four addition cases demonstrate this principle.

Case I:
$$\begin{array}{r} 010 = +2 \\ \underline{001 = +1} \\ 011 = +3 \end{array}$$
No overflow results since there was not a carry out of the two most-significant bits.

Case II:
$$\begin{array}{r} 011 = +3 \\ \underline{011 = +1} \\ 100 \neq +4 \end{array}$$
Overflow results since a carry has occurred out of the bit that precedes the sign bit.

Case III:
$$\begin{array}{r} 101 = -3 \\ \underline{111 = -1} \\ C = 1 \,|\, 100 = -4 \end{array}$$
No overflow results since carries have occurred out of the sign and preceding bit.

Case IV:
$$\begin{array}{r} 101 = -3 \\ \underline{110 = -2} \\ C = 1 \,|\, 011 \neq -5 \end{array}$$
Overflow results since a carry out of the sign bit has occurred.

Logical AND and OR Functions

In the interest of simplicity, earlier discussions concerning the ALU's arithmetic operation treated half- and full-adders as the fundamental blocks. However, these blocks are comprised of even more elementary logic circuits called AND and OR gates. The AND circuit principle can best be illustrated using the simple circuit in Figure 2-8.

The AND circuit contains two switches in series. For current to flow from the battery and light the lamp, both switch A and switch B must be closed. Hence, the term AND circuit. A chart (truth table) of the possible switch combinations and associated lamp status can be constructed as follows.

A \ B	OFF	ON
OFF	OFF	OFF
ON	OFF	ON

LETTING 0 = OFF
1 = ON

A \ B	0	1
0	0	0
1	0	1

The electronic symbol for the AND circuit is shown in Figure 2-9.

The A and B inputs to the AND circuit can be thought of as the series switches previously discussed. Using the truth table, two bytes of binary data can be ANDed as follows.

$$\begin{aligned} A &= 10011101_2 \\ B &= 11010100_2 \\ R &= \overline{10010100_2}. \end{aligned}$$

The OR circuit principle can best be illustrated using the simple circuit in Figure 2-10.

The OR circuit contains two switches in parallel. For current to flow from the battery and light the lamp, either switch A or switch B must be closed. Hence, the term OR circuit.

A truth table of the possible switch combinations and associated lamp status can be constructed as follows.

A \ B	OFF	ON
OFF	OFF	ON
ON	ON	ON

LETTING 0 = OFF
1 = ON

A \ B	0	1
0	0	1
1	1	1

The electronic symbol for the OR circuit is shown in Figure 2-11.

The A and B inputs to the OR circuit can be thought of as the parallel switches previously discussed. Using the truth table, two bytes of binary data can be ORed as follows.

$$\begin{aligned} A &= 10011101_2 \\ B &= 11010100_2 \\ \overline{R} &= \overline{11011101_2}. \end{aligned}$$

The EX-OR logic has been previously discussed. Its truth table is reviewed below.

A \ B	OFF	ON
OFF	OFF	ON
ON	ON	OFF

LETTING 0 = OFF
1 = ON

A \ B	0	1
0	0	1
1	1	0

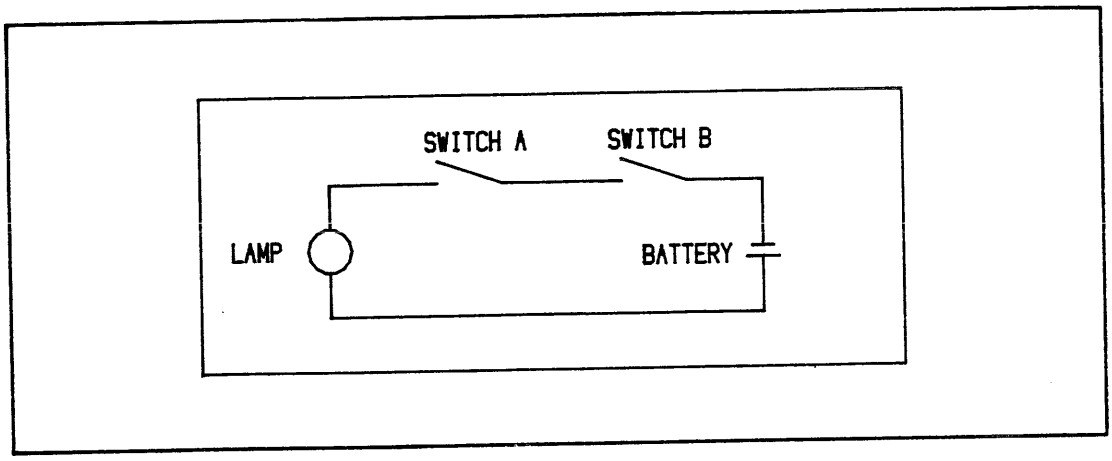


Figure 2-8. The "AND" Circuit Principle

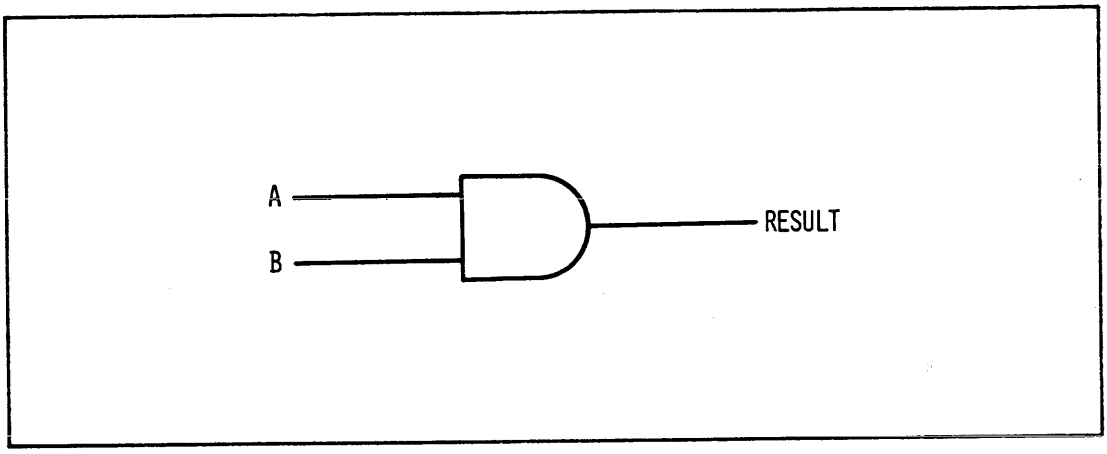


Figure 2-9. The Symbol for an "AND" Circuit

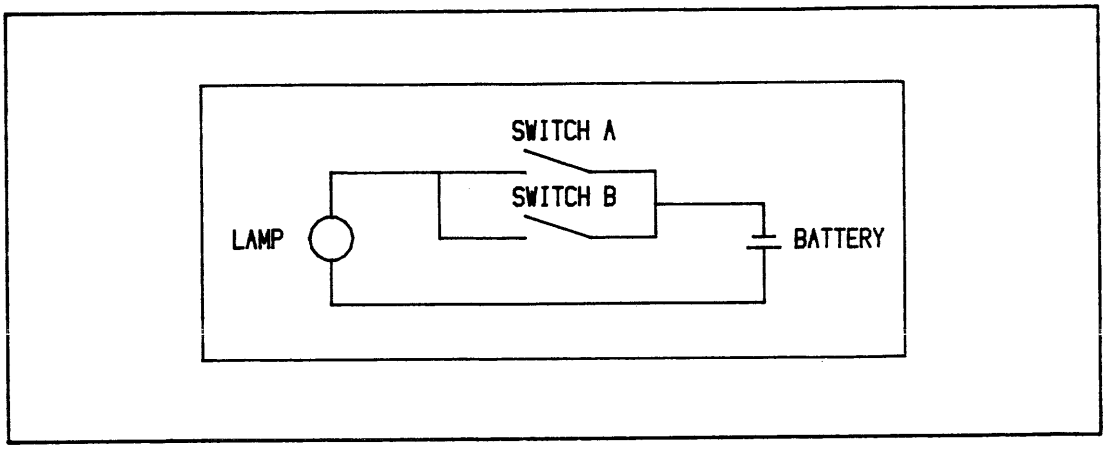


Figure 2-10. The "OR" Circuit Principle

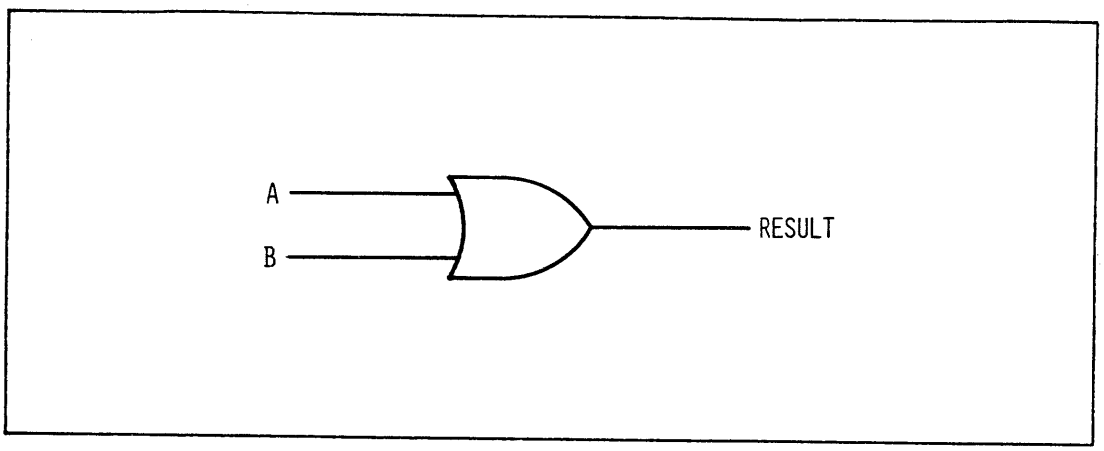


Figure 2-11. The Symbol for the "OR" Circuit

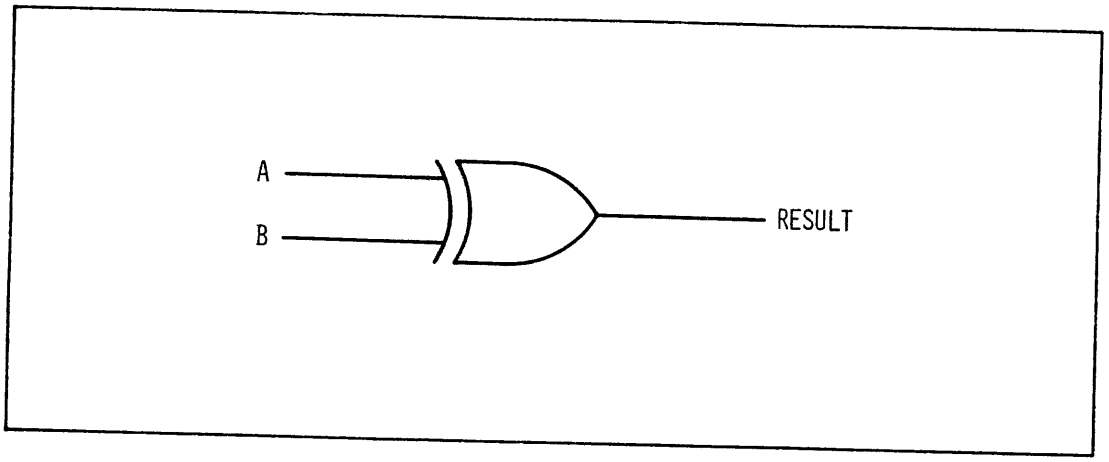


Figure 2-12. The Symbol for the "EX-OR" Circuit

The electronic circuit symbol for the EX-OR circuit is shown in Figure 2-12.

Two binary numbers may be EX-ORed in the following manner.

$$\begin{array}{l} A = 10011101_2 \\ \underline{B = 11010100_2} \\ R = 01001001_2. \end{array}$$

The adders contained in the ALU can now be broken down into the equivalent AND and EX-OR logic circuits as depicted in Figure 2-13.

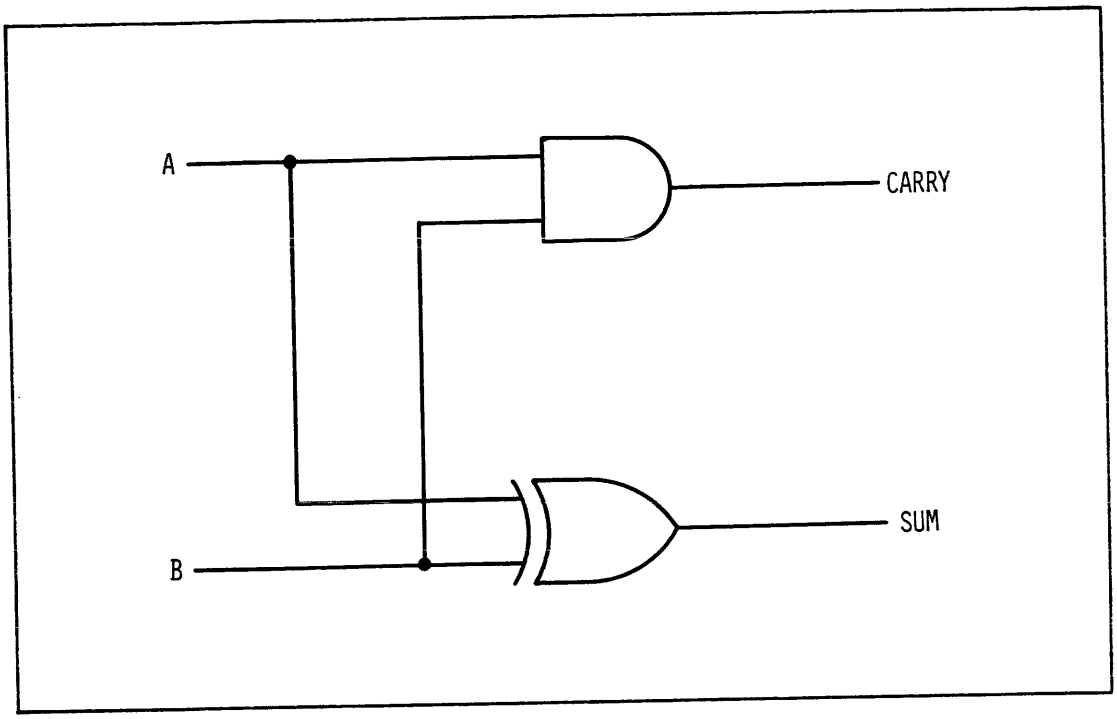


Figure 2-13. The "Adder" Circuit

The truth (addition) table can be written as follows.

A	0	0	1	1
B	0	1	0	1
SUM	0	1	1	0
CARRY	0	0	0	1

2.4 ON-BOARD TERMINAL

The TMS 9980A microprocessor, as a single integrated-circuit package, cannot function as a computer without additional related circuitry. This circuitry usually includes a clock generator for system timing, volatile memory storage called RAM (random access memory), communication ports, nonvolatile memory storage called ROM (read only memory), and a means for interacting with a user. The University Board contains all of the necessary components to form a microcomputer capable of being programmed.

The ROM on the University Board contains a program called UNIBUG. The name is taken from the first three letters in University and

the last three letters of debug. Debug is the term used to describe program trouble-shooting. The UNIBUG program enables the user to converse with the system via a set of key switches and display elements collectively known as the "on-board" terminal or, optionally, with an "off-board" terminal connected to the board.

The input portion of the calculator-like on-board terminal consists of a five-by-nine keyboard matrix pad. The keyboard pad contains a shift key, which causes the UNIBUG program to interpret each key in the matrix as one of two possible characters. Even though there are only 45 keys, this shifting procedure enables practically the full ASCII character and control set to be represented. Notable exceptions are the lower case letters and the ampersand (&) character.

Figure 2-14 shows the keypad's unshifted key code designations. Note the shift key located in the lower lefthand corner of the matrix pad.

The keys in the unshifted mode can be determined readily from their inscriptions. The space and carriage-return keys are marked Sp and Ret respectively.

Temporary depression of the shift key causes the keyboard matrix to assume the designations shown in Figure 2-15.

Many of the keys in the shifted mode are used for control, while the other keys are commonly used for special symbols. The control keys are used principally in data transmission protocol. The control key's abbreviations and descriptions are listed below.

ETB - End of transmission block
CAN - Cancel
EM - End of medium
DC1 - Device control 1
DC2 - Device control 2
DC3 - Device control 3
DC4 - Device control 4
NAK - Negative acknowledge
FF - Form feed
DEL - Delete
SO - Shift Out
SI - Shift In
DLE - Data-link escape
BEL - Bell (audible)
BS - Backspace
HT - Horizontal tab
LF - Line feed
VT - Vertical tab
STX - Start of text
ETX - End of text
EOT - End of transmission
ENQ - Enquiry
ACK - Acknowledge
ESC - Escape
SOH - Start of heading.

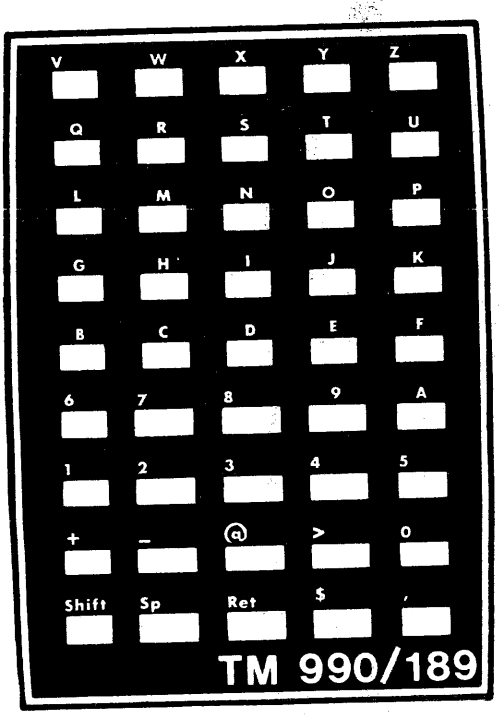


Figure 2-14. The Terminal Keypad (Unshifted)

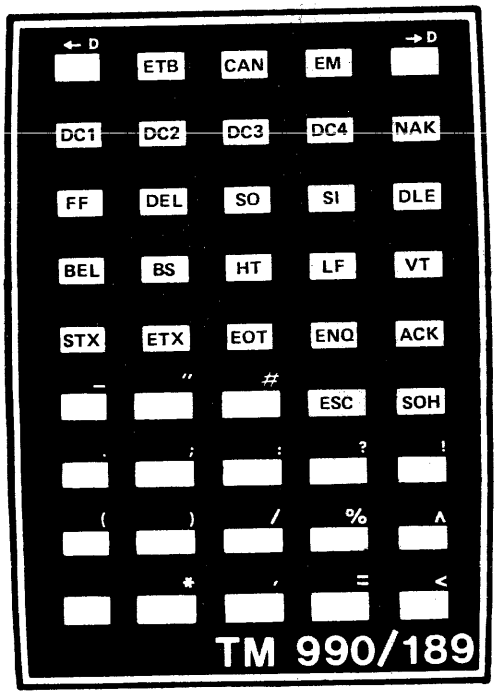


Figure 2-15. The Terminal Keypad (with Shift Key Depressed)

These ASCII control codes are not printable; therefore, their presence is represented by a blank in the display when they are keyed while in the UNIBUG monitor.

The output portion of the terminal consists of ten individual seven-segment displays as shown in Figure 2-16. These lamps will display the appropriate ASCII symbol each time a printable character's key is depressed. The DISPLAY LEFT (<---D), DISPLAY RIGHT (D--->) and the CURSOR are not ASCII keys, but represent special controls for manipulating the display characters.

2.5 UNIBUG MONITOR COMMANDS

Once the University Board has been initialized, the nonvolatile ROM on the board is programmed to respond to keyboard-command characters and perform certain functions. The monitor can perform the functions listed below.

Input Key	Monitor Function
A	Assembler execute with new symbol table
B	Assembler execute with current symbol table
C	CRU inspect/change
D	Dump memory to tape
E	Execute (to breakpoint)
F	Status register inspect/change
J	Jump to start of expansion EPROM
L	Load program from tape
M	Memory inspect/change
P	Program-counter inspect/change
R	Workspace register inspect/change
S	Single-step program execution
T	Typewriter function
W	Workspace-pointer inspect/change.

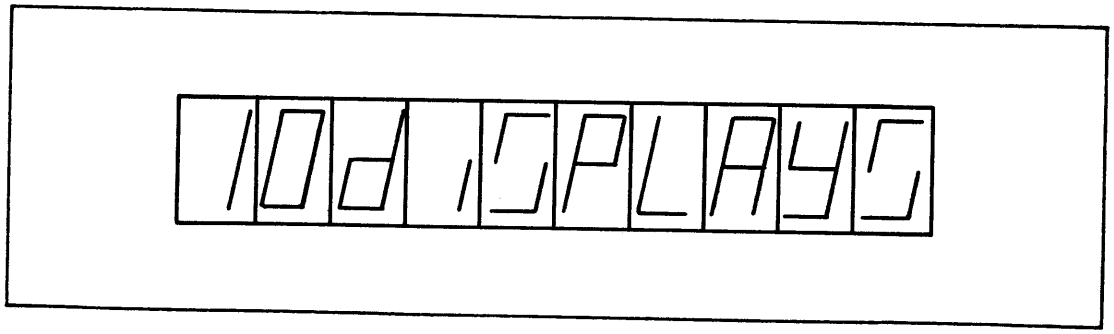


Figure 2-16. The Onboard Terminal Display

General Operation

Even though there are a multitude of programming functions to be mastered by the TM 990/189 user, one of the very first is the manual operation of the unit itself. Like learning to drive an automobile, the fastest method for becoming familiar with the University Board computer is to begin "hands-on" operation as soon as possible. Most of this section will use the "cookbook" approach toward learning to use the system; however, the operational knowledge gained here will be used throughout the remainder of the book.

The TM 990/189, like other electronic devices, must have electrical power to operate. Once power is applied, the circuit board is activated. This action causes an initialization routine within UNIBUG to begin execution. Part of this routine's responsibility is to perform a brief self-check operation on the major electronic components of the system. A part of the self-diagnostic is a series of audible "beeps" from the speaker, a series of flashes from the four user-addressable LED's, and a "CPU READY" message appearing in the display. This same initialization sequence can be accomplished by depressing the LOAD switch.

The SHIFT lamp is always off after the initialization process, indicating that the keypad is in the unshifted state. The user can now press the SHIFT key and observe the lighted SHIFT lamp and accompanying "click" from the speaker. The keypad is now in the shifted state and the keys assume the second set of ASCII values shown in Figure 2-15. The keypad can be returned to the unshifted state by depressing another key after the SHIFT key (except for the DISPLAY LEFT and DISPLAY RIGHT keys, which leave the keyboard in the shift state).

Since the TM 990/189 can service either the on-board terminal or an external terminal, the user must inform the system which terminal type will be used for interaction.

Ensuring that the keyboard is in the unshifted state, press Ret (carriage return) for use of the on-board terminal. (Otherwise, the letter P may be keyed for use of an external terminal device connected to the communication port.) Only the on-board terminal's usage will be discussed here.

After the Ret key has been pressed, a question mark (?) appears in the leftmost display. This symbol prompts the user to enter a command character: A,B,C, etc. All of the various commands are discussed later, but only the T (typewriter) command is discussed at this point.

Each display element has only seven segments to represent an ASCII character. This causes some of the symbols not to be readily discernable. The T command enables the user to key in the various characters and observe their associated symbols in the display. (Table 2-11 lists the printable ASCII characters and their associated symbols.) With the T command, these characters can be shifted or unshifted and entered in any order.

Table 2-11. Display Character Font

ASCII CHARACTER	DISPLAY CHARACTER	SEGMENTS ILLUMINATED	ASCII CHARACTER	DISPLAY CHARACTER	SEGMENTS ILLUMINATED
A		a,b,c,e,f	6		a,c,d,e,f,g
B		c,d,e,f,g	7		a,b,c
C		a,d,e,f	8		a,b,c,d,e,f,g
D		b,c,d,e,g	9		a,b,c,d,f,g
E		a,d,e,f,g	0		a,b,c,d,e,f
F		a,e,f,g	SPACE		(none)
G		a,c,d,e,f	@		a,b,d,e,f,g
H		b,c,e,f,g	$		a,c,d,f,p
I		c	*		b,c,g,p
J		b,c,d,e	'		b
K		b,e,f,g	>		a,b
L		d,e,f	+		b,c,g
M		a,c,e,g	-		g
N		c,e,g	(		d,e,g
O		c,d,e,g	)		c,d,g
P		a,b,e,f,g	%		b,e,g,p
Q		a,b,c,f,g	/		b,e,g
R		e,g	=		d,g
S		a,c,d,f	^		a,b,f
T		d,e,f,g	<		a,f,g
U		a,c,d,e,f	,		d,p
V		b.d.f.g	.		p
W		a,c,d,e	;		c,d,p
X		a,d,g	:		c,p
Y		b,c,d,f,g	?		a,b,e,g,p
Z		a,b,d,e	!		b,c,p
1		b,c	_		d
2		a,b,d,e,g	"		b,f
3		a,b,c,d,g	#		b,c,d,g
4		b,c,f,g			
5		a,c,d,f,g			

Using the key labeled (8/#) as an example, the unshifted mode produces an "8" symbol and the shifted mode produces the "#" symbol. Verify this operation by temporarily depressing the SHIFT key followed by the (8/#) key. Observe that the SHIFT lamp goes off and a "#" symbol is in the display. Again depress the (8/#) key and note that an 8 appears in the display to the right of the "#" symbol. The user may key various others characters to become more familiar with their representations.

Although only ten characters (including cursor) can be displayed at a time, an internal RAM memory-storage buffer allows up to 64 characters to be saved. The display normally contains the rightmost ten characters currently stored in the buffer. These rightmost characters are left-justified in the display. The tenth display character, a blinking cursor, indicates the position in the display where the next printable character will be placed.

The normal mode of display, as described above, can be altered by using the DISPLAY LEFT (shift V) and and DISPLAY RIGHT (shift Z) keys. Each depression of the DISPLAY LEFT key causes the six buffer characters to the left of those being displayed to be rotated into view. Conversely, each depression of the DISPLAY RIGHT key causes the six buffer characters to the right of those being displayed to be rotated into view.

To illustrate the operation of these function keys, activate the LOAD switch, key in an Ret followed by the letter T. While in the typewriter mode, key-in the alphabetic characters A through Z. Notice that once the display has been filled, depressing another key causes the leftmost character to be rotated off the display and into the storage buffer, while the rightmost character displayed is the one just entered. After the entire alphabet has been entered, depress the DISPLAY LEFT and DISPLAY RIGHT keys while observing the rotation of the characters to and from the buffer. After sufficient experimentaton, the typewriter mode can be exited by again pressing the LOAD switch.

UNIBUG Command Syntax

Most of the 14 UNIBUG commands either require or allow optional parameter fields to be entered with a command. The following explains the various command syntax conventions.

Unibug Command Syntax

T1	Indicates a space (SP) must be entered
T2	Indicates that either a space or comma must be entered
T3	Indicates that either a space, comma or return (Ret) must be entered
[ ]	Indicates that the content of the brackets is an optional item supplied by the operator
(Ret)	Return.

UNIBUG Command Descriptions

The principal UNIBUG commands are described in the following paragraphs of this section. Each command's explanation consists of an appropriately titled paragraph, proper syntax format, a brief description and an example of its usage. Some of the commands listed here are described in detail in the following chapters. In those cases, reference will be made to those chapters. The commands can be entered when the question mark prompt is displayed. Note that UNIBUG will display a space prior to displaying the first operand.

A--Assembler (Clear Symbol Table)

Syntax:

A [address]T3

Description: This command calls the symbolic assembler. Use of this command causes the symbol table storage area to be cleared in preparation for a different program's assembly. A more complete description of the assembler and its operation is given in Chapter 4.

Example:

?A 2000(Ret)

B--Assembler (Save Symbol Table)

Syntax:

B [address]T3

Description: This command calls the symbolic assembler; however, the symbol table is not cleared in memory. By saving an existing symbol table, a new program can be assembled which references labels in previously assembled programs or assembly can continue on the existing program. A more detailed description of this command's function is found in Chapter 4.

Example:

?B >2020(Ret) NOTE: >indicates hexadecimal

C--CRU Inspect/Change

Syntax:

C [CRU R12 ADDRESS] T2 [count] T3

Description: This command causes up to 16 bits of the given communication register unit (CRU) ports to be displayed in hexadecimal representation. The selected bits are displayed right-

justified in a hexadecimal word. The first addressed CRU bit is represented by the least-significant bit of the hexadecimal word. The next addressed CRU bit is represented by the next least-significant bit of the hexadecimal word, and so forth for the specified count. The CRU R12 ADDRESS is a hexadecimal value representing the hardware CRU bit address multiplied by two. (For example, the R12 ADDRESS for the CRU bit at CRU hardware address 40_{16} is 80_{16}.) The CRU R12 ADDRESS is a hexadecimal value which contains the 11-bit hardware CRU bit address in bit positions 4 through 14 of the word. The CRU output bits that correspond to the displayed CRU input bits displayed may be altered by keying in the desired hexadecimal data, right-justified. A terminating character after this data causes the data to be output. A carriage return as the terminating character causes a return to the UNIBUG command scanner following the data output. A minus sign (-) or a space as the output data terminating character causes the selected CRU output bits to be displayed again. The default value for the CRU R12 ADDRESS is zero and the default value for the count is 16.

Example:

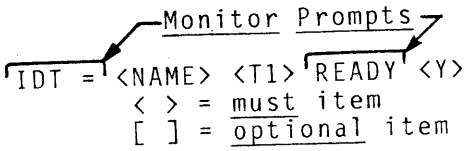
?C >20,4(Ret)	causes the state of the four user LED's on the University Board to be displayed as the rightmost hexadecimal digit. Changing this value, followed by a terminating character, would cause the LED's to reflect the value input.

NOTE: The CRU is further explained in subsection 6.5.

D--Dump Memory to Tape

Syntax:

D [start address] T2 [stop address] T2 [entry address] T2



Description: Memory is dumped to the cassette interface from the given "start address" to the "stop address." "Entry address" is the address in memory where program execution is to begin. IDT is the 1- to 8-character name to be given to the dumped data. Use of this command is described in Chapter 6.

E--Execute to Breakpoint

Syntax:

E [breakpoint address]T3

Description: The E command causes program execution to begin with the current values in the workspace pointer (WP), program counter

(PC), and status register (ST). An optional breakpoint (stop execution) address can be used to specify the location of an instruction where execution of the program stops and control returns to the monitor. This allows a program to be stopped at selected locations for debugging purposes.

Example:

?E (Ret)
?E 1000(Ret)

F--Flag (Status) Register Inspect/Change

Syntax:

F

Description: The user program's status register (ST) flags can be inspected and changed by this command. A full 16-bit value (4 hexadecimal digits) must be entered followed by an (Ret) to alter the register's contents. This requires the user to maintain all the flag-bit states except for those that are to be changed. The following list identifies the function of each flag bit in the register.

BIT 0 = Logical greater than (L>) (no sign bit)
BIT 1 = Arithmetic greater than (A>)(MSB is sign bit)
BIT 2 = Equal (EQ)
BIT 3 = Carry (C)
BIT 4 = Overflow (OV)
BIT 5 = Odd parity (OP)
BIT 6 = Extended operation (X)
BITS 7-11 = Reserved
BITS 12-15 = Interrupt mask.

Figure 1-9 shows the status register.

Example:
?F
?F = 2002 2000 (Ret) changes the contents of the user program's status register from 2002_{16} to 2000_{16}.

J--Jump to Start of Expansion EPROM

Syntax:

J [value 1] T2 [value 2] T2 [value 3] T3

Description: The J command causes a branch to the expansion EPROM on the University Board. The user must place a program in an EPROM device and place the device in the available socket on the TM 990/189. Up to three values can be passed to the program in the expansion EPROM; these will be stored starting at address 0080_{16}.

Example:

J 125A,3,72E(Ret)	causes the three values $125A_{16}$, 3, and $72E_{16}$ to be passed to addresses 0080_{16}, 0082_{16}, and 0084_{16}; and control given to a program in the expansion EPROM SOCKET.

L--Load Program From Tape

Syntax:

L

Description: Memory is loaded from the cassette interface using the addresses and data found on the tape. Refer to Chapter 6 for a detailed description of this command.

Example:

?L

M--Memory Inspect/Change

Syntax:

M [address]T3

Description: Memory inspect/change "opens" the selected hexadecimal memory location, displays the contents of that location and allows the option of changing the data in that location. There is also a provision for advancing to the next location or backing up to the previous address. The termination character used after a memory location is opened (and optional data is entered) causes different results. If the termination character is a

- Carriage return, control returns to the UNIBUG command scanner
- A space or comma, the next memory location is opened and displayed
- A minus sign, the previous memory location is open and displayed
- A hexadecimal value, it is entered prior to the termination character, and the displayed memory location is updated to the value entered.

The default value for the address operand is zero.

Example:

?M 200(Ret)

0200 = F0FF T2	F0FF is contents of location 0200_{16}
0202 = 0012 FFFF T2	causes $FFFF_{16}$ to replace 0012_{16} at location 0202_{16}.
0204 = A0B0 -	$A0B0_{16}$ is contents of location 0204_{16}; minus sign causes previous address to be displayed.
0202 = FFFF(Ret)	$FFFF_{16}$ is contents of location 0202_{16}; (Ret) exits memory command.

P--Program Counter Inspect/Change

Syntax:

P

Description: The user program's program counter (PC) can be inspected and, optionally, changed by this command. The value in this register is the address at which execution begins when the E or S command is used.

To alter the displayed value, enter a new hexadecimal value followed by a (Ret).

Example:

?P

?P = 20AE 220(Ret)	changes the contents of the user program's program counter from $20AE_{16}$ to 0220_{16}.

R--Register Inspect/Change

Syntax:

R[hexadecimal register number]T3

Description: The user-program's working-register contents can be inspected and, optionally, altered by this command. The contents of the selected register is displayed as four hexadecimal digits. The contents can be changed by entering a new hexadecimal value followed by a terminating character. The permissible terminating characters and the resulting action is the same as described with the M command. The selected register is referenced by a hexadecimal digit. The default hex register number is zero.

Examples:

?R9(Ret)	displays the contents of working register 9.
R9 = FFF7(Ret)	$FFF7_{16}$ is the contents of register 9.
?RB(Ret)	displays the contents of working register 11.
RB = 3362 7CD8	changes the contents of R11 from 3362_{16} to $7CD8_{16}$.

The selected register's number is displayed a long with the contents of the register. If subsequent registers are examined (by use of a space or minus character), the address of the subsequent register is displayed with the contents of the address.

For example, assume the workspace pointer is set to 100_{16}.

?R9(Ret)	operator specifies register 9.
R9 = 012A(Sp)	contents of register 9 (memory location 0112_{16} is $012A_{16}$. Operator enters space to examine next register.
0114 = 9D34(Ret)	114_{16} is address of register A_{16}. Contents of register A_{16} is $9D34_{16}$. Carriage return exits the command sequence.

Notice that the selected register should be specified as a hexadecimal digit. If more than one digit is entered, only the last digit is used to determine the selected register.

For example, if the operator enters R11 with the intention of examing register 11 (B_{16}), only the last digit will be used to select the register, therefore register 1 will be displayed.

S--Single Step

Syntax:
S

Description: The single-step command causes execution to begin at the address in the user's program counter. Execution is limited to one instruction at a time, which enables the use of the other UNIBUG command functions between instructions to debug the program. Successive instructions are executed by successive pressing of the S key. The display will show the letter S. To the right of the S is the address of the next instruction. To the left of the S is the least significant 8-bits (two hexadecimal digits) of the address of the instruction just executed. A carriage return exits the command.

Example:

?S

02S 0204 — 0204 is the address of the next instruction and 02 represents the leastsignificant 8-bits of the address of the instruction just executed.

T--Typewriter Program

Syntax:

T

Description: The typewriter command enables the user to evaluate the terminal characters and controls. The mode can be exited only by reactivating the LOAD switch.

Example:

?T ABCDEF123456789

W--Workspace Pointer Inspect/Change

Syntax:

W

Description: The user-program's workspace pointer (WP) can be inspected and changed by this command. The value in this register is the address at which the user's workspace registers start. This is the address of the workspace implemented whenever the E or S command is used.

Example:

?W

?W = 00CC 380(Ret) changes the contents of the user program's workspace pointer from $00CC_{16}$ to 0380_{16}.

2.6 SUMMARY

In this chapter one of the main objectives is to demonstrate fundamental facts common to different number systems. It is shown that positional notation is the avenue toward converting a value of any number base to an equivalent decimal value. Simple algorithms are used to convert decimal values into another base. The chapter deals mostly with the mechanical manipulation of the number systems in order to convey only the essential knowledge necessary for working with microprocessors.

Although the range of number bases is infinite, only the more useful ones such as decimal, binary, octal, and hexadecimal are used with computers. It is shown that binary numbers lend themselves to electrical switching principles and are, therefore, the basic system. Convenient grouping of the binary digits gave rise to the octal and hexadecimal systems as well as BCD and ASCII.

The EX-OR, ADD, INV, AND, and OR functions are shown to be the fundamental tasks performed by the Arithmetic Logic Unit with carry (C) and overflow (OV) being two important states output. Carry is characteristic of both signed and unsigned numbers while overflow pertains only to signed numbers.

Finally, usage of the UNIBUG commands is discussed to allow the user to perform and verify these logical functions with the TM 990/189. The facts learned in this chapter should prove useful in the subsequent chapters.

2.7 EXERCISES

Positional Notation

1. Write the following numbers in positional notation.
   (a) 1357_{10}
   (b) 409_{10}
   (c) 5_{10}

2. Write the following numbers in shorthand notation.
   (a) $[7 \times (10^2)] + [2 \times (10^1)] + [0 \times (10^0)]$
   (b) $[3 \times 10^1)] + [8 \times (10^0)]$
   (c) $[9 \times (10^1)] + [9 \times (10^0)]$

Binary Conversions

3. Convert the following binary numbers to decimal.
   (a) 1011_2 = _____$_{10}$
   (b) 11111111_2 = _____$_{10}$
   (c) 10000000_2 = _____$_{10}$

4. Convert the following decimal numbers to binary.
   (a) 128_{10} = _____$_2$
   (b) 15_{10} = _____$_2$
   (c) 255_{10} = _____$_2$

Octal Conversions

5. Convert the following decimal numbers to octal.
   (a) 8_{10} = _____$_8$
   (b) 29_{10} = _____$_8$
   (c) 312_{10} = _____$_8$

6. Convert the following octal numbers to decimal.
   (a) 10_8 = _____$_{10}$
   (b) 100_8 = _____$_{10}$
   (c) 100_8 = _____$_{10}$

7. Convert the following binary numbers to octal.
   (a) 010110010_2 = _____$_8$
   (b) 11001011_2 = _____$_8$
   (c) 101_2 = _____$_8$

8. Convert the following octal numbers to binary.
   (a) 177_8 = _____$_2$
   (b) 35_8 = _____$_2$
   (c) 127_8 = _____$_2$

Hexadecimal Conversions

9. Convert the following decimal numbers to hexadecimal.
   (a) 32_{10} = _____$_{16}$
   (b) 128_{10} = _____$_{16}$
   (c) 300_{10} = _____$_{16}$

10. Convert the following hexadecimal numbers to decimal.
   (a) D_{16} = _____$_{10}$
   (b) $CA7_{16}$ = _____$_{10}$
   (c) $FADE_{16}$ = _____$_{10}$

11. Convert the following binary number to hexadecimal.
   (a) 10000001_2 = _____$_{16}$
   (b) 10111111_2 = _____$_{16}$
   (c) 1011_2 = _____$_{16}$

12. Convert the following hexadecimal numbers to binary.
   (a) $1AF_{16}$ = _____$_2$
   (b) $5BE_{16}$ = _____$_2$
   (c) $9CD_{16}$ = _____$_2$

Fractional Conversions

13. Convert the following decimal fractions to the three different bases.
    (a) 3.1428_{10} = _____ 2
    (b) 10.125_{10} = _____ 8
    (c) 1.768_{10} = _____ 16

14. Convert the following three base fractions to decimal.
    (a) 1.1011_{2} = _____ 10
    (b) 7.345_{8} = _____ 10
    (c) $A.CDE_{16}$ = _____ 10

BCD Conversions

15. Color-in the lamps representing the following decimal numbers.

(a) 17_{10} =
0---8---0
0---4---0
0---2---0
0---1---0

(b) 80_{10} =
0---8---0
0---4---0
0---2---0
0---1---0

(c) 61_{10} =
0---8---0
0---4---0
0---2---0
0---1---0

16. Convert the following binary values to BCD equivalents.
    (a) 01010110_{2} = _____ BCD
    (b) 1000.0001_{2} = _____ BCD
    (c) 001110000111_{2} = _____ BCD

17. Convert the following BCD values to binary digits.
    (a) 73_{BCD} = _____ 2
    (b) 1.23_{BCD} = _____ 2
    (c) 14.0_{BCD} = _____ 2

ASCII Conversions

18. Using the hexadecimal ASCII table, evaluate the following characters. To the right, evaluate the same characters using odd parity.
    (a) A = _____ 16 _____ 16
    (b) Z = _____ 16 _____ 16
    (c) 0 = _____ 16 _____ 16
    (d) 9 = _____ 16 _____ 16

19. Using the hexadecimal ASCII table, evaluate the following hex codes. To the right, indicate an odd or even number of bits.

(a) 48_{16} = _____ ; _____
(b) 55_{16} = _____ ; _____
(c) 33_{16} = _____ ; _____
(d) $7F_{16}$ = _____ ; _____

20. Perform the following logical functions on the hexadecimal values 7DF3 and ABCD.

(a) Half-add (EX-OR)
(b) Full-add (ADD)
(c) Invert (each value)
(d) TWO's complement (each value)
(e) Subtract first from second
(f) 'AND'
(g) 'OR'

2.8 LAB EXPERIMENTS

The exercises presented in this section are used to reinforce the knowledge gained during the course of this chapter. First solve the problem manually and then verify the solution by using the University Board. Enter (via the on-board terminal) and execute short programs which leave the correct result in the specified registers.

Each of the given programs has the same workspace area and same starting address but requires breakpoints at various addresses. The following instructions pertain to the proper entry and execution procedure needed to obtain the correct problem solution for each program. This procedure is described only once but is used throughout the remainder of the lab experiments in this chapter. Refer to the UNIBUG command descriptions, if necessary, for further explanations.

(1) Enter a workspace pointer value of hexadecimal 300 using the W command.

(2) Enter a starting address value of hexadecimal value 200 (labeled START) using the P command.

(3) Clear the status register by entering into it the value zero using the F command.

(4) Enter the listed hexadecimal values representing the short program using the M command starting at memory address 200_{16}.

(5) Verify proper entry of the program by once again cycling through memory and ensuring that the correct instruction value is contained in the proper location.

(6) Finally, the single-step S commands can be used to sequentially execute the program's instructions until the address labeled STOP is reached.

(7) The user should then inspect the requested register's contents.

1. Exclusive - OR (half add) the two hexadecimal values 27EF and C541. First convert the values to binary numbers.

Manual: $27EF_{16}$ = _____ 2
$C541_{16}$ = _____ 2
Answer = _____ 2 = _____ 16

Program:

Address	Code			
0200	0200	START	LI	R0,>27EF
0202	27EF			
0204	0201		LI	R1,>C541
0206	C541			
0208	2801		XOR	R1,R0
020A	1000	STOP	NOP	

Inspect R0 = _____ 16

E2AE

2. Full-add the two hexadecimal values 7FFF and 7FFF. Also determine the resulting OVERFLOW and CARRY states after the addition. First convert the values to binary numbers.

Manual: $7FFF_{16}$ = _____ 2
$7FFF_{16}$ = _____ 2
Answer = _____ 2 = _____ 16

OV = _____ 2 CY _____ 2

Program:

Address	Code			
0200	0200	START	LI	R0,>7FFF
0202	7FFF			
0204	C040			MOV R0,R1
0206	A040		A	R0,R1
0208	1000	STOP	NOP	

Inspect R1 = _____ 16

Inspect Flag Reg. = _____ 16

Overflow = _____ 2

Carry = _____ 2

3. Fifteen's complement the decimal value 16. First, convert the value to hexadecimal, then complement.

Manual:

16_{10} = _____ 16 = _____ COM

Program:

Address	Code		Instruction	
0200	0200	START	LI	R0,16
0202	0010			
0204	0540		INV	R0
0206	1000	STOP	NOP	

Inspect R0 = _____ 16

4. Sixteen's complement decimal value 16. First convert the value to hexadecimal.

Manual:

16_{10} = _____ 16 = _____ COM

Program:

Address	Code		Instruction	
0200	0200	START	LI	R0,16
0202	0010			
0204	0540		INV	R0
0206	0580		INC	R0
0208	1000	STOP	NOP	

Inspect R0 = _____ 16

5. Subtract 16_{10} from 100_{10} using sixteen's complement. First convert the values to hexadecimal.

Manual: 100_{10} = _____ 16

-16_{10} = _____ 16

Answer = _____ 16

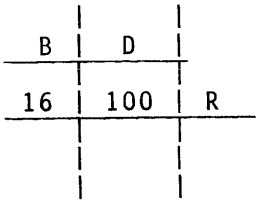
Program:

Address	Code		Instruction	
0200	0200	START	LI	R0,100
0202	0064			
0204	0220		AI	R0,-16
0206	FFF0			
0208	1000	STOP	NOP	

Inspect R0 = _____ 16

6. Using successive subtractions determine how many times 16_{10} can be removed from 100_{10}. Also give the remainder.

Manual:



Program:

Address	Code	Instruction		
0200	04C0	START	CLR	R0
0202	0201		LI	R1,100
0204	0064			
0206	0221	LOOP	AI	R1,-16
0208	FFF0			
020A	1702		JNC	FINISH
020C	0580		INC	R0
020E	10FB		JMP	LOOP
0210	0221	FINISH	A1	R1,16
0212	0010			
0214	1000	STOP	NOP	

Inspect R0 = _____ 16 COUNT

Inspect R1 = _____ 16 REMAINDER

CHAPTER 3

INTRODUCTION TO COMPUTER ADDRESSING AND PROGRAM DEVELOPMENT

3.1 INTRODUCTION

This chapter introduces the user of the University Board (TM 990/189) to three major computer programming topics. Computer addressing is presented in relation to computers in general, and to the TMS 9980A microprocessor in particular. Computer instructions are presented and explained, along with related exercises and application notes. Then, the proper procedure for producing a computer program is introduced. These three topics are correlated with exercises and lab experiments.

As a background to this introduction to computer programming, the three principal levels of programming language will be discussed briefly. These are

- Machine language
- Assembly language
- High-level language.

Machine Language

Machine Language (also known as machine code) is the most elementary (though not the simplest) of these levels. It is the "software" language of ONE's and ZERO's by which the computer can be programmed directly. It is important, indeed necessary, to have an understanding of this level of computer programming language, even if one intends to program mainly in assembly language or a high-level language. This understanding is necessary for a programmer to modify (patch) the machine code of a program. Program patching is generally done when small changes are needed. When a bug is discovered, patching is beneficial to check out a proposed solution. Such verification avoids investing time in documenting an incorrect solution. It is unwise for a programmer to attempt to patch a program if he lacks a clear understanding of the related machine-language formats.

Assembly Language

Although machine language is the most elementary level, assembly language is currently the most popular level for microprocessor applications and is the focus for this book. Assembly language

is "one step above" machine language. It could be referred to as the "humanized" version of machine language. Assembly language can be defined simply as mnemonic code, which uses word abbreviations having a one-to-one relationship to machine-language instructions. Assembly language is most popular for microprocessor applications for the following reasons.

- In contrast to machine language, it provides a balance between readability and capability to control computer functions.
- In contrast to high-level language, it generally executes faster, and it requires less memory space.

Although assembly language is the most popular level for microprocessor applications, it is important to consider the third principal level of computer programming language.

High-Level Language

The interest in high-level language is illustrated by the proliferation of languages such as FORTRAN, COBOL, Pascal, BASIC, and many others which are oriented toward users. In these languages each source statement becomes five or ten machine codes in contrast to the one-to-one relationship for assembly language. Further distinctions between assembly language and high-level language are given in Chapter 4.

3.2 COMPUTER ADDRESSING: WHAT DOES IT MEAN?

Computer addressing means a lot of different things. It is considered here first with respect to computers in general, then the discussion focuses upon the specifics of computer addressing with respect to the TMS 9980A microprocessor.

The word "addressing" or "address" refers to some location, such as the address of a friend's house. The friend who lives in the house and the address of the house are two different things, but the address does indicate the location of the house and, thus, the location of the friend.

With computers in general, an address, or addressing, similarly refers to a location. The items in these locations can be considered in three categories:

- Location of an operand
- Location of next instruction
- Location of a peripheral device.

Location of an Operand

The first and primary category of addressing is the location of an operand. An operand is an item to be operated upon, such as an addend in addition. With computers in general, the operand's location will be one of five different types:

- Peripheral input or output device
- Register
- Instruction (called an immediate operand)
- Stack
- Memory.

Location of Next Instruction

The computer address can also refer to the location of the next instruction. Typically, the next instruction is located immediately following the current instruction. Thus, the program counter (PC) is simply incremented during the execution of the current instruction in preparation for the next instruction fetch. However, a transfer of control instruction (often called jump, branch, or skip instructions) can be used to change the program flow, in which case the address of the PC is modified so that it points to an instruction other than the next one. Thus, the location of the next instruction can be determined by any one of the following.

- Program Counter--The program counter usually contains the location of the next instruction because it is incremented during the execution of the current instruction.
- Register--The location of the next instruction may be located in some register; therefore, the contents of this register would be moved into the program counter.
- Instruction--The location of the next instruction may be immediately a part of the current instruction; if so, this constant would be moved into the program counter.
- Stack--The location of the next instruction may be in a hardware or software stack, e.g., the return address of a subroutine.
- Memory--The location of the next instruction may be in some memory location.

Location of a Peripheral Device

As indicated, addressing refers to the location of an operand or it may refer to the location of the next instruction. A third category of computer addressing relates to the location of a peripheral device. These devices may fall into several subcategories as mentioned in Chapter 1. All three of these addressing categories are summarized in Table 3-1.

Table 3-1. Summary of Addressing Categories

Location of an Operand	Location of the Next Instruction	Location of a Peripheral Device
Peripheral input/output device	Program Counter	Input and/or output Serial and/or parallel
Register	Register	Type of device control
Instruction	Instruction	-Program-controlled
Stack	Stack	-Interrupt-driven
Memory	Memory	-DMA

Address and Contents Distinguished

It is important to clearly distinguish the address of the location from the contents of the location. Computer addressing is similar to street addressing. On each house there is a number, which is the address. Each house has a person (or persons) living in it, which corresponds to the contents of the house. It is essential in programming a computer to keep this distinction clear since both the contents and the address will be numbers. Even though they look similar, they are different! One number, which is called the address will refer to the location where something is. This location will contain the contents. So, the numerical address will point to the location which contains the numerical contents to which reference is made.

For example, memory location 0400_{16} (address) may contain the number 1234_{16} (contents), which is the number to be added to the contents of register 1. Note that the operand has an address which is different from the contents. In all likelihood, there is also a location 1234_{16} (address) which probably contains a different number.

From this point on in the book, any hexadecimal value may be indicated with the greater than symbol (>) to simplify notation. Hence,

0400_{16} = >0400
1234_{16} = >1234.

Program Example Introduced

In the first half of this chapter, the computer instructions necessary to implement a program example are presented. This program example calculates the function of N1 and N2 represented in the following equation:

$$F(N1,N2) = [4\ (N1 - N2)]^2$$

This function can be clarified by breaking it down as follows. First, subtract N2 from N1, then, multiply that result by 4 and, finally, square that result.

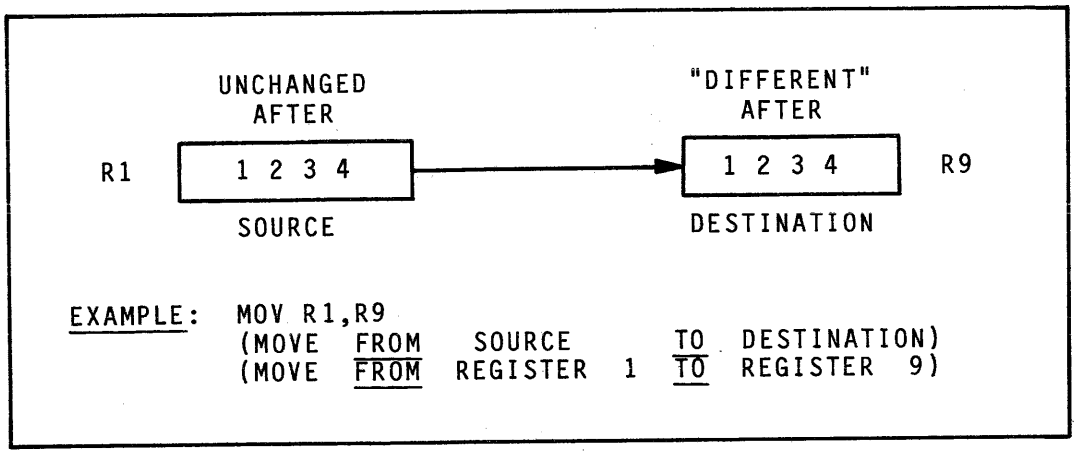


Figure 3-1. Register Direct Addressing Illustration

Copy Operation

To introduce the six principal addressing modes of the 990/9900 family (of which the TMS 9980A is a member), the copy operation will be considered first. This is one of the most elementary computer operations. It copies from the contents of one location (the source) to the contents of another (the destination). In this operation, the contents of the destination are changed to become identical with the contents of the source, but the source is left unchanged. This operation is similar to sending a page through a copy machine. In the TMS 9980A instruction set, this copy operation is called a "move." It is stressed that the source does not move in the normal sense but remains in the same place. To illustrate, suppose the operator desires to transfer or copy the contents of register 1 to register 9. This would be written in mnemonic code as: MOV R1,R9. The contents of register 1 are copied into the contents of register 9 with the contents of register 1 left unchanged (see Figure 3-1).

Register Direct Addressing

Register direct addressing is the first of the six principal addressing modes and is illustrated in Figure 3-1. With this mode, the contents of the register are used directly by the instruction operation. In mnemonic code, it is written as MOV R1,R9. Note that the address of the source operand is register 1. The contents of this register may be any possible 16-bit number. After execution of the instruction, the contents of register 9 (the destination) will be the same as that in register 1 (the source). Thus, when the contents of a register are referred to directly, it is called register direct addressing. Register direct addressing is indicated in mnemonic code simply by referring to the register number (with no special character preceding it). Correspondingly, in machine code code, there is a special field to indicate how the register is

to be used. This special field is explained further on. Recall from Chapter 1, in the TMS 9980A architecture, registers are in memory. Consequently, when the instruction refers to a register, it is referring to a memory location. Each of the 16 workspace registers, as well as having a specific memory location, has an abbreviated name for the corresponding address, i.e., a register number. The location of these workspace registers depends upon the contents of the workspace pointer register (as discussed in Chapter 1).

Depending on the instruction, either the source operand or the destination operand (or both) can use register direct addressing.

3.3 INSTRUCTION SUBSET 1A

As each of the instructions is presented in this book, an instruction summary is provided in accordance with a standard instruction summary format indicating: (1) the instruction name, (2) the mnemonic opcode, (3) the assembly language format, (4) the machine code format, (5) the instruction length, (6) the abbreviated indication of the result, and (7) the status bits affected. The instruction summary for the Move Word instruction is presented as a model (Instruction Summary 3-1). An explanatory item is provided for each of these parts of the format. Each part of the standard instruction format is keyed to an explanatory item by the number in parentheses. These numbers in parentheses are not part of the standard format, but are provided as a convenience to the reader.

(1) The instruction name presents the name of the instruction (at upper left).

(2) The mnemonic opcode presents the standard abbreviation (assembly language opcode) for the instruction (at upper right).

(3) The assembly language format (CODE) presents the opcode and operand format for this instruction's implementation in assembly language.

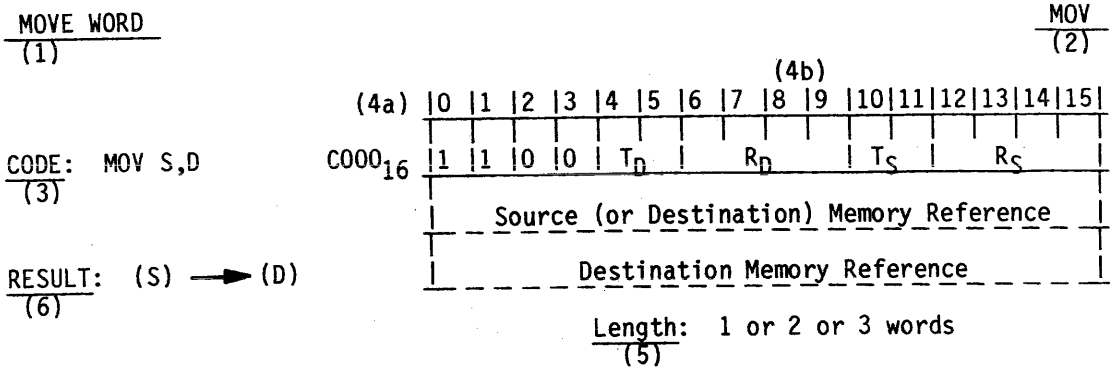
In the assembly language format, there are certain operand symbols and key words:

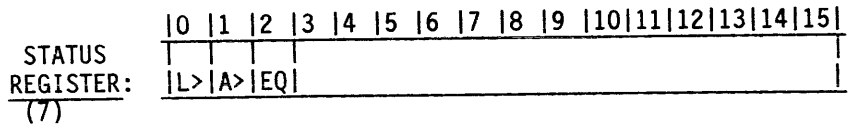
S	indicates general* source operand
D	indicates general* destination operand
R	indicates workspace register only
IOP	indicates immediate operand
C	indicates count (a value from 0 to 15)
Displacement	indicates relative displacement (PC relative or CRU relative).
Location	indicates the target address for a jump instruction.

*This means all five general addressing modes (register direct, register indirect, register indirect autoincrement, symbolic memory, and indexed symbolic memory). These are explained later in this section.

MOVE WORD (1)

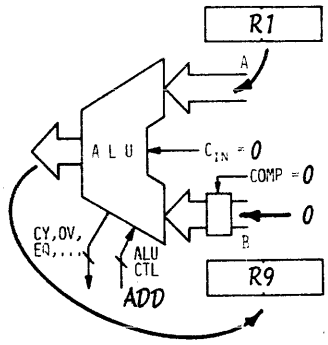
MOV (2)





OPERATION: (8)

The destination operand is replaced with a copy of the source operand (16 bits). The operand is compared with zero, and the L>, A>, and EQ status bits are affected accordingly.



NOTES: (9)

This MOV instruction allows the operator to move a 16-bit word from one location to another. It permits moving a word from one general memory location to another general memory location in one instruction. This is one of the key aspects of the advanced architecture of the 990/9900 computer family.

The MOV instruction is used frequently to set up counters, to save results, and to initialize various items at the start of software routines. Note that it is called MOV, but it really is a copy instruction.

The store and load instructions used in other computers represent a subset of this instruction.

Example:

MOV R1,R9 Machine Code: C241

before:	after:
(R1) = >1234	(R1) = >1234
(R9) = >FFFE	(R9) = >1234
	L>=1, A>=1, EQ=0

Instruction Summary 3-1

(4) The machine code format for this instruction is in two forms: summarized and detailed.

a. The summarized format presents the machine opcode.

b. The detailed format presents the machine opcode in binary and indicates the location and contents of each of the related fields. In the detailed machine format, there are symbols used for the various fields (in addition to those used in assembly language format):

T_S -- T field for the source workspace register (indicator of which general mode of addressing)
R_S -- Source workspace register field
T_D -- T field for the destination workspace register (indicator of which general mode of addressing)
R_D -- Destination workspace register field.

Also the detailed format indicates the number of words (in machine code). Dashed lines indicate possible words, depending on choice of addressing mode. If there are references to memory, then the source reference will precede the destination reference.

(5) The instruction length (LENGTH) specifies the various possible lengths (1 word for register-to-register addressing, 2 words for register-to-memory addressing, and 3 words for memory-to-memory addressing).

(6) The abbreviated indication of result (RESULT) indicates, in "shorthand" style, the result of the operation.

Parentheses mean "contents of," i.e., (S) means contents of the source.

(7) The status bits affected (STATUS REGISTER) indicates which status bits are affected as a result of the instruction.

The operation result is compared with zero to affect the L>, A>, and EQ status bits, unless the instruction operation description specifically indicates otherwise. The C and OV status bits are affected as a result of arithmetic operations as discussed in Chapter 2, unless specifically indicated otherwise.

(8) The instruction definition (OPERATION) verbally explains the operation of the instruction, and, as appropriate, a figure showing the specific inputs, outputs, and control signals presents the related ALU operation.

(9) The notes (NOTES) provide application notes, examples, typical uses, and programming hints.

EXAMPLE: MOV R7,R9

MNEMONIC CODE		MACHINE CODE					
MOV R7,R9	=	C247					
		OPCODE	DESTINATION FIELD		SOURCE FIELD		
		1100	T_D	R_D	T_S	R_S	
MOV S,D	=	1100	xx	xxxx	xx	xxxx	= CXXX
MOV R7,R9	=	1100	00	1001 (R9)	00	0111 (R7)	= C247

FIGURE 3-2. Machine Code Format for MOV Instruction

Machine Code Format Example

Machine code formats must be understood to successfully use machine code. The machine code for the MOV instruction is presented in Figure 3-2 as a primary example of machine code formats for the TMS 9980A instruction set.

From Instruction Summary 3-1 and Figure 3-2, note that the MOV instruction has a 4-bit field for the operation code (C) and 12 additional bits to be defined. These undefined bits can be subdivided into two fields: six bits for the source and six bits for the destination. Each of these 6-bit fields can be further subdivided into two fields: four bits for the register field and two bits for the T field. This format is common to 11 other instructions in the TMS 9980A instruction set. The operation code appears in the leftmost portion of the instruction word. In the case of the MOV instruction, this will be a hexadecimal C or binary 1100. In the MOV instruction, the rightmost six bits are for the source. Of these six bits, the rightmost four bits will be the source-register field, and the preceding two bits are for the source T field which indicates how the source register is used. In the first example, Figure 3-2, the register field is used as register direct. The T-field indicator for this is binary 00. There are several other ways in which this register can be used. In this example (MOV R7,R9) the source is register 7, so the 4-bit source-register field (at the right) contains a 7 (binary 0111). Since the register is being

used as register direct, its related T field is also binary 00. Furthermore, the destination-register field contains 9 (binary 1001) for register 9. Since the destination register is also used as register direct, its related T field is also binary 00.

With reference to the T field, a binary 00 always indicates register direct addressing. This is indicated in mnemonic code by having no character preceding the register symbol (R7 is a predefined symbol for register 7 in this application). It is not necessary for the source and destination operands to use the same addressing mode. The options depend upon the particular instruction. One can see that there are at least four addressing modes based on the fact that the T field can take on one of four values (of which only one has been discussed). For the instruction expressed in mnemonic form (MOV R7,R9), the hexadecimal machine code is C247. Notice that the source register in the machine code can be read easily, while the destination register cannot be read as easily since it is "split" between two hexadecimal digits.

Instruction Survey

Before introducing the other instructions to be considered in this chapter, it would be helpful to present a brief survey of the arithmetic instructions, the data manipulation instructions (Load, Move, Store, and Swap) and shift instructions. (See Figures 3-3 and 3-4.)

There are arithmetic instructions to add and subtract both word and byte values. Also, there are multiply and divide instructions (both in microcode). Furthermore, there are some other arithmetic instructions to perform special arithmetic operations.

With regard to data manipulation instructions, either word or byte values can be moved. Shift instructions provide for shifting bits to the left and right within registers.

Detailed Description of Instructions

The ADD WORDS Instruction. The word instruction for add can be remembered easily because it is simply represented by the mnemonic code, A, as shown in Instruction Summary 3-2. Both source and destination have general addressing capabilities. To produce the machine code for the instruction A R3,R4, the bit pattern is composed from right-to-left as follows. The source register field is 3 (binary 0011) and its related T-field for register direct is binary 00. Further, the destination register field is 4 (binary 0100) and its related T-field for register direct is binary 00. The operation code (the next four bits) is a hexadecimal A (binary 1010). Thus, the machine code for this instruction is $1010\ 00\ 0100\ 00\ 0011_2$ or >A103. To verify the A R3,R4 instruction, the reader can enter the following sequence of machine codes as a machine language program starting at location >380.

Location	Code	Mnemonics
380:	02E0	LWPI >0300
382:	0300	
384:	A103	A R3,R4
386:	0340	IDLE

Initialize registers 3 and 4 by entering the number 2 in memory location >306 and the number 3 in memory location >308. Execute the program starting at location >380, and inspect the result at memory location >308 (register 4). It should contain 5 (3 + 2 = 5).

The SUBTRACT WORDS Instruction. The subtract instruction is another instruction that performs arithmetic operations on 16-bit numbers, as shown in Instruction Summary 3-3. The reader should note carefully which number is subtracted from which. Of course, in the addition operation the order does not matter. In subtraction, the source is subtracted from the destination, and the result replaces the destination.

To illustrate, the instruction S R3,R4 is used. The reader can enter the following sequence of machine codes as a machine-language program starting at location >380.

Location	Code	Mnemonic
380:	02E0	LWPI >0300
382:	0300	
384:	6103	S R3,R4
386:	0340	IDLE

To perform this example program, initialize registers 3 and 4 by entering a number such as 2 in memory location >306 (register 3) and a number such as 3 in memory location >308 (register 4). Execute the program starting at location >380. Inspect the result at memory location >308 (register 4); it should contain 1 (3 - 2 = 1).

3.4 JUMP ADDRESSING AND RELATED INSTRUCTIONS

It is unlikely that a program will be written in assembly language which will not contain a jump instruction. Thus, it is very important for the operator to understand how jump instructions work. There are 13 different jump instructions, but the addressing scheme for each is exactly the same. Jump addressing allows the program to change the program counter within a range of +127 to -128 words (or +254 to -256 bytes). In brief, the program counter can shift its contents by a certain "reach" with each jump instruction. This is called "relative addressing" since the destination is not absolute but is relative to the jump instruction's location.

ARITHMETIC INSTRUCTIONS

ADD	WORDS	A	S,D	(A000)
		AI	R,IOP	(0220)
	BYTES	AB	S,D	(B000)
SUBTRACT	WORDS	S	S,D	(6000)
	BYTES	SB	S,D	(7000)
MULTIPLY		MPY	S,R	(3800)
DIVIDE		DIV	S,R	(3C00)
INCREMENT BY ONE		INC	S	(0580)
INCREMENT BY TWO		INCT	S	(05C0)
DECREMENT BY ONE		DEC	S	(0600)
DECREMENT BY TWO		DECT	S	(0640)
ABSOLUTE VALUE		ABS	S	(0740)
CHANGE SIGN		NEG	S	(0500)

NOTE:

S indicates any general source (register or memory)
D indicates any general destination (register or memory)
R indicates any workspace register
IOP indicates an immediate operand
C indicates count

Figure 3-3 Survey of Arithmetic Instructions

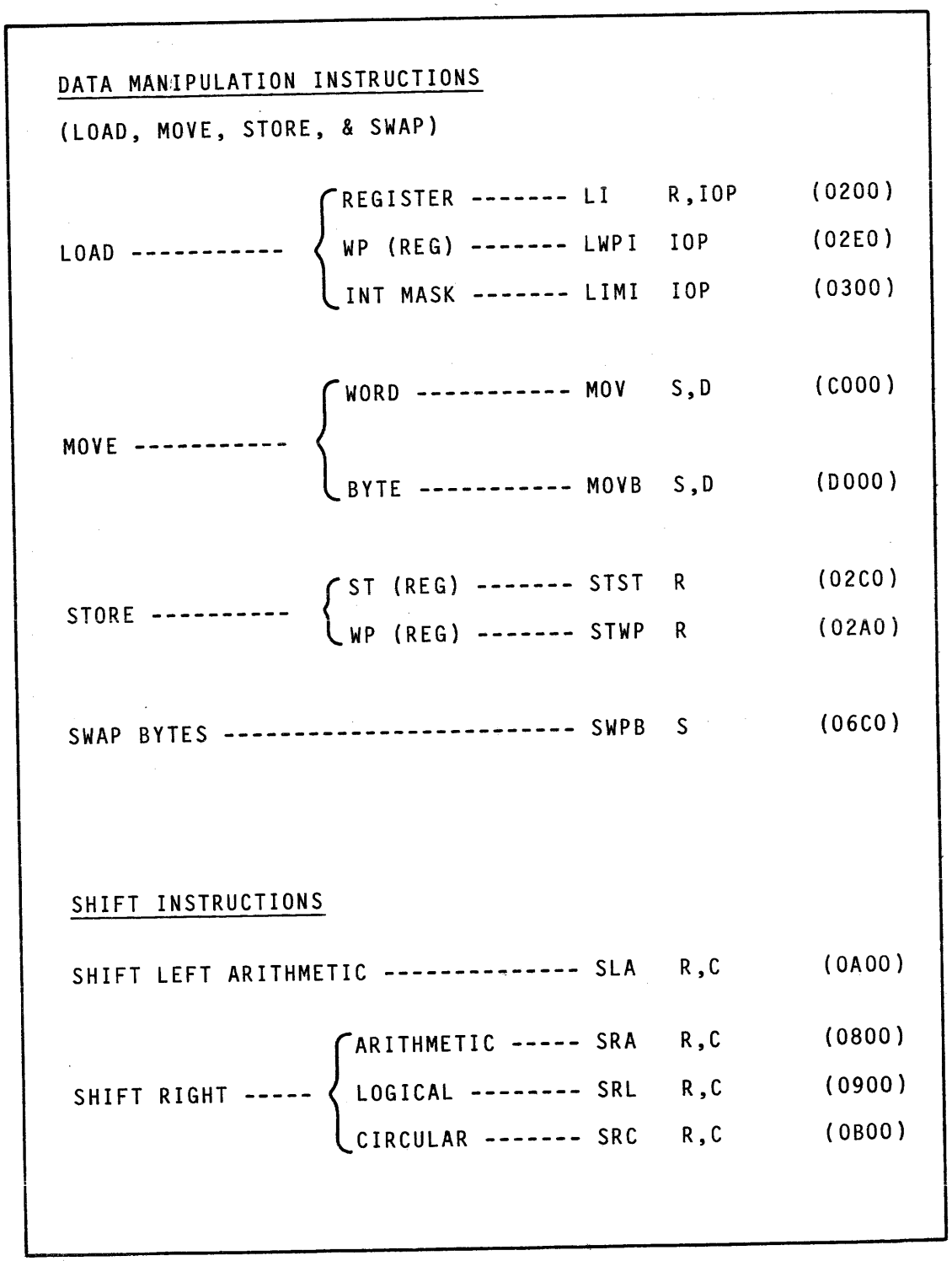
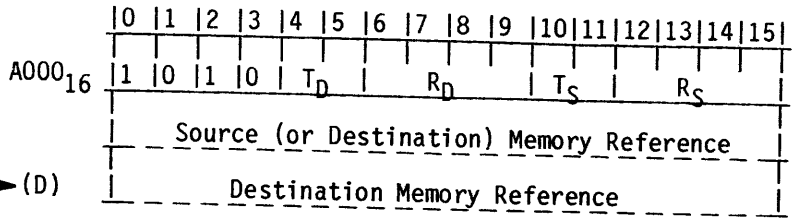


Figure 3-4. Survey of Data Manipulation and Shift Instructions

ADD WORDS

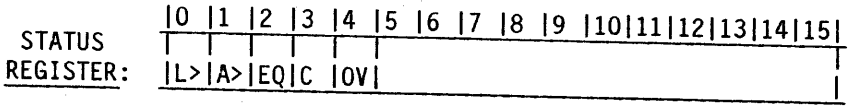
A

CODE: A S,D



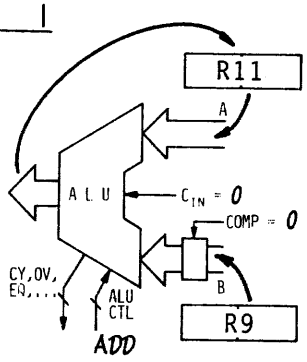
RESULT: (D) + (S) ⟶ (D)

Length: 1 or 2 or 3 words



OPERATION:

The source operand is added to the destination operand and the resulting sum replaces the destination operand. The 16-bit result is compared with zero, and the L>, A> and EQ status bits are correspondingly affected. The addition operation causes the C and OV bits to be affected.



NOTES:

The ADD instruction is used to take the sum of two 16-bit numbers. This instruction will be used quite frequently when arithmetic operations are involved. Take note that the source operand is added to the destination operand and the destination operand is changed. For example, assume that register 3 contains a number which is to be added to the number in register 4, so that the result will be located in register 4. Register 3 is said to be the source, and register 4 to be the destination. Write the mnemonic instruction A R3,R4 to produce the desired result. The format for the machine code is the same as that discussed under the MOV instruction (see Figure 3-2).

Example:

A R9,R11 Machine Code: A2C9

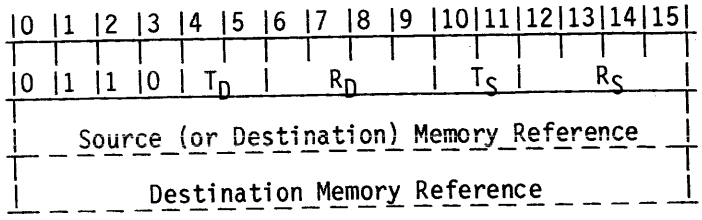
before:	after:
(R9) = >2468	(R9) = >2468
(R11) = >1234	(R11) = >369C
	L>=1, A>=1, EQ=0, C=0, OV=0

Instruction Summary 3-2

SUBTRACT WORDS

S

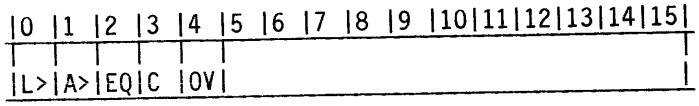
CODE: S S,D

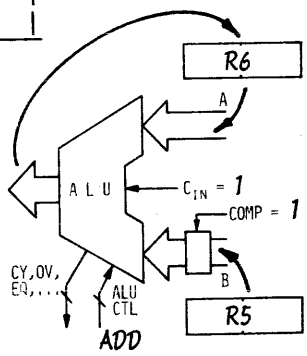


RESULT: (D) - (S) ⟶ (D)

Length: 1 or 2 or 3 words

STATUS REGISTER:





OPERATION:

The source operand is subtracted from the destination operand and the result replaces the destination operand. The 16-bit result is compared with zero, and the L>, A> and EQ status bits are correspondingly affected. The subtraction operation also causes the C and OV status bits to be affected.

NOTES:

The SUBTRACT instruction is used to take the difference of two 16-bit numbers. This instruction will be used quite frequently when arithmetic operations are involved. Take note that the source operand is subtracted from the destination operand and the destination operand is changed. For example, say that register 5 contains a number which is to be subtracted from the number in register 6. Register 5 is said to be the source, and register 6 to be the destination. Write the mnemonic instruction S R5,R6 to produce the desired result. The format for the machine code is the same as that discussed under the MOV instruction.

Example:

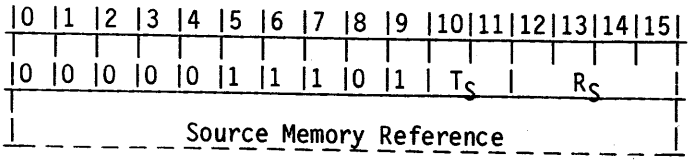
S R5,R6 Machine Code: 6185

before:	after:
(R5) = >1234	(R5)=>1234
(R6) = >369C	(R6)=>2468
	L>=1, A>=1, C=1, OV=0, EQ=0

ABSOLUTE VALUE

ABS

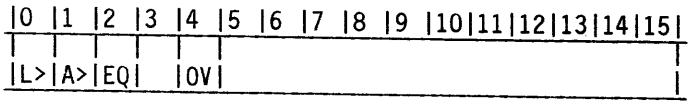
CODE: ABS S



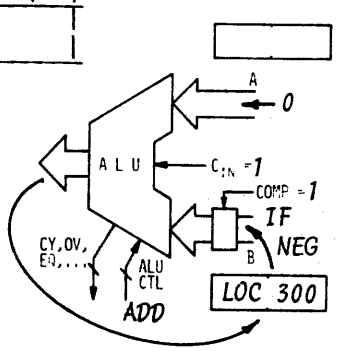
Length: 1 or 2 words

RESULT: |(S)| ⟶ (S)

STATUS REGISTER:



OPERATION:



Compute the absolute value of the source operand and replace it with this result. In other words, if the source operand is negative, replace it with its corresponding 2's complement; if positive, leave unchanged. The original source operand is compared with zero, and the L>, A>, and EQ status bits are correspondingly affected. If the result is > 8000, the overflow bit will be set to ONE.

NOTES:

The ABS instruction allows the operator to be sure that a number is positive. If a number is already positive, it will remain unchanged. If a number is negative, the CPU will attempt to make it a corresponding positive number. For example, a minus 2 will become a plus 2. There is one negative number which does not have a positive counterpart, namely >8000. If the absolute value instruction is attempted on >8000, the number is unchanged, but the overflow status bit is set to ONE.

Experiment with the following program to take the absolute value of the contents of location >0300:

Location	Machine Code	Mnemonic Instruction
380:	0760	ABS @>300
382:	0300	
384:	0340	<stop>

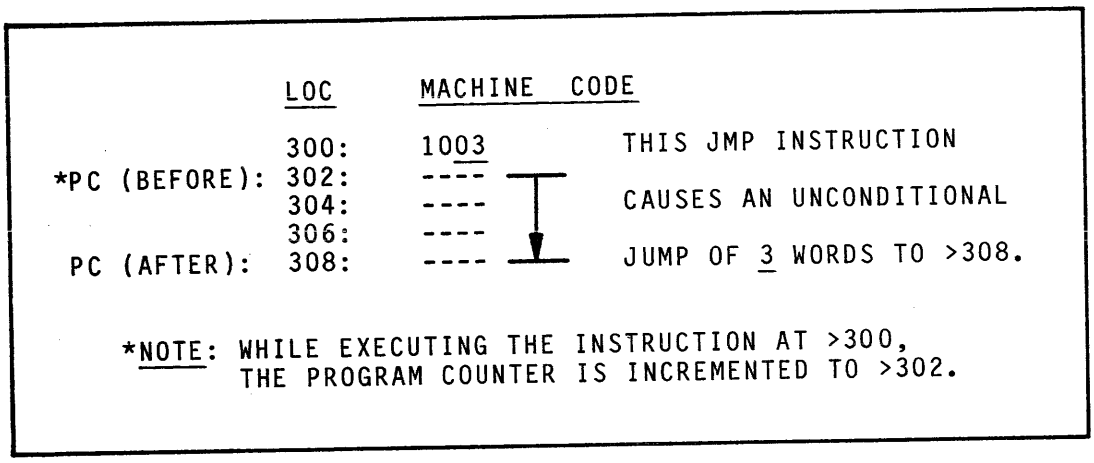


Figure 3-5. Jump Addressing Illustration

Machine Code Format

Each jump instruction requires one 16-bit word, and its displacement (or reach) is a signed 8-bit field. The contents of this 8-bit field will cause the program counter to be incremented or decremented by the amount of the machine code displacement (in words). For example, in Figure 3-5 the jump instruction is shown to cause the PC to increment by three words (or six bytes) from location >302 (the next instruction) to location >308. If the displacement were zero, the program counter would simply point to the next instruction since the PC is always automatically incremented to point to the next instruction. In effect, an unconditional jump instruction with a zero displacement is a NO Operation (NOP) instruction.

Assembly Code Format

Assembly code for jump instructions has two formats. The first of these is to "jump to" a location indicated by an absolute address, such as >0388, or by a label such as a Y (e.g., JMP >0388 or JMP Y). Obviously, the jump instruction's target location must be within the reach of the instruction. This format has the advantage that the programmer does not have to count the number of words of displacement. This counting task is handled by the assembler.

The second format is still a jump to a location, but it is a jump to the current location plus or minus a displacement. However, the displacement in the machine code format and the displacement in the assembly code format are not the same. This will be clarified with an example.

Referring to Figure 3-6, assume that a jump instruction is located at >0380 and a jump to location >0388 is desired. If location >0388 had a label of Y associated with it, one could write JMP Y

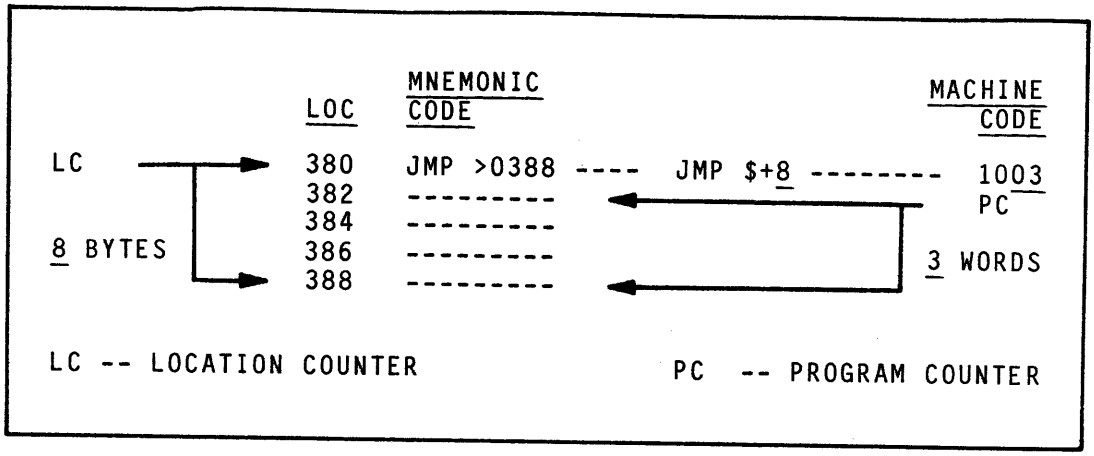


Figure 3-6. Comparison of Jump Addressing Formats

or, equivalently, JMP >0388. In assembly code the current location of the instruction (also referred to as the location counter, LC) can be represented by a dollar sign ($). In this instruction, the current location is >0380. Thus, using the second jump instruction format, the mnemonic code is JMP $+8, which is equivalent to JMP >0388. The three instruction formats, JMP Y, JMP >0388, and JMP $+8 are all equivalent if the instruction is located at at >0380.

It can be seen further from Figure 3-6 that the program counter starting at location >382, increments by one word to >384, by two words to >386, and by three words to >388. The machine code displacement is therefore three words. Note in Figure 3-6 that the location counter for this instruction and the program counter differ by one word (or two bytes). Note further that the assembly code displacement is in bytes whereas the machine code displacement is in words. Thus, the eight bytes in the assembly code displacement format are equivalent to the three words in the machine code displacement. This can be derived by converting the eight bytes of assembly code displacement to the equivalent four words and then subtracting one word due to the fact that the location counter and the effective program counter reference differ by one word. That is, eight bytes of assembly code displacement divided by two minus one word equals three words of machine code displacement (8/2 - 1 = 3). In other words: Location Counter + assembly code displacement (in bytes) = Program Counter + 2 times machine code displacement (in words).

Most programmers prefer not to have to count these bytes and words. Assuming that an assembler is available (as it is with the TM 990/189), the programmer can always write a jump to a labeled location (such as JMP Y). But one should still understand the machine code format of the jump instruction to be able to patch a program.

Survey of Jump Instructions

As was mentioned, there are 13 different jump instructions. One, the JMP instruction, is unconditional, which means that the jump will be executed regardless of the condition of the status register. In contrast, the other 12 jump instructions require the CPU to test the condition of one or more status bits. The CPU will then execute the jump if the specified condition is met. Otherwise, it will simply step to the next instruction. There are six status bits tested by the 12 conditional jump instructions. These status bits are: logical greater than (L>), arithmetic greater than (A>), equal (EQ), carry (C), overflow (OV), and odd parity (OP). The 13 jump instructions are listed and categorized in Table 3-2.

> NOTE: The length of the jump is a signed displacement in words maintained in the eight least-significant bits of the machine code. When a jump is to occur, this signed displacement is translated into bytes and added to the program counter value (in bytes) in order to determine the address of the next instruction. This displacement range, or "reach" of the instruction, is from -128 to +127 words (-256 to +254 bytes) from the program counter as it points to the next instruction following the jump.

It is obvious from Table 3-2 that there are many kinds of jump instructions. This table summarizes the 13 jump instructions and categorizes them by the status bits that are examined by each instruction. The only means by which the status bits can be tested directly, is with these instructions. Notice that each instruction has its machine code in parentheses beside it. The two leftmost hexadecimal digits are the 8-bit machine opcode for the instruction and the two rightmost "00" digits are reserved for the 8-bit displacement field. Each of these instructions is covered in more detail in this and subsequent chapters.

Regarding the 13 jump instructions listed in Table 3-2, note that some cause a jump to occur upon the condition of a single status-register bit. In other cases, a jump occurs upon a combination of status-register bits. Furthermore, note that there are certain jump conditions which may occasionally be desired for which jump instructions do not exist. These jump instructions can be derived by a combination of two or three existing jump instructions.

Detailed descriptions of the Unconditional Jump (JMP) and the Jump if Greater Than instructions are given in Instruction Summaries 3-5 and 3-6.

3.5 PROGRAMMING EXAMPLE

At this juncture, sufficient background has been presented to write the program code based on the programming idea indicated

Table 3-2. Jump Instructions Survey

LOGICAL CONDITIONS		(L> & EQ)
Low	- JL	(1A00)
Low or Equal	- JLE	(1200)
High	- JH	(1B00)
High or Equal	- JHE	(1400)

CARRY CONDITIONS		(C)
Carry	- JOC	(1800)
No Carry	- JNC	(1700)

ARITHMETIC CONDITIONS		(A> & EQ)
Less Than	- JLT	(1100)
Greater Than	- JGT	(1500)

OVERFLOW CONDITIONS		(OV)
No Overflow	- JNO	(1900)

EQUAL CONDITIONS		(EQ)
Equal	- JEQ	(1300)
Not Equal	- JNE	(1600)

PARITY CONDITIONS		(OP)
Odd Parity	- JOP	(1C00)

UNCONDITIONAL JUMP

JMP (1000)

NOTES:

1. Logical values do not have a sign bit.
2. Arithmetic values have a sign bit.
3. Odd parity occurs in byte operations with byte values having an odd number of ones.

CODE: JMP Location 1000_{16}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	0	0	0	Displacement (words)							

Length: 1 word

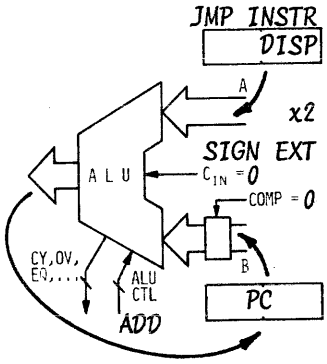
RESULT: (PC) + (Displacement in bytes) ⟶ (PC)

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

Add the signed displacement in bytes of the machine code instruction to the contents of the PC and place the sum into the contents of the PC.



NOTES:

The JMP instruction is used any time one wants to do a short-range unconditional jump. The following example illustrates two mnemonic instruction formats for the same instruction.

Example:

Location	Mnemonic Instruction	Alternate Mnemonic Instruction	Machine Code
>200:	JMP Y	JMP $+6	1002
	-----	-------	----
	-----	-------	----
>206:	Y -----	-------	----

Maximum Reach Backward

Loc	Mnemonic Code	Machine Code
202	----------	
300	JMP $-254	1080

Maximum Reach Forward

Loc	Mnemonic Code	Machine Code
300	JMP $+256	107F
400	----------	

The jump range is expressed as: (PC) - 128 words $\leq$ PC Result $\leq$ (PC) +127 words
or: (PC) - 256 bytes $\leq$ PC Result $\leq$ (PC) +254 bytes

JUMP IF GREATER THAN

JGT

CODE: JGT Location 1500_{16}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	0	1	Displacement (words)							

Length: 1 word

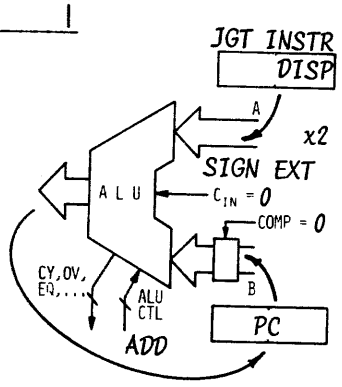
RESULT: If A > = 1, (PC) + Displacement in bytes ⟶ (PC)
If A > = 0, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

If the A> status bit is set to ONE, add the signed displacement in bytes (of the machine-code instruction) to the contents of the PC and place the sum into the PC; otherwise, leave the PC unchanged.



NOTES:

The JGT instruction is a conditional jump instruction which will be used quite frequently since it facilitates the implementation of a program loop.

This instruction tests the A> status bit. It can be used to determine whether a previous result is both positive and nonzero. It is quite useful in certain applications. For example, it provides a means of going through a loop a certain number of times. One could set a register to a specific value such as +10, then count down toward zero by ones and test the result each time until it is not greater than zero. This test is accomplished immediately following the subtraction from the loop counter by using a jump if greater than instruction. Recall that the subtraction operation will cause the status bits to be affected.

An example of its use in a program loop:

```
      LI   R1,1     PUT THE VALUE 1 IN R1
      LI   R9,8     LOAD R9 WITH THE VALUE 8        (Set loop counter)
LP    --------      *loop:                          (Perform loop operation)
      --------      *loop
      --------      *operation
      --------      *block
      S    R1,R9    SUBTRACT 1 FROM R9              (Decrement loop counter by one)
      JGT LP        JUMP TO LP IF LOOP COUNTER >0   (Continue loop if counter +,
```

earlier. Recall that the programming idea was to implement the function (FTN) indicated by the equation:

$$F(N1,N2) = [4\ (N1 - N2)]^2$$

Program Specification

It is important to have a programming specification which states precisely the programming objectives. The program specification for this function could be stated as:

> Calculate the function: $F(N1,N2)=[4(N1-N2)]^2$, where N1 and N2 are two signed 16-bit integers. Assume that the values for these integers will not produce an overflow. Place all initial values as required in workspace registers before the program is executed. Develop the program using as few instructions as possible without concern for error conditions. Provide the results of the function calculation in register 0 (R0).

Flowchart and Algorithm

Assuming one understands the function, the next step is to produce a logical and sequential representation of the actions to achieve the desired result for this program. One form of this is called a flowchart. It is a higher level graphical representation from which an operator can almost directly produce an assembly language program. In a flowchart, rectangles containing words are used to indicate specific operations, and diamonds to indicate conditional tests. These conditional tests have one entry point and at least two exits to indicate yes and no answers to the tests. Flowcharts use directed lines (arrows) to indicate the flow of the operation.

Prior to constructing the flowchart, the programmer should have in mind an approach for what must be accomplished to implement this programming idea. One approach, for example, is to simply take the difference of the two numbers which are in registers, multiply the result by four, and then take the result and multiply it times itself. One might ask how multiplication is to be done, since the multiply instruction has not been discussed yet. To accomplish multiplication by four, a number can be added accumulatively to itself four times. In the case of the second multiplication (or squaring), the number could be added to itself an appropriate number of times. For example, if the number to be squared is 12, 12 added accumulatively to itself 12 times is 144. This approach is an algorithm (method of attack) for this program.

At this point, the programming idea and program specification can be implemented in flowchart form. The program flowchart given in Figure 3-7 is oriented generically and independent of the instruction set to be used.

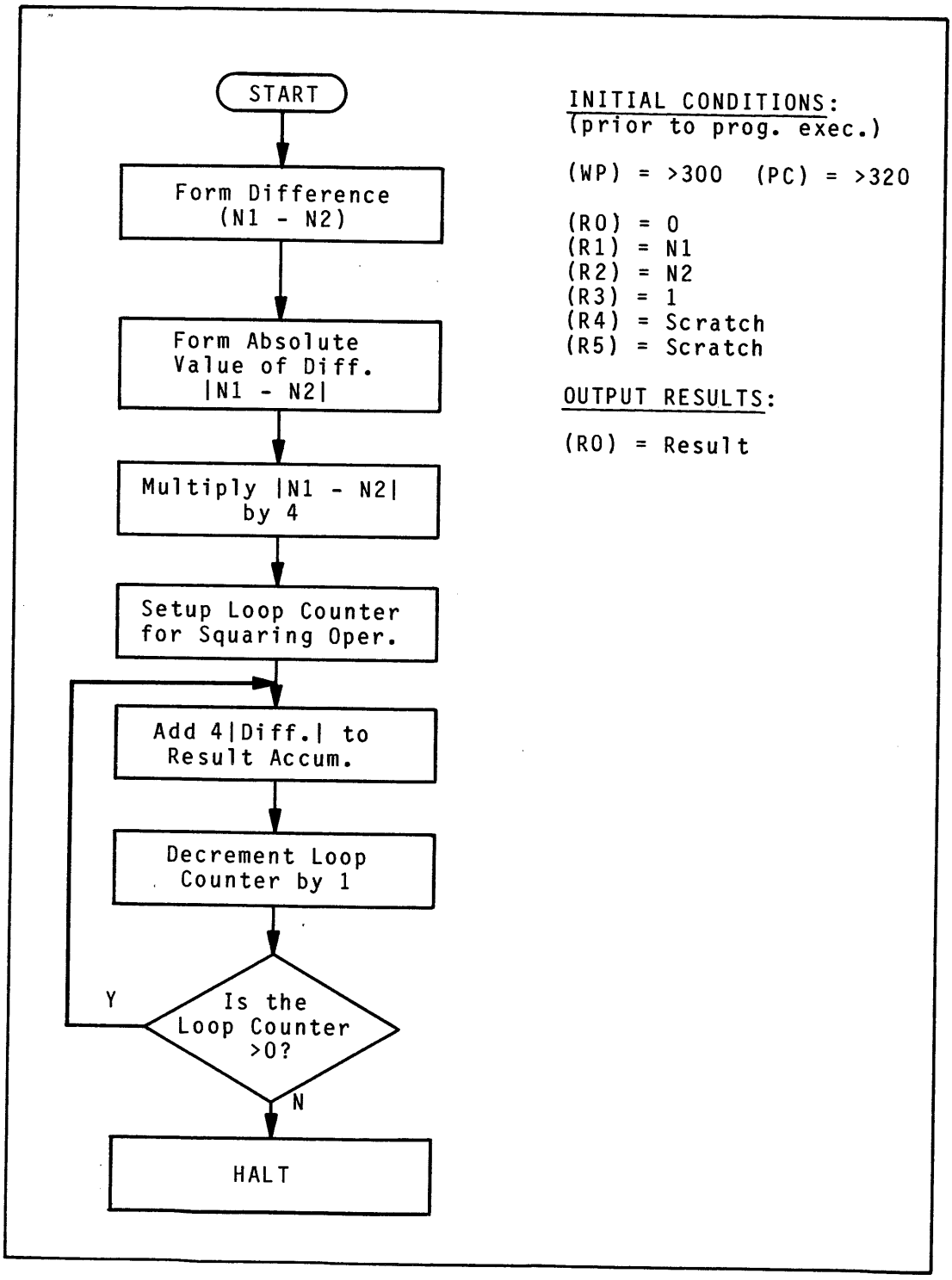


Figure 3-7. FTN1 Program Flowchart

Observe that certain initial conditions are assumed (see upper right-hand portion of Figure 3-8). Values for N1 and N2 must be assigned. Also, the result accumulator is initialized to zero. Two scratch registers are required to maintain intermediate program values and to preserve input data.

The first program step is to form the difference (N1-N2). The second program step is to take the absolute value of this difference. Since the result of a squaring operation is positive and this difference may be negative, taking an absolute value at this point ensures a positive result at program completion. The third program step is to multiply by four. This is done by adding the number to itself and then adding that result to itself. The fourth program step is to set up a loop counter equal to the result of four times the difference. The fifth program step is to add this result into the result accumulator and decrement the loop counter until the loop counter is zero. When this occurs, the program halts with a jump instruction to itself.

Figure 3-8 is a code sheet showing the resulting mnemonic code program with the corresponding machine code.

The reader can try hand-assembling this program and compare the resulting machine code with that shown in Figure 3-8. Then the machine code along with the initial conditions (indicated in Figure 3-7) can be entered and the program executed. Suggested values for N1 and N2 are 5 and 3. The result found in R0 should be >0040 (64_{10}).

Considering this program's implementation, it is important to observe the initial conditions. The initial MOV instruction was done to allow preservation of the initial value of N1. If the programmer desired to shorten the program and use fewer registers, he could delete the first instruction and could substitute R1 for R4 and R2 for R5. Further, as additional instructions are introduced, more elegant implementations can be considered.

Note that in the process of producing this program, the following steps were used: programming idea, program specification (including related algorithm), program chart (e.g., a flowchart), and program encoding.

3.6 COMPUTER SYSTEM CONCEPTS REVISITED

As discussed in Chapter 1, a computer system can be viewed as having three major parts: processor (or CPU), memory, and input/output (devices).

The processor acts as the central administrator and executes the program. The memory stores the program as well as providing data storage and buffering. Program memory may or may not be changeable during program execution.

FTN1 PROGRAM: $F(N1,N2)=(4*(N1-N2))^2$

MNEMONIC CODE	COMMENT	LOC	CODE
ST MOV R1,R4	MOVE N1 TO R4 (TEMP.)	0320	C101
S R2,R4	FORM DIFF: N1 - N2 IN R4	0322	6102
ABS R4	FORM ABS VALUE OF DIFF	0324	0744
A R4,R4	*MULTIPLY |DIFFERENCE|	0326	A104
A R4,R4	*BY 4	0328	A104
MOV R4,R5	SET UP LOOP COUNTER IN R5	032A	C144
LP A R4,R0	ADD 4|DIFF| TO RESULT ACCUM	032C	A004
S R3,R5	DECREMENT LOOP COUNTER BY 1	032E	6143
JGT LP	IF THE LOOP COUNTER >0, JUMP	0330	15FD
HL JMP HL	OTHERWISE, HALT	0332	10FF

INITIAL CONDITIONS:

(WP) = >300 (R1) = N1
(PC) = >320 (R2) = N2
(R0) = 0 (R3) = 1

(R4) & (R5) = SCRATCH

OUTPUT RESULTS:

(R0) = RESULT

Figure 3-8. FTN1 Program: Mnemonic and Machine Code

The input/output provides contact with the external "world" to receive and transmit information. Some examples of input/output devices are a printer, a keyboard, a visual display, a motor, and a magnetic-tape drive. A computer system without input/output would not provide any usable results.

The key concept behind a computer system is that of a stored program. The processor performs under control of instructions stored in memory. Its performance can be altered by changing these instructions without rewiring the hardware. Furthermore, these instructions can be permanently stored in a type of memory which will not lose information when power is removed.

In connection with the stored-program concept, there are three terms which need to be defined: hardware, software, and firmware. Hardware can be defined simply as those items which can be seen and touched such as printers, computer boards, IC chips, transistors, wire, chassis, visual displays, and other components.

Software refers to the program or sequence of instructions written to cause the computer system to perform some function. It is called "soft"-ware because typically it can be considered as a separate entity from the hardware, and it cannot be directly seen or touched when stored in the machine. It can be changed easily (thus the "soft" reference), and may even be lost, as when power is removed from certain types of volatile storage media. Firmware is a concept which fits in between hardware and software. One definition of the term "firmware" is computer program instructions stored in memory such that they will remain unchanged during execution and will be retained when power is removed. This generally means the program is stored in ROM (as opposed to RAM); thus it is "firm"-ware. Firmware cannot be changed as easily as software.

One should recognize that the boundaries between these terms are not always clearly observed. Sometimes firmware is included with hardware; sometimes it is even loosely referred to as software. Sometimes firmware is specifically used to refer to microprogram code. Sometimes, and definitely incorrectly, programming of microcomputers is referred to as microprogramming.

3.7 REGISTER ADDRESSING MODES

Register direct addressing, explained in section 3-2, accesses the contents in the register or registers specified. One example is MOV R1,R2 where the contents within register 1 is placed into register 2. There are two other types of register addressing used within this instruction set:

- Register indirect
- Register indirect autoincrement

In the first of these, register indirect addressing, the contents of the register are not used as an operand; instead, the contents

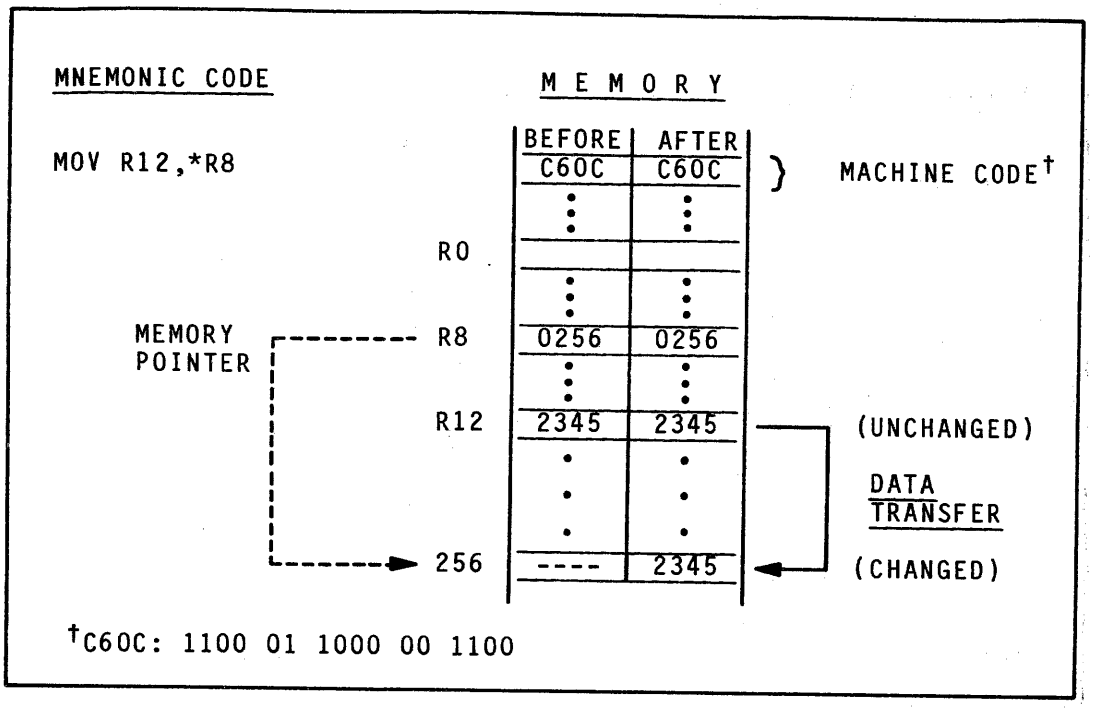


Figure 3-9. Register Indirect Addressing Illustration

of the register are used as a pointer to the operand (in memory). This type of addressing can be used for either a source or a destination operand, or both.

To illustrate its use in connection with a destination operand, consider a postman coming to a house to deliver a package. The house corresponds to a register and the package to an operand. In illustrating register direct addressing, the postman simply delivers the package to the house. In contrast, to illustrate register indirect addressing, the postman looks in the mailbox and finds a card which says, for example, "Deliver the package to 1234 Maple." In the case of register indirect addressing, the contents of the register, as with the card, points to the desired destination.

A more specific example is depicted in Figure 3-9. One may wish to move a number from register 12 to the memory location pointed to by the contents of register 8. If register 12 contains the number >2345 and register 8 contains the number >0256, then the CPU transfers the number >2345 from register 12, not to register 8, but to the memory location >0256 pointed to by the contents of R8. This instruction is written in mnemonic code as: MOV R12,*R8. Note that with the destination operand there is an asterisk prefix. This prefixed asterisk indicates that the register is to be used as register indirect (memory pointer), rather than as register direct. In machine code, this asterisk corresponds to a T field of binary 01. To translate into machine code the mneumonic instructions MOV R12,*R8, one

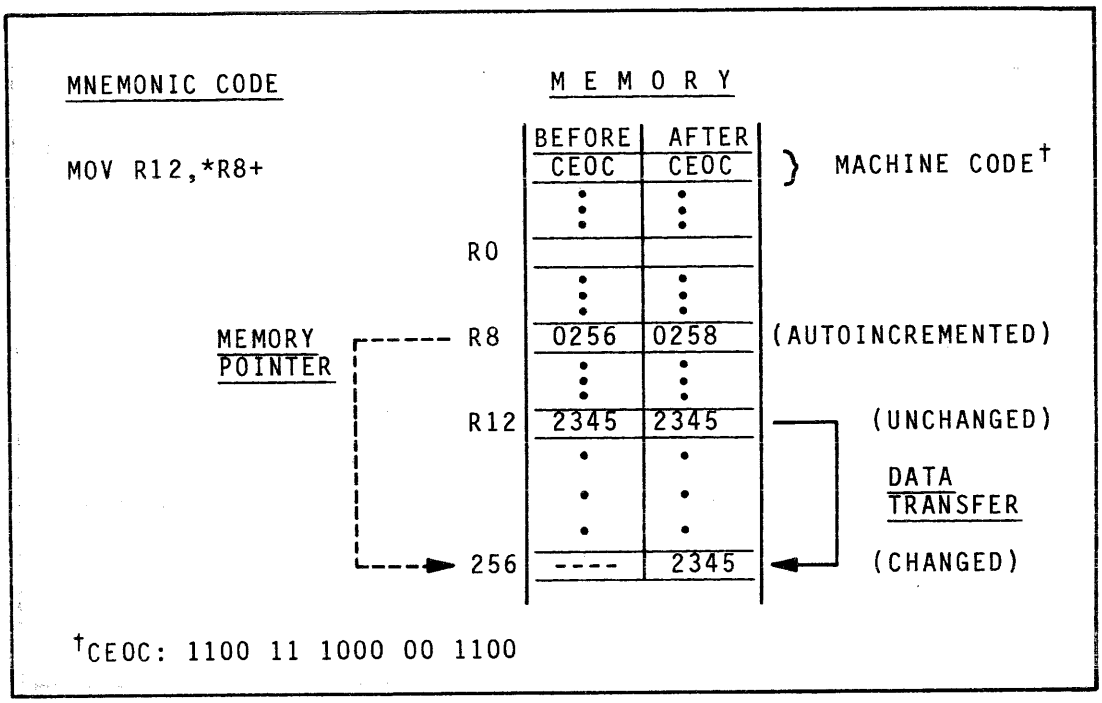


Figure 3-10. REGISTER INDIRECT (Autoincrement) Addressing Illustration

would, going from left to right (see Figure 3-9), use a binary 1100 (>C) for the operation code field, then a binary 01 (register indirect) for the destination T field, a binary 1000 (R8) for the destination register field, a binary 00 (register direct) for the source T field, and, finally, a binary 1100 (R12) for the source-register field. Altogether, the machine code in hexadecimal is C60C.

The other type of register indirect addressing is called register indirect autoincrement. Notice that the name of this third type of register addressing is very similar to register indirect addressing, but has the additional feature of being autoincrement. To observe the one item of difference, consider the previous example with register indirect autoincrement. It is written in mnemonic code as: MOV R12,*R8+. The "+" suffix indicates autoincrement and applies only to register indirect. Figure 3-10 illustrates this addressing mode.

Using the same numbers again one can see that the number >2345 will move from register 12 to memory location >0256 pointed to by the contents of register 8. Up to this point, it is exactly the same as register indirect addressing, but there is one more step, the autoincrement feature. After using R8 as a pointer, the CPU automatically increments the contents of the specified register. In this case, R8 is incremented by 2, i.e., from >0256 to >0258, and now points to the next word in memory. The CPU increments by 2 because it is a word instruction, i.e., it operates upon 16 bits.

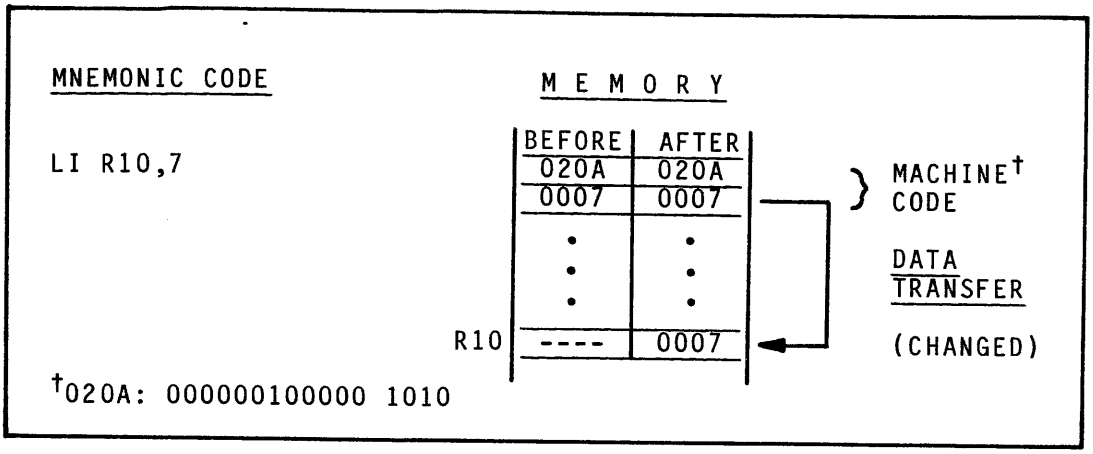


Figure 3-11. Immediate Addressing Illustration

On a similar instruction of the byte type, the autoincrement is by 1. This type of incrementing allows the programmer to index automatically through byte-oriented tables with byte instructions and through word-oriented tables with word instructions.

Note in Figure 3-10 that the T field for register indirect autoincrement contains binary 11.

Thus, there are three types of register addressing. For instructions where there is a choice of general source or destination addressing, register direct addressing is indicated by a T field of binary 00, register indirect by a T field of binary 01 and register indirect autoincrement by a T field of binary 11 (see Table 3-3).

3.8 IMMEDIATE ADDRESSING

Another type of addressing is called immediate. With it, the numerical constant (operand) to be used by the instruction is located within the instruction. If the programmer desires to load the constant, 7, immediately into register 10, then he writes a load immediate instruction as LI R10,7. With this instruction, the first word of the machine code indicates the operation of load immediate into register 10. The immediate operand, 7, is in the second instruction word. The operation is portrayed in Figure 3-11.

Note that register 10 is indicated by the >A in the right four bits of the first machine code instruction word and the immediate operand (7) is the second 16-bit machine code instruction word. The immediate operand for the load immediate instruction can be any 16-bit number (signed or unsigned). This is the same as a copy function except with immediate addressing. The copying in this case is done from a word in the instruction to a register.

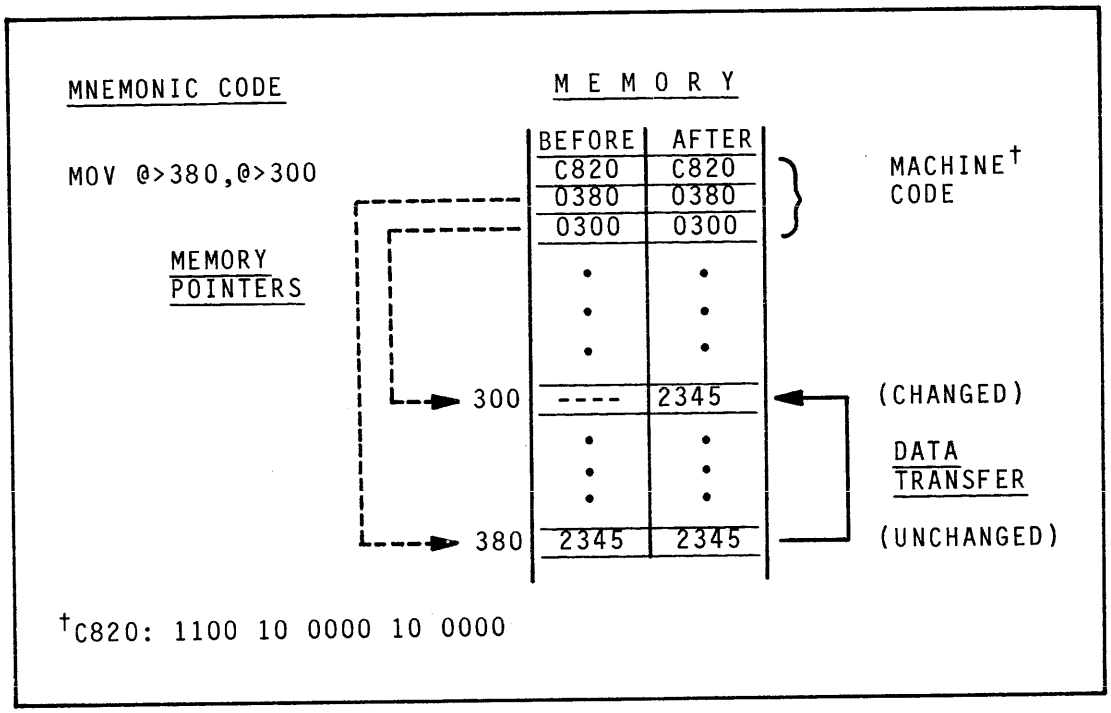


Figure 3-12. Memory Symbolic Addressing Illustration

3.9 SYMBOLIC MEMORY AND INDEXED ADDRESSING

There are two additional subcategories of addressing which need to be mentioned, both of which come under the general category of memory addressing:

- Symbolic memory (not indexed)
- Symbolic memory indexed

With symbolic memory addressing, one can, for example, move an operand from one general memory location to another with one instruction. To move a number from memory location >0380 to memory location >0300, simply write MOV @>0380,@>0300 (note Figure 3-12). In mnemonic code, the "@" symbol is used to explicitly indicate memory addressing, thus clearly distinguishing between a memory address value and a register value.

If the programmer had attempted to write register >0380 as part of an instruction, it would have been an error because the register numbers go only from 0 to 15. Note that there is a register number 10 as well as a memory location 10, so that the following operands refer to different types of addressing:

- 10
- *10
- *10+
- @10

In particular, the first of these items indicates that the operand is contained in register 10 (mnemonic code defaults to decimal). The second item (register indirect) indicates that the operand is located in a memory location pointed to by the contents of register 10. The third item is the same as the second except that the contents of R10 are automatically incremented during instruction execution. The fourth and last item (symbolic memory) indicates that the operand is located at memory location 10. When there is an option as to which type of general addressing is to be used, the T-field code, summarized in Table 3-3, indicates to the CPU which is to be used.

Table 3-3. T-Field Indicators

Addressing Mode	T Field	Mnemonic Equiv.
Register direct	00	RX
Register indirect	01	*RX
Memory	10	@LOC
Register indirect autoincrement	11	*RX+

The second subcategory of symbolic memory addressing is called symbolic memory indexed addressing. In this addressing mode, the symbolic memory address can be modified by adding to it the value in a designated register. In assembly language, the register is specified in parentheses following the symbolic memory address:

```
MOV @>380(R8),@>300
```

In this case, a word will be moved into memory address >0300 from an effective memory address calculated by adding the contents of register 8 to the value >380.

For this instruction the concern is for the contents of the effective source address. The effective source address in this case is the number >0380 plus the contents of register 8. Thus, the instruction is pointing to a table starting at >0380 offset by the number contained in register 8 (the related index register). If the register contains the number 4, the effective source address is >0380 + 4 or >0384, and it is the contents of location >0384 which are moved to memory address >0300.

This mode of addressing enables the programmer to access tables more easily. For example, suppose there is a table, organized into two-word blocks, from which the programmer wants to access the first word of each block. The table begins at memory address >0380. The programmer can designate successive memory locations >0380, >0384, >0388, etc.; however, in writing an algorithm to repetitively access every other word in the table, some complicated means must be used to change the memory address each time. Indexing makes this memory address modification easy. By using a base address

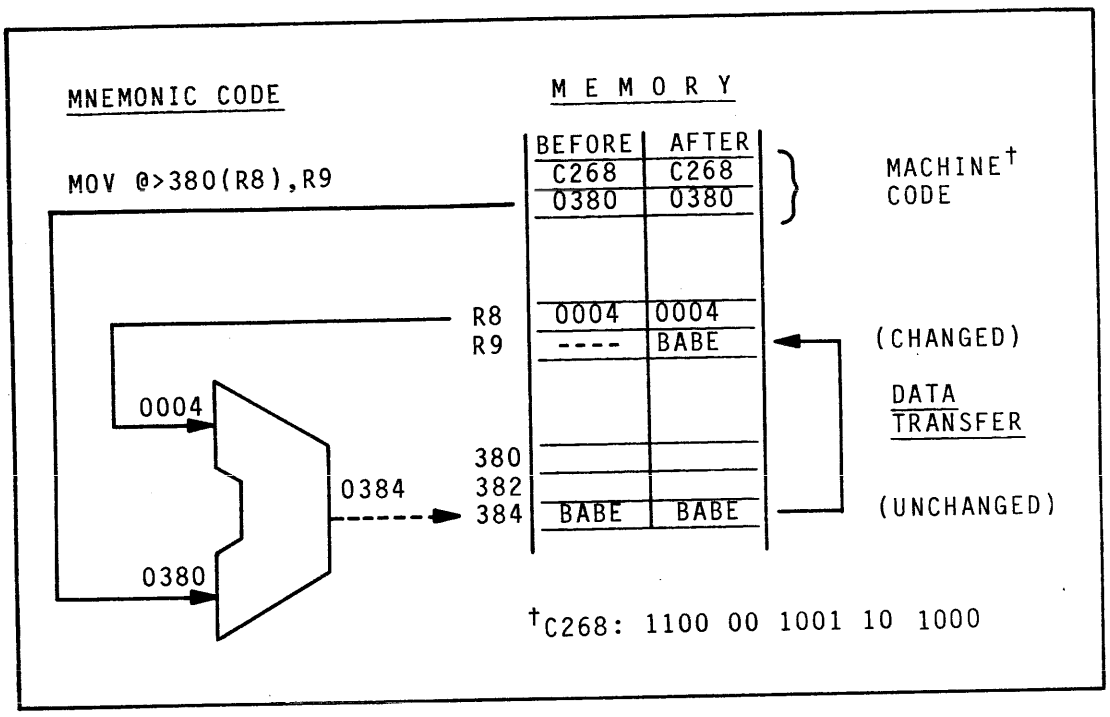


Figure 3-13. Memory Indexed Addressing Illustration

(pointing to the starting address of the table) and an index register (providing the increment value), the start of each block is accessed by a single instruction with the index register being updated before each access. By adding 4 to the index register each time, the desired word can be moved and acted upon by the algorithm. Access is made to the first word of each two word block in the table as shown in Figure 3-13.

Thus, indexing allows for flexible programming by inclusion of a variable into the instruction. By arranging for an index register to have a different value (or offset) each time the instruction is executed, the instruction accessed a different word in the table each time it is executed.

3.10 ADDRESSING SUMMARY

The five general modes af addressing plus immediate addressing are summarized with examples in Figure 3-14.

Copy Function Revisited

To summarize the addressing modes, reconsider the copy function. Figures 3-15 through 3-20 illustrate the operations of the six modes of addressing.

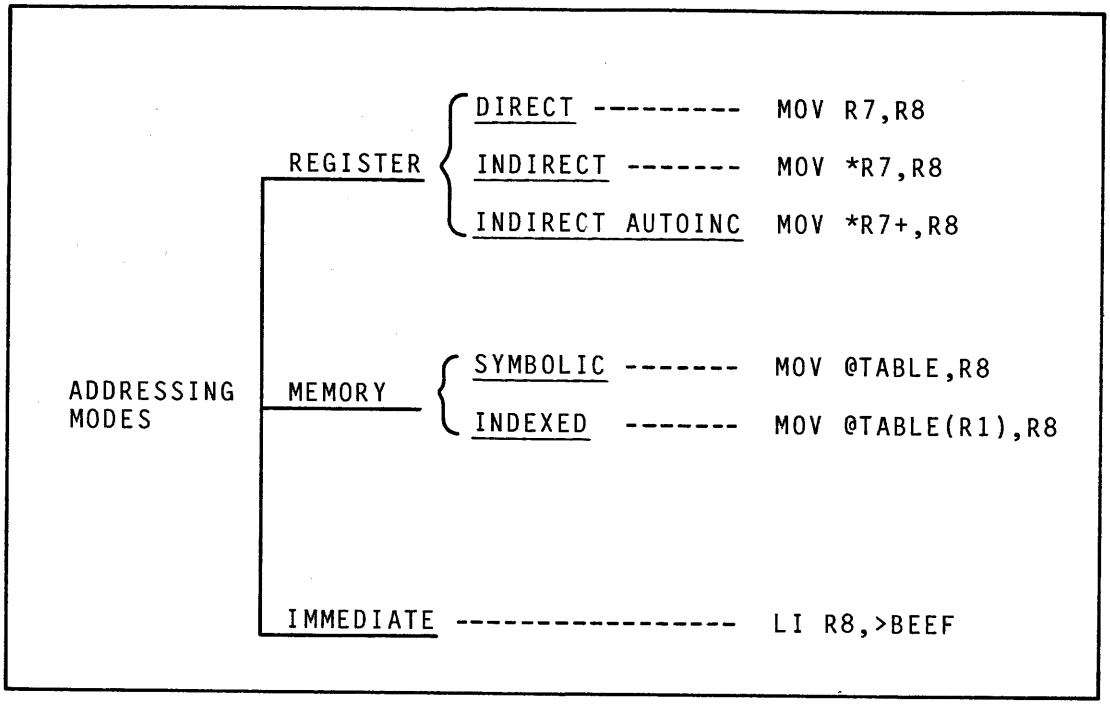


Figure 3-14. Summary of Addressing Modes

3.11 INSTRUCTION SUBSET 1B

Load Immediate Instruction

The Load Immediate (LI) instruction is the first of seven instructions having an immediate operand in the second 16-bit word. Each of these seven instructions is easily identified since each contains the word "immediate" in the name of the instruction, and each mnemonic code has an "I" as the final letter. The symbol "IOP" is used in the instruction summaries to represent an immediate operand.

The Load Immediate instruction, shown in Instruction Summary 3-7, is frequently used to initialize various registers at the outset of a program. During the course of a program, if the programmer desires a constant to be placed in a specific memory location, he would likely use a Load Immediate instruction with a register and then move the register contents to the memory location. Note that a Load Immediate instruction always transfers a constant from the instruction to a register; thus, only register direct addressing is being used for the "destination." Since this is the only kind of register addressing used with this instruction, there is no associated T field. Thus, the registers in immediate instructions can be used only in the direct mode and not in the indirect mode.

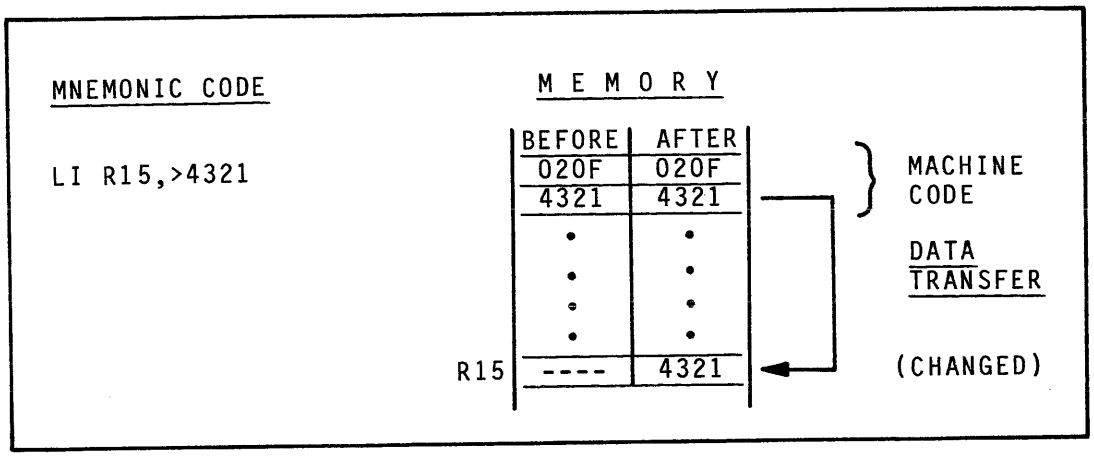


Figure 3-15. Immediate Addressing Illustration

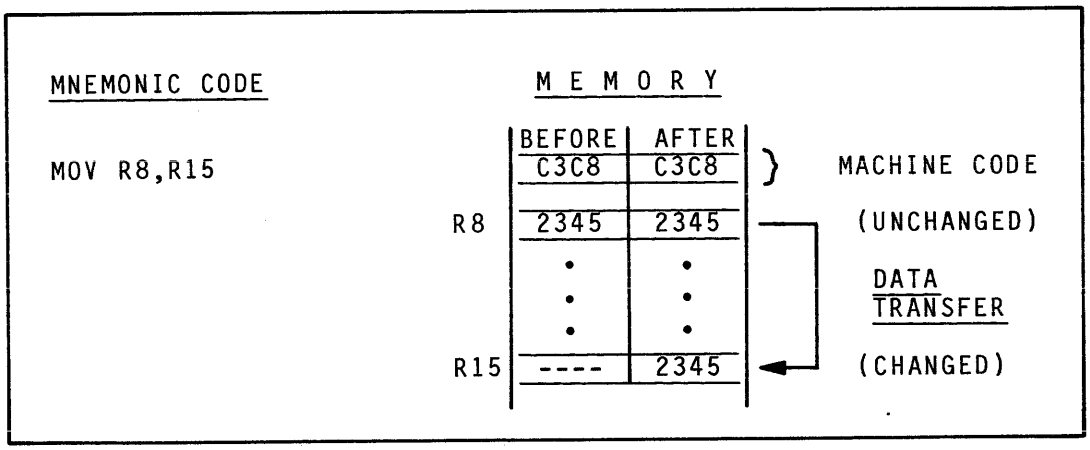


Figure 3-16. Register Direct Addressing Illustration

This limitation of workspace registers as the only type of destination will apply to the first five of these immediate instructions. Each of the remaining two of the seven will have as its destination a non-workspace register within the CPU.

As an example, a Load Immediate instruction is written LI R9,>400 with the register indicated first followed by a comma and the 16-bit constant. Since there is no T field involved, there is no need for any special marks as in the MOV instruction. Also, it is occasionally desired to load a register with a symbolic address such as ST, AF, CD, etc. In such case one merely specifies the symbol in the immediate operand field without the preceding @ sign since there is no T field to be defined. For example, LI R2,ST (load R2 with the address corresponding to the symbol ST).

With the LI instruction, one can initialize R0 to zero using LI R0,0 at the beginning of the FTN1 program rather than initializing

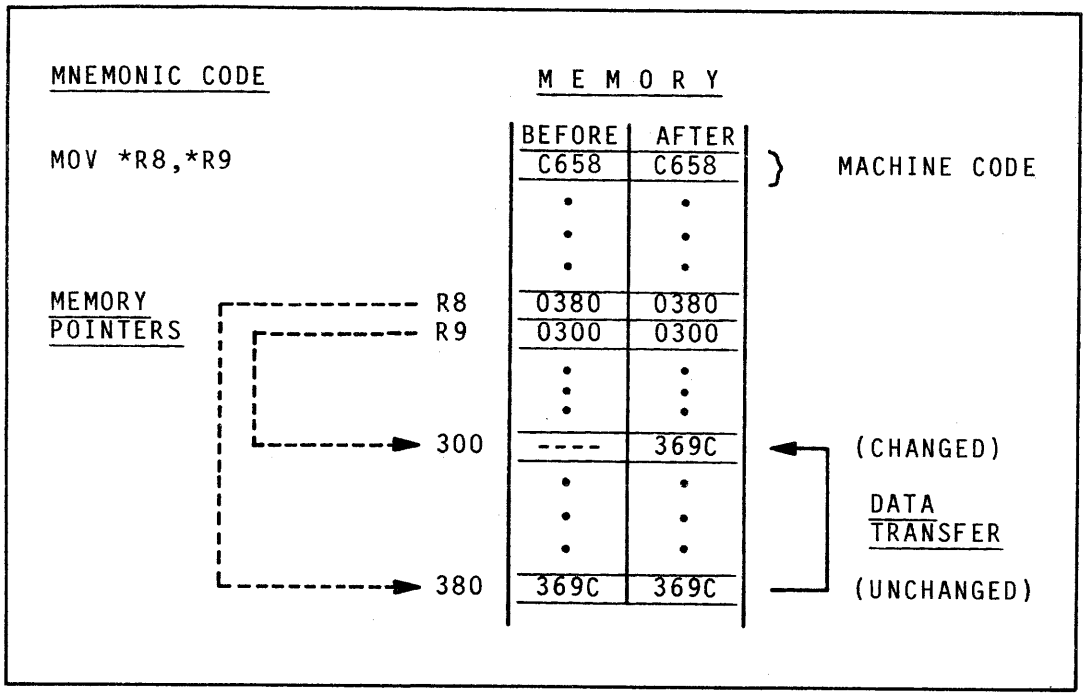


Figure 3-17. Register Indirect Addressing Illustration

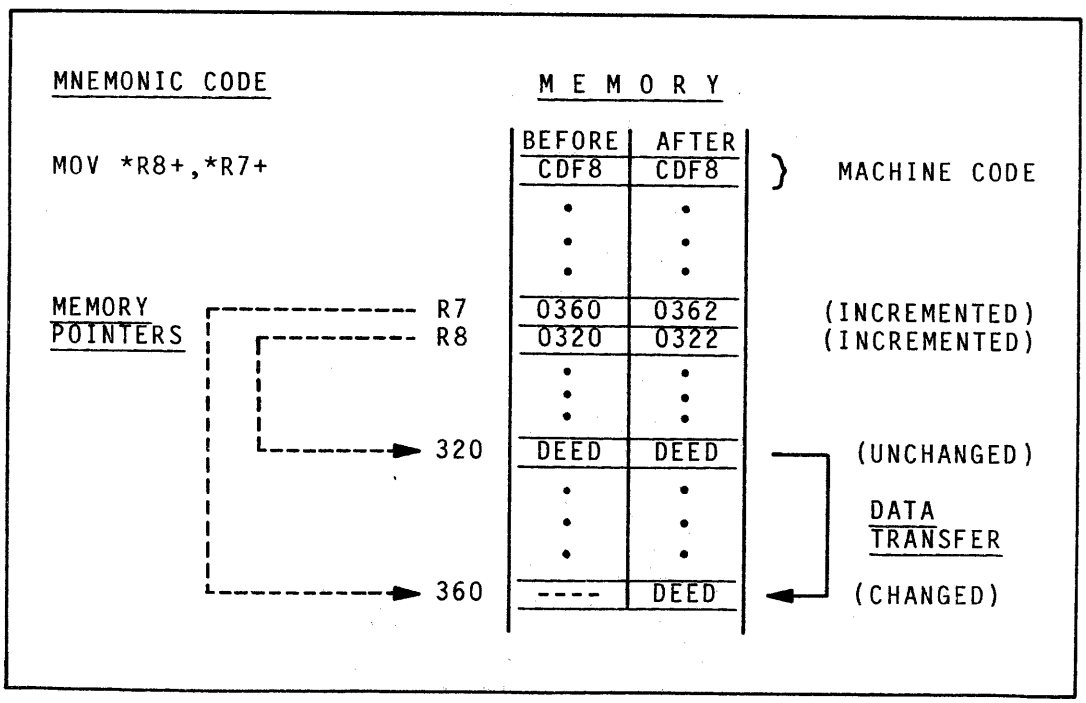


Figure 3-18. Register Indirect Autoincrement Addressing Illustration

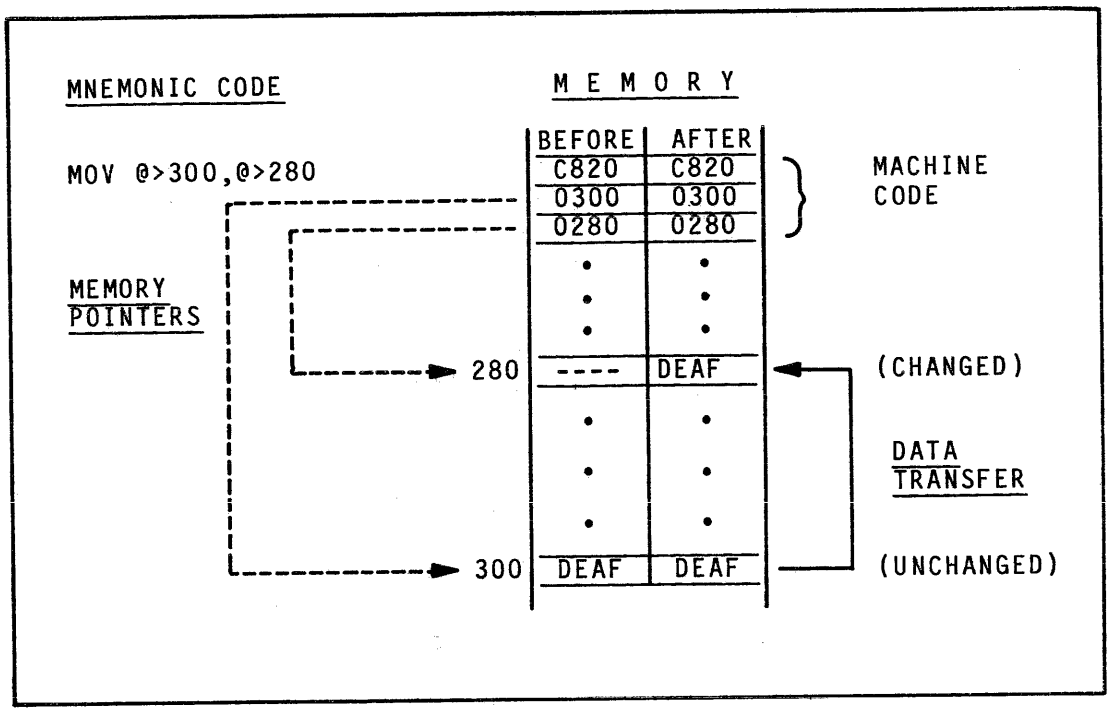


Figure 3-19. Memory Symbolic Addressing Illustration

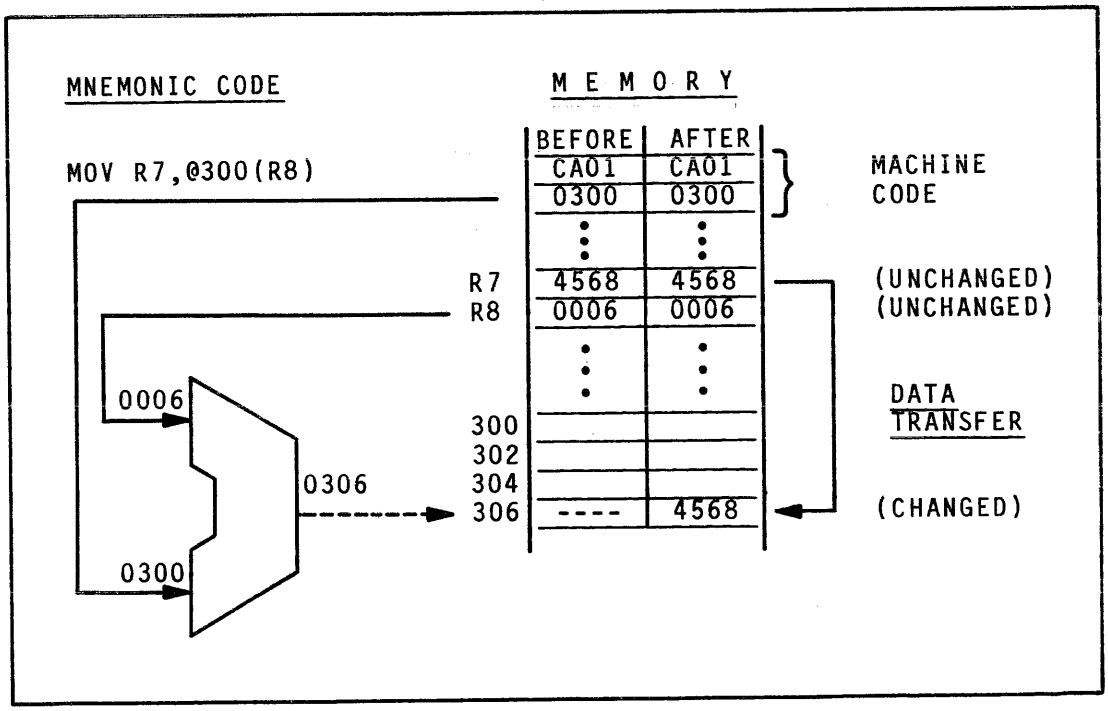


Figure 3-20. Memory Indexed Addressing Illustration

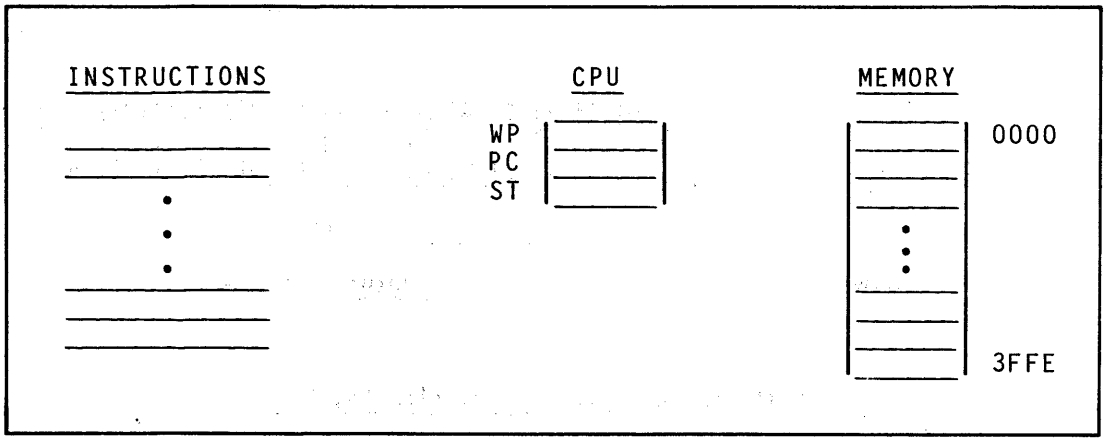


Figure 3-21. TM 990/189 Microcomputer (Software Perspective)

register 0 to zero with the UNIBUG monitor program. Since the machine opcode for this instruction translates into >0200 (the least-significant 0 for R0, with the next word being >0000 (for the constant 0).

Add Immediate Instruction

The Add Immediate (AI) instruction is the second of the seven immediate instructions with an immediate operand as the second instruction word. It is used to add signed 16-bit constants to workspace registers. There is no subtract immediate instruction. This is not a difficulty since a negative number can be used with the Add Immediate instruction. The Add Immediate instruction is used when the programmer desires to add or subtract a constant from an existing number in a register, for example, when stepping up or down through a table of numbers. See Instruction Summary 3-8.

In the FTN1 program example, a register was initialized with the value 1 and then this 1 was subtracted from the loop counter to provide a loop which repeated a definite number of times. The Add Immediate instruction can be used to accomplish the same function using -1 as the immediate operand. For example, AI R2,-1. The hexadecimal machine code is 0222 FFFF. This instruction could be substituted for the S R3,R2 instruction in Figure 3-8.

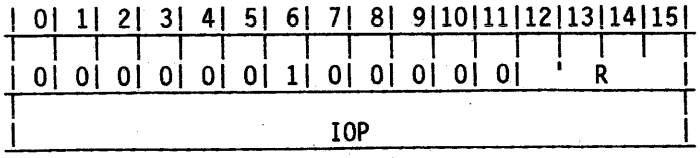
An Additional Addressing Illustration

From a software viewpoint, the TM 990/189 contains software instructions, memory, and a CPU with three hardware registers WP,PC, and ST as shown in Figure 3-21.

To more fully explain the operation of various instructions and addressing modes, another example is presented. See Figure 3-22

LOAD IMMEDIATE — LI

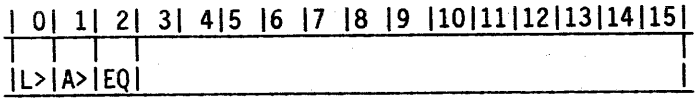
CODE: LI R,IOP



RESULT: IOP ⟶ (R)

Length: 2 words

STATUS REGISTER:



OPERATION:

Place the 16-bit immediate operand in the specified workspace register. The 16-bit value is compared to zero and the L>, A>, and EQ status bits are set accordingly.

NOTES:

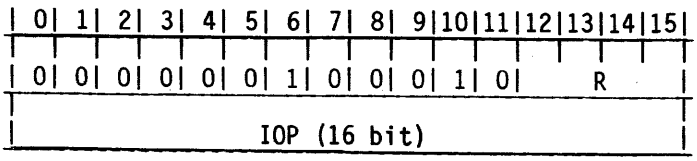
The LI instruction is used frequently to initialize constants in registers, and addresses or constants in various counters.

Examples:

Mnemonic Instruction	Comment	Machine Code
LI R5,>FEED	LOAD R5 WITH THE ADDRESS >FEED	0205 FEED
LI R7,10	LOAD R7 WITH THE CONSTANT TEN	0207 000A
LI R1,>0200	LOAD R5 WITH THE ADDRESS >0200	0201 0200
LI R2,>320	LOAD R2 WITH THE ADDRESS >320	0202 0320
MOV R2,@400	AND STORE IT IN MEMORY LOCATION >400	C802 0400

ADD IMMEDIATE — AI

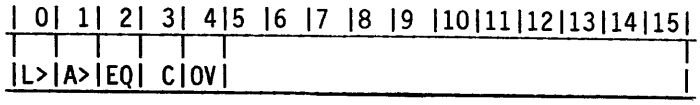
CODE: AI R,IOP 0220_{16}



RESULT: (R) + IOP ⟶ (R)

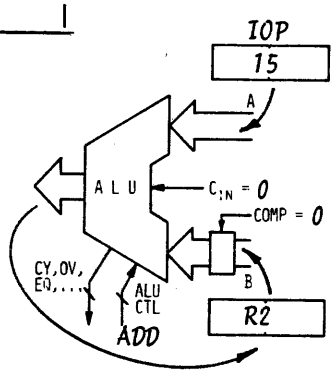
Length: 2 words

STATUS REGISTER:



OPERATION:

Add the immediate 16-bit operand to the specified workspace register. The sum is compared to zero and the L>, A>, and EQ status bits are set accordingly. The C and OV status bits are affected by the addition operation.



NOTES:

This instruction is used to add an immediate value to a workspace register.

Example:

Mnemonic Instruction	Comment	Machine Code
AI R2,>15	ADD >15 TO R2	0222 0015

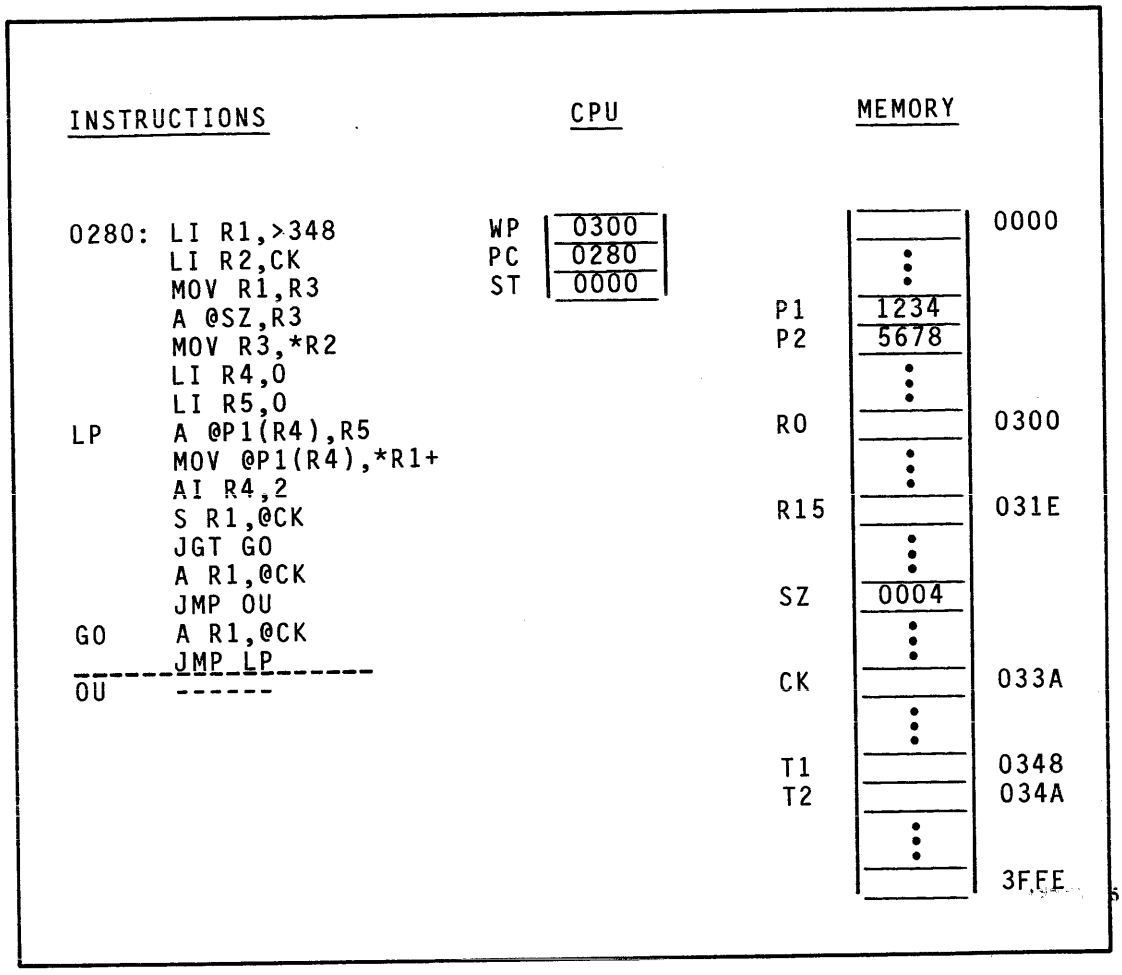


Figure 3-22. Addressing Review Program Example

for the program example and the given initial conditions of the CPU registers and memory.

After executing the instructions through the JMP OU, the memory locations contain the values indicated in Figure 3-23. The intermediate steps are left as an exercise for the reader.

Decrement/Increment Instructions

The instruction AI R2,-1 in the previous discussion introduces the entire field of incrementing and decrementing using small numbers. This instruction set supports the use of arithmetic with small numbers such as 1 and 2 with certain special instructions. Consider first what is required if the programmer wants to subtract 1 from a number in a memory location. As an example, he would first have to select a register such as R14 for the intermediate work, then write the

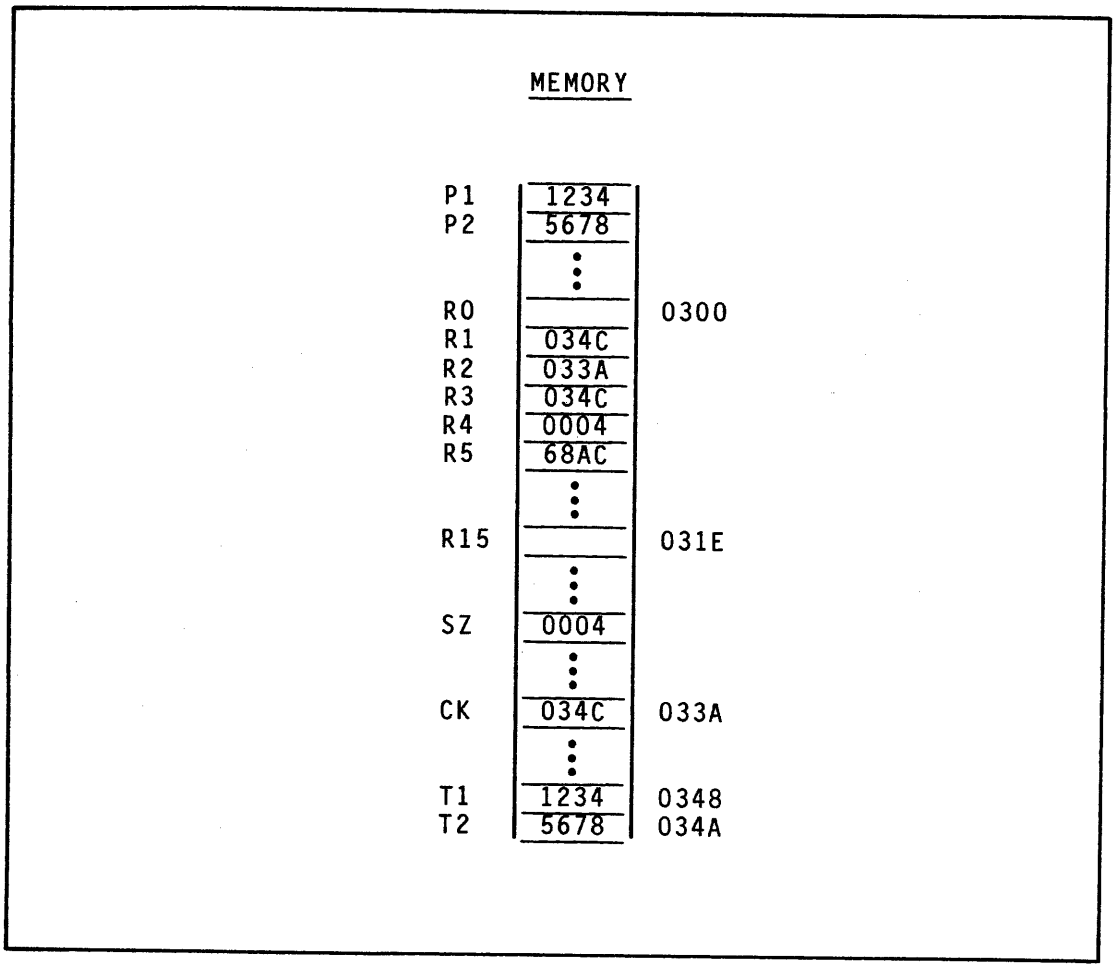


Figure 3-23. Memory Contents After Execution of Addressing Review Program Example

instruction LI R14,-1. Using memory location >2468 as an example, he would write A R14,@>2468. Note that this operation of subtracting one from memory location >2468 requires two instructions and four instruction words. It has already been shown how this can be accomplished with one instruction if the destination result is a register, with the AI instruction. Whether a general memory location or a workspace register, this operation can be accomplished with half as many words using a Decrement instruction.

This Decrement instrucion (DEC) is used often because it allows the programer to count down to zero to implement a repeated loop, with the completion test being accomplished without a compare instruction since the arithmetic operation of the Decrement instruction causes the status bits to be set by comparing the result to zero. Also, the TMS 9980A instruction set allows the programmer to perform an autoincrement (register indirect autoincrement) on the contents of a register, but it does not allow him to perform an autodecrement;

Table 3-4. Instruction Subset 1.

1. MOV	7. LI
2. A	8 AI
3. S	9. DEC
4. ABS	10. INC
5. JMP	11. DECT
6. JGT	12. INCT

therefore, a loop involving decrementing will generally involve a DEC instruction. See Instruction Summary 3-9.

For example, to perform an operation ten times:

```
        LI   R1,10
LOOP    _____        BEGINNING OF
        _____        REPEATED OPERATION
          .
          .
        _____        END OF OPERATION
        DEC R1
        JGT LOOP
```

The contrasting instruction is the Increment (by one) instruction (INC). This instruction works similarly to the Decrement instruction but adds one to the source operand. See details in Instruction Summary 3-10.

Instruction Summaries 3-11 and 3-12 detail the Decrement by Two (DECT) and Increment by Two (INCT) instructions.

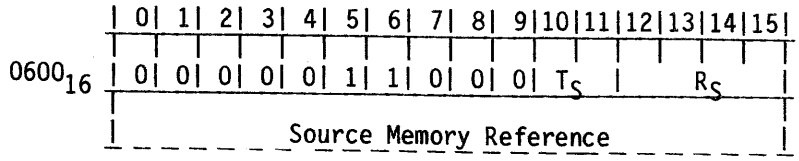
Instruction Review

At this point, the reader should be familiar with the operation of the instructions given in Table 3-4.

The first three (MOV, A, S) involve two operands (both general) and may be one, two, or three words in length.

DECREMENT (BY ONE) DEC

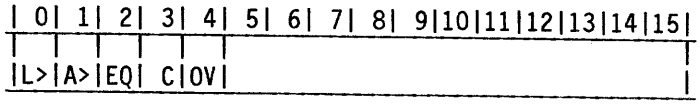
CODE: DEC S



Length: 1 or 2 words

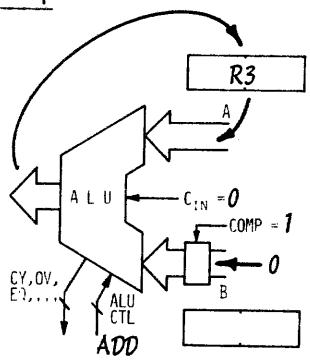
RESULT: (S) - 1 ⟶ (S)

STATUS REGISTER:



OPERATION:

Subtract 1 from the source operand and replace the source operand with the difference. The 16-bit result is compared with zero and the L>, A>, and EQ status bits are correspondingly affected. The subtraction operation also affects the C and OV bits.



NOTES:

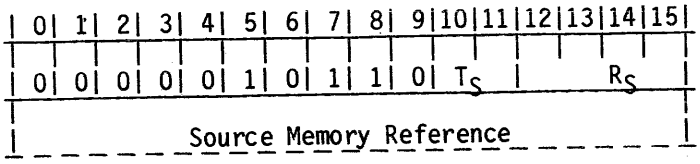
DEC is particularly useful in loop operations.

Example:

To perform an operation 200 times:

```
        LI R3,200
LP      _____________   (BEGINNING OF REPEATED
        _____________       OPERATION)
        _____________
        _____________
        _____________   END OF OPERATION
        DEC R3
        JGT LP
```

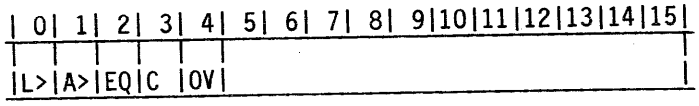
CODE: INC S 0580_{16}



RESULT: (S) + 1 ⟶ (S)

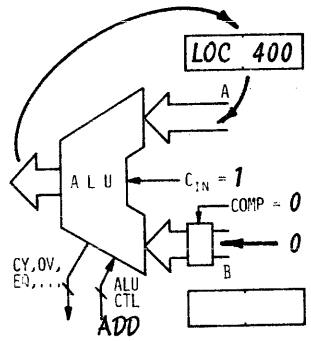
Length: 1 or 2 words

STATUS REGISTER:



OPERATION:

Add 1 to the source operand word and replace the source operand with the sum. The 16-bit result is compared with zero and the L>, A>, and EQ status bits are correspondingly affected. The addition operation also affects the C and OV status bits.



NOTES:

The INC instruction will often be used to provide a counter. In fact, it allows any memory location to be used directly as a counter since the incrementing of a memory location can be accomplished with one instruction.

This instruction is also useful in stepping through various tables (particularly byte-oriented tables) when the autoincrement feature is not conveniently available.

These instructions (INC and DEC) allow the operator to implement "up" and/or "down" counters in any memory location.

Examples:

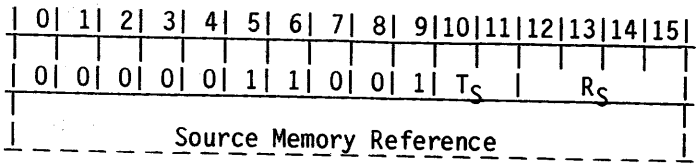
```
    LI R7,-20          INITIALIZE R7 FOR TIMER OPERATION
LP  INC R7             COUNT UP TO ZERO
    JNE LP             IF (R7) ≠ 0, JUMP TO LP

    INC @>400          INCREMENT MEMORY LOCATION >400
```

The hexadecimal machine code for INC @>400 is 05A0 0400.

DECREMENT BY TWO

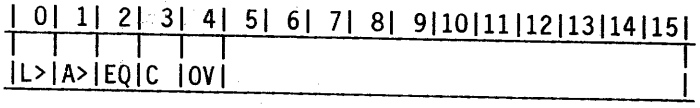
DECT

CODE: DECT S



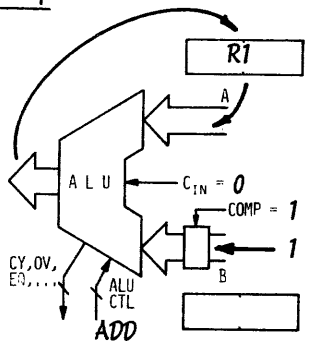
RESULT: (S) - 2 ⟶ (S)

Length: 1 or 2 words



OPERATION:

Subtract 2 from the source operand and replace the source operand with the difference. The result is compared to zero and the L>, A>, and EQ status bits are set accordingly. The C and OV status bits are also affected.



NOTES:

This instruction is useful in stepping through word-oriented tables. This instruction is particularly useful if decrementing is required relative to a word table since there is no auto-decrement feature.

Example:

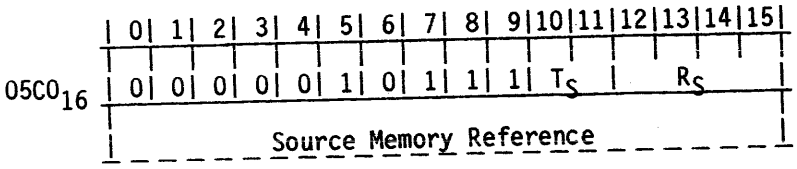
```
   LI R1,20                      LOAD R1 WITH DECIMAL 20
NX MOV @>500(R1),@>600(R1)       MOVE FROM ONE TABLE TO ANOTHER
   DECT R1                       DECREMENT INDEX REGISTER R1
   JGT NX                        JUMP BACK TO NX IF R1>0
```

The machine code for DECT R1 is: 0641.

INCREMENT BY TWO

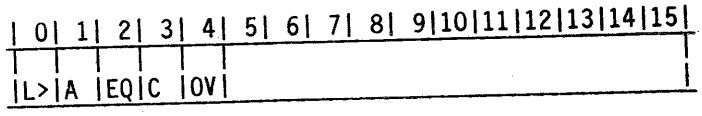
INCT

CODE: INCT S



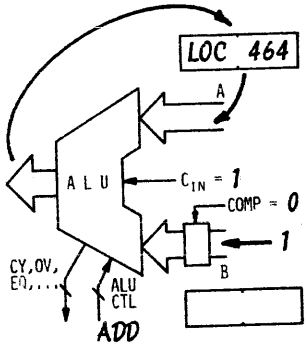
RESULT: (S) + 2 ⟶ (S)

Length: 1 or 2 words



OPERATION:

Add 2 to the source operand and replace the source operand with the sum. The 16-bit result is compared to zero and the L>, A>, and EQ status bits are correspondingly affected. The addition operation also affects the C and OV status bits.



NOTES:

This instruction is useful in stepping through word-oriented tables.

Example:

INCT @>450(R7) INCREMENT CONTENTS OF INDEXED ADDRESSED LOCATION BY 2

The hexadecimal machine code for INCT @>450(R7) is 05E7 0450.

For ALU operation above, assume (R7) = >14.

The next one (ABS) involves one operand (general) and may be one or two words in length.

The next two (JMP and JGT) are only one word in length and use PC relative or jump addressing.

The next two (LI and AI) use immediate operands and are always two words in length. These are the first two of seven immediate operand instructions, each of which is two words in length.

The last four (DEC, INC, DECT, and INCT) are special "one operand" instructions, each of which are one or two words in length.

3.12 PROGRAM PRODUCTION PROCESS

Now that the program production process has been illustrated with a specific example, it is important to discuss the various steps of the process explicitly, as outlined in Figure 3-24.

The steps outlined in Figure 3-24 are briefly discussed in the following paragraphs.

(1) The first step in producing a program is to have a general programming idea. This may emerge from a project on which one is working or it may be assigned by someone. It is simply a general statement of what is to be accomplished.

(2) The second step is to specify or objectify this idea with a specific program specification. This specification indicates in objective detail what are to be the inputs and outputs and what is to be accomplished by the program. Although programmers write program specifications, on high-level, large, or sophisticated programs, this is the task for a systems engineer or a system analyst. Failure to provide such a specification at an early stage may invite problems. On very short stand-alone programs (about ten or fewer instructions) the difficulties and changes are usually not significant. The specification step also includes documenting any algorithms or methods of attack as well as a plan to test the resulting program.

(3) The third step in producing a program is to construct a program chart--a logical and sequential presentation of the program structure. There are a number of different types of charting techniques. The one selected generally depends on the type of program at hand. In any case, this chart representation functionally spells out the program operation showing the sequence and the logic. A primary example of this is a flowchart. This is an important step, for it forces one to think out the details of the problem at hand on a high level and to document the logic and the sequence. It is helpful (sometimes crucial) to the phase of debugging and error correction as well as for later understanding of program operation. This again is often produced by programmers, but for high-level, large, or sophisticated programs, this should be the task for a systems engineer or a systems analyst.

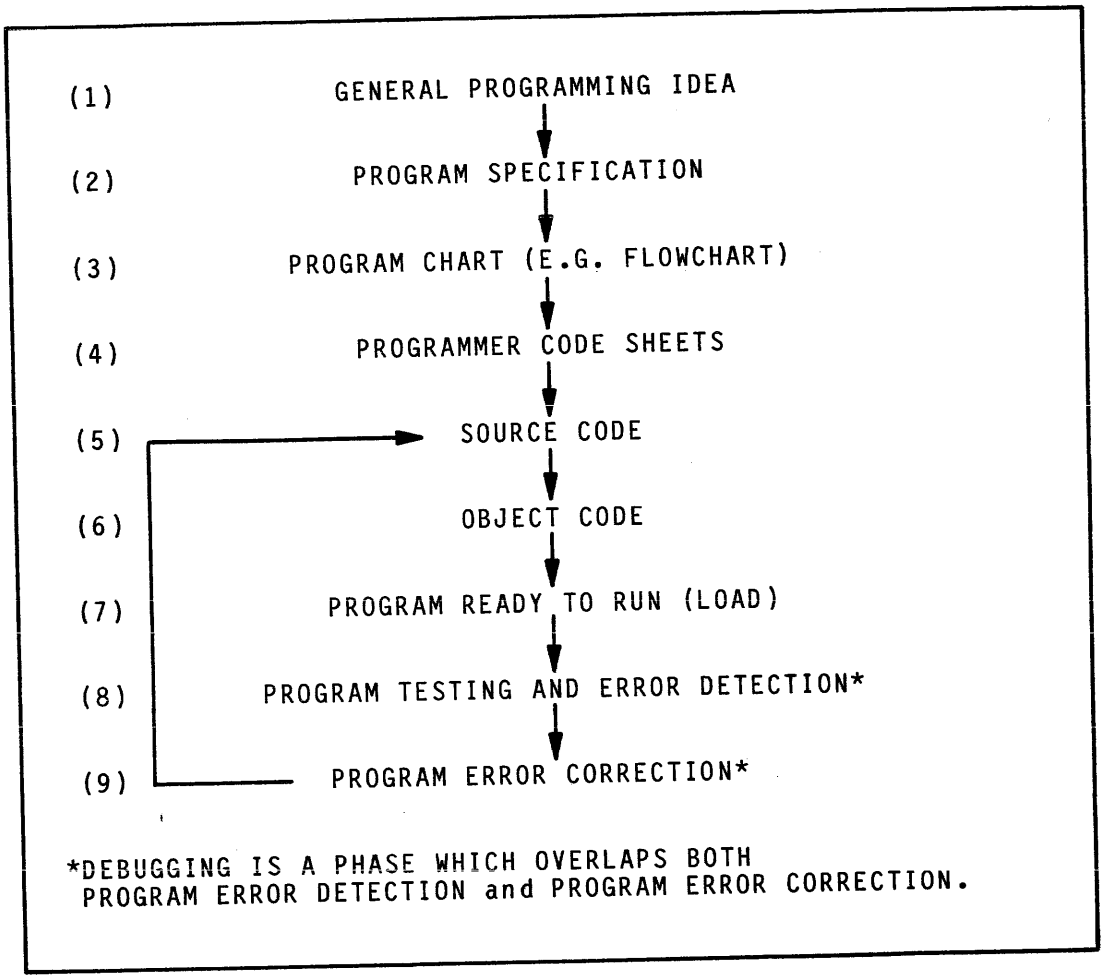


Figure 3-24. Program Production Process

(4) The fourth step is the actual programming, in which the programmer "translates" the specification and the program chart into machine related language (whether high-level, assembly, or machine language). This generally results in programmer code sheets, which may or may not be readable by someone else. This task is called programming (or sometimes called coding). If the specification and chart are clear, logical, and consistent, this step is fairly straightforward but often particular "knots" will exist in this process.

(5) The fifth step is a clerical one of making the resulting program readable by the machine. When referring to assembly language or high-level language, this machine-readable form is called source code. Basically, the process of going from programmer code sheets to source code is clerical and is called source entry. In the case of the TM 990/189 (University Board), it simply entails entering the mnemonic code via the keyboard from the code sheets.

(6) When the source code is composed of assembly language instructions, the sixth step is the work of the assembler program to convert from source code to object code. The source input is in mnemonic code and the output is in machine code (sometimes specifically formulated for easy loading from machine-readable medium). The output of the assembler on the TM 990/189 is machine code which is stored in RAM memory ready for execution.

(7) The seventh step is to load the object into the computer via the loader program to provide a program in memory ready to run. In many applications, the program is loaded from some magnetic medium. For the TM 990/189, it is done by the assembler directly. Or, if the program has been "dumped" to cassette then it is done by "loading" from cassette. The format on the tape in that case is, in Texas Instruments' terminology, "standard object format."

(8) The eighth step is to run the program to checkout its operation and detect program errors (bugs). This is best done according to the specified test plan. Generally, a monitor program, such as UNIBUG, is available and allows the operator to set and inspect registers, etc. This phase is called program testing. The operator may also refer to it as a phase of debugging if he assumes that there are program errors. Such an assumption will generally be well-founded.

(9) The ninth step is to correct program errors. This is the second phase of debugging. It is one thing to detect a bug and another to identify and correct it. It should be noted that one may well produce new bugs while correcting a previous bug.

This process of correcting program errors is best done to the permanent record of the source program. But in general it could be done to:

- The resulting machine code (i.e., patching)
- The machine-readable object code

- The source code
- The programmer's code sheets.

Then, if errors are still present, the cycle is repeated.

The programmer continues around this loop of detecting, correcting, and reassembling until no more errors are detected, which may mean the program is "error free;" or, it may mean the bugs are hiding in an uninvestigated area.

3.13 SUMMARY

This chapter has endeavored to provide the reader with a clear understanding of

- The three types of register addressing
- The two types of memory addressing
- Immediate addressing
- Jump addressing.

In addition, the instructions in subset 1A and 1B have been described and illustrated, using both mnemonic and machine code formats. The nine steps in the program production process were introduced to provide the reader with a methodology that can lead to success in in programming.

3.14 EXERCISES

For exercises 1-10 assume for each exercise the following contents for the WP and for each of the memory locations indicated.

(WP)	= >0300		
(0300)	= >0406	(0400)	= >2468
(0302)	= >CACE	(0402)	= >0101
(0304)	= >8234	(0404)	= >BABE
(0306)	= >0405	(0406)	= >0000
(0308)	= >0406	(0408)	= >FFFF
(030A)	= >0408	(040A)	= >8000
(030C)	= >030C	(040C)	= >CBA9

Execute each of the following stand-alone instructions and determine the result (location and contents) and the condition of the status register. Indicate the related machine code.

		Result	Machine Code
Example:	MOV @>404,R2	(R2)=>BABE L>=1, A>=0, EQ=0	C0A0 0404
1.	A R2,R0		
2.	S *R3,R1		
3.	MOV *R4,*R5		
4.	ABS @>040A		
5.	MOV *R0,@>100(R6)		
6.	A @>040C,@>0303		
7.	LI R5,>1234		
8.	AI R3,>0101		
9.	INCT @>0300		
10.	DEC @>FFFB(R3)		

11. Write the mnemonic code equivalent of MOV R2,*R3 using memory indexed addressing.

12. Determine the machine code for the following jump instructions.

(a) JMP ST (b) JGT $+12 (c) JMP $-6

_____
_____
_____
ST _____
_____

13. Will the following program reach the instruction located at OU? If it does, what will be the values in R7, R8 and R9? How many times is the JMP LP instruction executed?

```
   LI R9,0
   LI R7,-4
   LI R8,9
LP DECT R8
   A R8,R9
   INC R7
   JGT OU
   JMP LP
OU JMP OU
```

14. Consider the following program segment given:

```
   (R3) = >220      (>220) = >ABCD
   (R4) = >280      (>280) = >9876
   (PC) = AQ
AQ A *R3+,*R4+
   JGT NX
FS JMP AD
NX
```

(a) What is the PC value after the execution of the first two instructions?
(b) What is the contents of the status register?
(c) What memory locations have changed and what are their contents?

3.15 LAB EXPERIMENTS

1. Load and execute the FTN1 program listed in Figure 3-8 using the values for N1 and N2 given in Table 3-5 and indicate results. See Chapter 2 for UNIBUG commands to do this.

Table 3-5. FTN1 Program Input Values

Inputs N1	5	2	10	35	400_{10}	$30{,}000_{10}$			
N2	3	5	-2	-30	100_{10}	2.000_{10}			
Hexa-decimal Result									
Decimal									
Comments									

2. Encode, enter, and execute the FTN1 program with the change suggested on pages 141 & 144 for the LI and A instructions.

3. Enter and execute the sample programs illustrating the A instruction and the S instruction on page 117. (Note initial values.)

4. Write, encode, enter, and execute a program to step through a table of words starting at location >300 and continuing to >3FF. If a nonpositive word is found, stop with an appropriate pointer in register 1. If the table is completed without finding a nonpositive number, the pointer should contain >400.

5. Write a timing program to turn on the "idle" light (see picture of TM 990/189 board, Figure 1-33) after a specified number of seconds. End the program with >0340 to turn on this light. Use the following basic timing loop.

```
      LI RX,>220F     LOAD REGISTER X WITH >220F
LP    DEC RX          DECREMENT REGISTER X
      JGT LP          IF (RX)>0, JUMP TO LP
```

This timing loop requires approximately 100 milliseconds to complete.

6. Write a program for N greater than zero to calculate the N-factorial function, e.g., 5! (5-factorial) = 5 * 4 * 3 * 2 * 1 = 120. Provide the specified N in R1 (using a LI R1,N) or initialize via UNIBUG. Provide the result in R0.

CHAPTER 4

ASSEMBLY LANGUAGE

4.1 INTRODUCTION

This chapter introduces the user of the University Board to three items of special interest: the context and use of assembly language, the functions and features of the Symbolic Assembler, and Instruction Subset 2. These three items and correlated exercises using the assembler program are covered in this chapter.

Categories of Software (An Overview)

As background, it is important for the reader to know that there are different categories of software, and that each has different challenges and requires different approaches. Generally, software programming can be categorized as systems, utility and support, and applications.

Systems Programming. Consider first the category called systems programming. This generally refers to the development of operating systems. An operating system is a software package which will take control of a computer and will interface between the applications programs and the computer system to facilitate:

- Accessing a disk
- Controlling a printer
- Configuring memory
- Loading programs from the disk
- Controlling a CRT
- Coordinating the sharing of computer time between pro-programs
- Taking care of error modes, failure modes, power failures, power restarts, etc.

A special category of systems programs interface a computer with one or more peripheral devices--a printer, a CRT, a keyboard, or all of these. The principal goal for operating systems, and systems programming in general, is to control and coordinate computer system resources to include computer-system facilities, computer time, memory space, and the peripheral devices.

Utility and Support Programs. Utility and support programs, in contrast, are those programs which facilitate development of their own applications or which facilitate the use of a production computer system. Several examples of these are the software tools commonly used to aid in developing programs such as (a) an assembler which will convert mnemonic codes into machine codes such as the University Board Symbolic Assembler, (b) a text editor which assists users in editing or changing programs during development, such as the TM 990/302 Text Editor (the TM 990/302 is discussed in Chapter 10), (c) a compiler which helps programmers convert from high-level language to machine language such as the TI 990 DX10 FORTRAN Compiler, (d) a linker which helps a programmer link two separately developed programs together into one working module, and (e) an interpreter which allows the use of high-level languages such as the Texas Instruments POWER BASIC (mentioned in Chapter 10). An additional example of a utility program is one which provides an output-print format, with the utility program handling the details involved or a disk utility program to reorganize files on a disk. Furthermore, there may be special mathematical packages for the applications programmer to use.

Applications Programs. An applications program is one written for a specific application, such as for a word-processing system, or a motor-control system, or a mathematical calculation. These are the uses for which most programs are written. Applications programs can be subdivided further into three basic categories: scientific, commercial, and real-time processing and control. These three subcategories require different types of programming discipline.

Applications programs in the scientific category generally require significant understanding of mathematics by the programmer in their implementation into software programs. This area is generally represented by work in high-level languages such as FORTRAN. Such programs in commercial usage generally involve accounting and bookkeeping tasks, extensive manipulation of files, and detailed formatting of outputs. In real-time processing and control generally involved are high-speed analysis and reaction to random inputs, coordination of correlated functions being performed simultaneously, and detailed control of external devices.

Overview of Principal Levels of Computer Language

As is described in Chapter 3, there are three principal levels of computer language: machine language, assembly language, and high-level language. The most elementary level, although not the simplest, is machine language. Some programmers have written programs in machine language with as many as 200 instructions. Shorter programs of less than 100 instructions written in machine language are common. Lengthy programs in this language tend to be particularly troublesome to write, debug, and modify. But, it is essential for a programmer to know machine language if he is to modify (patch) a program. Patching is expected to be necessary in the final stages of de-

bugging. If an operator does not have an assembly program to convert from assembly or mnemonic code to machine language then he must program in machine language (as is done in the previous chapter's exercises). He may, as in Chapter 3, code in mnemonic code then hand assemble. This provides an important level of documentation one step above machine language code.

The next level, which is the focus of this chapter, is assembly language. It is defined as a one-for-one correspondence between mnemonic code and machine code. This correspondence is illustrated in Figure 4-1.

There are two facts which must be emphasized:

- Assembly language is the most commonly used level of computer language for microprocessor and real-time applications due to the balance between readability, execution speed, and efficient use of memory space.
- The assembly language programmer must understand the computer's architecture to implement a program.

The principal advantage of assembly language over machine language is its readability.

The third level of computer language is called high-level language. It is also referred to as problem-oriented or programmer-oriented language. And, as mentioned in Chapter 3, each source statement usually will cause generation of five to ten machine codes by the compiler, rather than the one-for-one correspondence of assembly language. The reader may have heard of a number of high-level languages. Some of the more common ones are:

- FORTRAN
- COBOL
- PL/1
- ALGOL
- BASIC
- Pascal
- PL/M
- ATLAS.

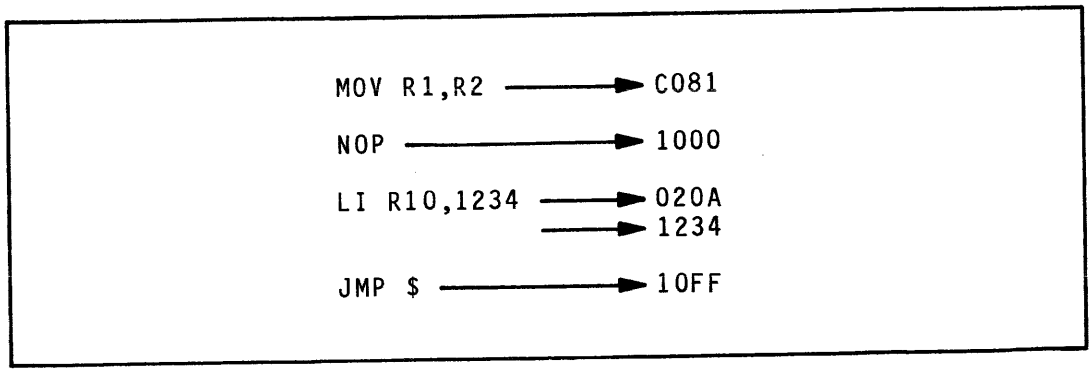


Figure 4-1. Mnemonic Code and Corresponding Machine Code

The primary advantage of the use of high-level language for implementing applications programs is the saving in programming time.

High-level programs tend to be less CPU dependent, that is, the programs may be transportable from one computer to another without total rewriting and, possibly, with little or no change. The primary disadvantage of high-level language is that the resulting program is generally not as efficient in terms of memory space and computer time when compared to a similar program implemented in assembly language.

To further illustrate this difference between these two levels of computer language, consider Figure 4-2. As was stated in Chapter 3, one typically starts the development of a program with a programming idea.

Figure 4-2 shows that there are many ways to express a program, starting with a programming idea, progressing to a flowchart, then to assembly language, and finally to machine language. It also illustrates how the same program is expressed in higher level languages. Note that the degree of readability to the casual eye becomes less as we proceed from the programming idea to the flowchart to the assembly language and then to machine language. In fact, most would consider the machine language unreadable. In contrast, the higher level language implementation can be seen to correspond to this programming idea, even if one does not know these languages. For the interested reader, Example A in Figure 4-2 is FORTRAN, Example B is a version of BASIC, and Example C is PL/1.

On the other hand, from the machine's point of view, the machine language is much more "readable" than assembly language or higher level language, and the flowchart and the programming idea are totally unreadable. Thus, the degree of readability depends upon one's perspective. So there is a need for a middle ground between the human perspective and the machine perspective. This usually is met by use of assembly language or higher level language.

4.2 OVERVIEW OF ASSEMBLER FUNCTIONS

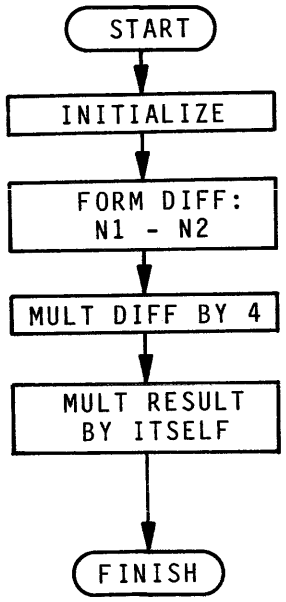
Assembly language is the input format to an assembler program (usually referred to as the "assembler"). There are certain basic functions which all assemblers perform, including the symbolic assembler provided with the University Board. These basic functions are

- Translation
- Address bookkeeping
- Symbolic constant definition (assembly-time)
- Error indications
- Output control.

TYPES OF PROGRAMMING EXPRESSION

PROGRAMMING IDEA: CALCULATE $[4(N1 - N2)]^2$

FLOWCHART



ASSEMBLY LANGUAGE

```
START LWPI >0300
      CLR  R0
      S    R2,R1

      SLA  R1,2

      ABS  R1
      MOV  R1,R2
LOOP  A    R1,R0
      DEC  R2
      JGT  LOOP

HALT  IDLE
      END  START
```

MACHINE LANGUAGE

```
02E0
0300
04C0
6042

0A21

0741
C081
A001
0602
15FD

0340
```

HIGHER LEVEL LANGUAGE

EXAMPLE A

```
NFTN = (4*(N1 - N2))**2
GO TO ENDJOB
END
```

EXAMPLE B

```
NFTN =  N1 - N2  *4
NFTN = NFTN * NFTN
END
```

EXAMPLE C

```
START:  PROCEDURE;
        NFTN = (4*(N1 - N2))**2;
        END START;
```

Figure 4-2. Types of Programming Expression

Translation

The primary function of an assembler is translation. It translates mnemonic opcodes into machine opcodes. For example, the University Board Symbolic Assembler translates the mnemonic opcode:

```
AI R7,15
```

into hexadecimal machine code as:

```
0227
000F
```

Most would agree that the first expression is easier to read and easier to remember, as well as easier to compose as part of a program, than the second.

Address Bookkeeping

Another function of the assembler program is to provide support in the area of address bookkeeping. For example, if one enters (starting at >410):

```
Mnemonic Code

   JGT KP
   _____
   _____
KP _____
```

Then the resulting machine code will be (starting at >410):

```
Location  Code

  410     1502
          _____
          _____
  416     _____
```

NOTE: The 02 in the machine code 1502_{16} is the displacement to reach from the instruction at >410 to the destination at >416 (the location corresponding to the label KP as explained in Chapter 3).

As the above example illustrates, one can refer to memory locations by means of labels and allow the assembler to do the address bookkeeping. In the case of jump instructions, this saves one from having to count the words or bytes, or even having to understand how that counting is done, because the assembler will do the counting for the programmer. This is particularly important when one needs to change a program by inserting instructions. If an instruction needs to be inserted between a jump instruction and its destination, then the displacement field in the machine code of the jump instruction

would need to be changed. If this jump reference is done symbolically, as illustrated above, then the assembler will automatically do the necessary address bookkeeping.

Symbolic Constant Definition (Assembly-Time)

Generally an assembler supports symbolic constants to be used at the time it is making its conversion from mnemonic code to machine code (i.e., at assembly time). The use of symbolic constants is important because these make it easier to effect program changes. The time of this operation is called assembly time. The use of symbolic constants is illustrated in the following lines of code.

```
Mnemonic Code          Comment

FG EQU 15              FG SYMBOL=15
    AI R7,FG           ADD 15 TO R7
CD EQU >A55A           CD SYMBOL=>A55A
    LI R9,CD           LOAD R9 WITH >A55A
```

The first line of mnemonic code indicates that the symbol FG is to be equated (at assembly time) to the numerical constant 15_{10}. This, in combination with the second line, AI R7,FG, will assemble exactly like its equivalent code: AI R7,15. This occurs because the assembler will substitute a 15_{10} for the FG at assembly time based on this first line of code. This third line of code indicates that the symbol CD is equated with the literal constant: >A55A. This in combination with the fourth line of mnemonic code is equivalent to CI R9,>A55A since the assembler will substitute a >A55A for the symbol CD.

With symbolic constants, if one desires to change a symbolic reference, then one needs to change its value in only one place, and the assembler can take care of all of the other changes automatically. For example, if one desires to change the "code" >A55A (this symbol being abbreviated as CD), and this code is used in 100 different places, then one merely has to change the one EQU statement to accomplish this objective. On the other hand, if this constant were to be changed in machine code directly, then it would have to be found and changed in all of the 100 different places where it occurs. Thus, the use of symbolic constants at assembly time provides centralized control of constants.

Error Indications

Another function of an assembler program is to provide error messages indicating various syntax errors. A syntax error occurs when the mnemonic input (source code) does not follow prescribed rules. When a syntax error occurs, e.g., MOX is entered for MOV, the assembler produces an error message to indicate its occurrence. It is emphasized that the assembler program indicates only syntax errors; the detection of logic errors is left for the programmer.

Output Control

Output control is still another function which assemblers perform. The assembler input (as was indicated in Chapter 3) is called the source, and its output is called the object (or object code). An assembler program, in fact, generally has two outputs, an object and a listing. The object is machine code formatted to be read by the computer. The listing is an output of the source code input and machine code, formatted to be read by people.

These five general assembler functions discussed provide the foundation for an in-depth description and discussion of the University Board Symbolic Assembler.

4.3 UNIVERSITY BOARD SYMBOLIC ASSEMBLER

The symbolic assembler provided with the University Board translates the mnemonic codes into machine codes as these are entered via the keyboard and places the machine codes directly into memory. This assembler supports labels (symbolic address references) and other symbols consisting of one or two characters. The first character of a label or other symbol must be alphabetic (A-Z), and the second character must be alphanumeric (A-Z or 0-9). A label is a special kind of symbol which corresponds to a specific memory address. This assembler produces the resulting output in "one pass"* at the input source code. So any reference to a symbol which is used or defined later in the program is called an undefined forward reference. This assembler supports forward references in most cases. Forward references of the word type** and involving jump addressing are supported by this symbolic assembler.

Examples of these are:

```
MOV @BD,@EF
```

where BD and EF are labels for instructions or directives not yet entered (forward references), and

```
JMP RW
```

where RW is a label for an instruction not yet entered (forward reference). In each case, a reference to the undefined symbol is retained by the symbolic assembler in a symbol table.

References to previously defined symbols (backward references) are also supported. However, if a jump instruction attempts to "reach" too far, an error will result.

---

* Most assemblers perform this process using two or more passes at analyzing the input source code.

** Forward references of the word type refer to memory addressing and immediate operands of two-word instructions.

Execution of the Symbolic Assembler

To execute the assembler enter an A (for assemble) to UNIBUG followed by the desired starting point in memory (default is 0).

```
?A[starting memory address]Ret
```

The typical program resides between >0200 and >03FF. Location 0 should not be used. It is in the area for interrupt vectors.

To illustrate, for an assembly starting at location >0200. Enter A 0200 after the monitor prompt character:

```
?A0200Ret
```

After entering a return (Ret), the assembler responds by displaying the requested starting address and awaits entry of the first instruction. Now the user is ready to enter the source statements, each being concluded with a return (Ret).

When the assembler is entered via the "A" command, it is initialized so as to not refer to any previously entered program label or a symbol. If the assembler is exited normally, (via the END directive explained later in this chapter), it can be re-entered so as to continue assembly of the same program (maintaining previously referenced symbols) by entering the assembler using the B command followed by the desired restart point:

```
?B[restart memory address]Ret
```

This allows the programmer to exit to the monitor, to perform other functions, and then to resume assembly of the same program.

Functions of the Symbolic Assembler

With this symbolic assembler, the source input is expected to be from the keyboard and the output goes directly to the memory locations specified by the program. Having the output of the assembler stored directly into the memory is particularly useful because the program will be ready to run immediately upon completion of program entry. It also means that the specified destination RAM space must be available at the time of program assembly.

Upon entry of the assembler, typically, the user will want to set the location counter. The initial setting of the location counter can be done at the time of entry to the assembler from UNIBUG or after entering the assembler. The location counter can be set to a specific value in the assembler by use of the Absolute Origin (AORG) directive. For example, if the location counter*

*The location counter contains the location (such as >280) of the first word of the instruction currently being entered. It always appears and is always an even number (with System ROM REV. B).

is to be set at >0280, then the operator merely enters AORG >0280, and the assembler causes the location counter to be set to >280. This directive can be used at any time during the process of entering source code, but generally it is only used as the first line of source code.

Entry Fields

Turning to the entry of a normal line of assembly code, a user's entry fields consist of the following four fields: first, the optional label field, then the opcode field, an operand field, and finally, an optional comment field.

```
[label]Ø...<opcode>Ø[operand(s)][Ø...comment]Ret

[...]optional field
<...>required field
Ø one space only
Ø...one or more spaces
```

Focusing on each of these individually, the label field consists of either a space (when there is no label) or one or two characters to indicate a label. The first label character must be alphabetic, but the second character, if there is one, can be alphanumeric. The label field, if any, is followed by one or more spaces.

After the label field and a space or spaces, the opcode field always begins with an alphabetic character such as an A for add, or S for subtract. This field is also used for directives such as AORG. It consists of one to four alphabetic characters followed by one (and only one) space.

Next is the operand field which is used to call out one or two operands, as required by the particular instruction. An example of an operand field is R1,R2. Note that the operand field has no spaces within it, and multiple operands are separated by commas. The operand field is concluded by a space or an Ret. Several instructions have no operand, in which case the operand field is simply not used.

The comment field, if used, may include any character, and will continue until the receipt of a carriage return. The Ret at the end of either the operand field or the comment field marks the end of the line. The comment field is useful only if an optional hardcopy terminal is used, connected via the user-option EIA port. Comments are not stored in memory (only assembled object).

Examples:

```
XY MOV R1,@SV    SAVE R1 in SV Ret
Z  S R1,R2       TAKE DIFF Ret
```

There are certain predefined symbols which should be noted. When an operand includes a dollar sign ($) as an initial character, it is considered to refer to the contents of the location counter. For example, at location >370, JMP$+8 is equivalent to JMP >378. In calling out register operands, the programmer can use the symbol R followed by a decimal number. This includes R0 through R15. For example, MOV R1,R15 is the same as MOV 1,15.

Note that the default number system with this assembler is decimal; hexadecimal numbers are indicated with a > prefix.

4.4 DIRECTIVES

There are various directives which may be used to indicate operations to be performed by the symbolic assembler such as:

- Origin control
- Line cancellation
- Assembler exit
- Block declaration
- Word initialization
- Symbolic constant definition (assembly-time)
- String constant initialization

Keep in mind that a directive is an instruction for the assembler to perform a certain operation at assembly time. It does not correspond to any machine opcode and does not generate any object code except word initialization or string constant generation. Each of the directives is discussed briefly below. The directive is entered in the opcode field preceded, optionally, by a label.

Origin Control (AORG)

The location counter can be set to a specific value during assembler operation by use of the Absolute Origin (AORG) directive:

```
AORG <origin>
```

This directive can be used at any time during the process of entering source code. It will generally be used as the first program entry for the starting location of the assembled code. For example, AORG >300 will result in the next instruction assembled to be at memory address >300.

Line Cancellation (CANC Character)

Occasionally, the user will make an error and desire to cancel the line and start over again. This can be accomplished by entering the CANC character (shift key followed by "X" key) at any point during the line. Entry of the CANC character causes the assembler to start the line over again with the location counter as it was when the line began.

Assembler Exit (END)

The assembler can be exited at any time by entering the directive END in the opcode field. This directive causes exit from the assembler to UNIBUG. It has an optional operand field which can be used to indicate the starting point of the program. This optional operand can be a numerical address or a defined label symbolizing the starting address. Entry of an operand, at this point, can allow the user to insert where the program is to start and, thus, to set up the program counter in UNIBUG for use by the E command upon exit from the assembler.

Example: END>[program start]

Upon completion of the end directive, that is, END followed by a return or END S (start location) followed by a return, the assembler will display the number of unresolved forward references, if any. If all forward references have been resolved, the assembler will display a 0. In either case, the assembler then accepts the entry of any character to cause a return to UNIBUG.

Block Declaration (BSS)

The programmer may desire to declare a block of RAM for variable storage without initializing the space, such as setting aside space for the workspace registers. This may be accomplished with the BSS directive. BSS is an abbreviation for: Block Starting with a Symbol.

To set aside 32 bytes for a workspace starting at >360, one writes:

```
360:  WS BSS 32 RESERVE A BLOCK OF 32 BYTES
380:
```

This block's beginning address can be referred to by the label WS (for Workspace). The location counter is increased by 32 (>20) bytes from >360 to >380.

The assembler simply increases the location counter by the appropriate number of bytes specified by the directive:

BSS <number of bytes to be reserved>

The number of bytes specified must be zero or positive. The resulting value of the location counter is rounded down to an even number, if necessary.

Word Initialization (DATA)

The programmer may desire to cause a word or words of memory to be initialized to a particular value. This is particularly useful for entering a table of data as part of a program. This can be

accomplished with the DATA directive. At any point during the program assembly, one can insert a DATA directive in the opcode field followed by a literal constant or a symbol. The symbol may be a forward reference to be resolved later.

Examples:

Location	Statement	Comment
3B0:	DATA >1234	CAUSES LOCATION >3B0 TO BE INITIALIZED TO >1234.
3B2:	DATA AX	IF AX=>3456, CAUSES LOCATION >3B2 TO BE INITIALIZED TO >3456.
3B4:	DATA GH	IF GH IS AN UNDEFINED FORWARD REFERENCE, LOCATION >3B4 WILL BE INITIALIZED TO THE VALUE CORRESPONDING TO GH WHEN GH IS RESOLVED.

The operand for the DATA directive may consist of either: (1) an unresolved forward reference, (2) a numerical constant, (3) a defined symbol, or (4) a string of numerical constants and defined symbols interconnected by plus and minus signs indicating sum and difference, respectively. In the case of the string of sums and differences, no regard is given to carry or overflow.

Example: DATA 1+5-3 (equivalent to DATA 3)

The DATA directive supports a sequence of constants separated by commas.

DATA [constant (defined or undefined), constant,..., constant]Ret

An unresolved constant is acceptable only as the first operand of each DATA directive.

One can contrast the use of the BSS directive and the DATA directive as follows. The BSS directive simply sets aside memory space without initializing it; whereas, the DATA directive sets aside space and initializes it. The BSS directive is like buying raw property without making any changes to it. The DATA directive is like buying property and building a house on it. In any case, memory space is claimed.

Symbolic Constant Definition (Assembly-Time) (EQU)

A programmer can define a value for a symbolic constant by use of the equate (EQU) directive. For example, the symbol CD can be defined to be equal to hexadecimal A55A by entering CD EQU >A55A. Or one can set the symbol G to be equal to decimal 1000 by entering G EQU 1000. This directive allows the programmer to predefine certain values before entering related program assembly codes. Thus, if

one desires to use symbolic constants liberally throughout the program, he can easily implement them by using the EQU directive.

The EQU directive in the TM 990/189 symbolic assembler allows the user to assign the value of one defined symbol to the value of another defined symbol. For example, if A is already defined as decimal 1000, then merely enter the code B EQU A, and B also becomes defined as decimal 1000.

No directive is provided to change the value of a symbol once it is defined.

```
<label> EQU <defined constant>
```

Example:

```
CD EQU >A55A        CD REPRESENTS >A55A
FG EQU 15           FG REPRESENTS 15
A  EQU FG           A  REPRESENTS 15
```

String Constant Initialization (TEXT)

The assembler also provides a directive to allow the user to enter a sequence or string of characters and have these characters translated to ASCII code and stored as part of the program. For example, if one wants to store the text ABCD in memory in ASCII format, then he enters TEXT 'ABCD' and the corresponding memory locations would contain the values >4142 and >4344. This data represents to the computer system the letters A, B, C, and D. In fact, any character entered, other than the CANC character or the single-quote (') character, results in its ASCII code being entered into memory. The text string entered thereby may be as long as desired. It must be preceded by a single-quote character and terminated by a single-quote character. The single-quote character indicates inputs to be converted to ASCII. The only normal string terminator is a single-quote character. Furthermore, since the system's extended operation to output a string of characters (discussed in Chapter 8) expects a null (>00) to indicate the end of the text string, this directive automatically inserts a null (>00) at the end of the string in memory:

```
[label] TEXT '<character string>'
```

4.5 LABELS AND INSTRUCTION SYNTAX

Label and Operand Correction

There is a degree of correction capability within the label field and the operand field. A label (whether in the label field or in the operand field) can be corrected by continuing to enter the proper characters. For example, if the label BZ is entered

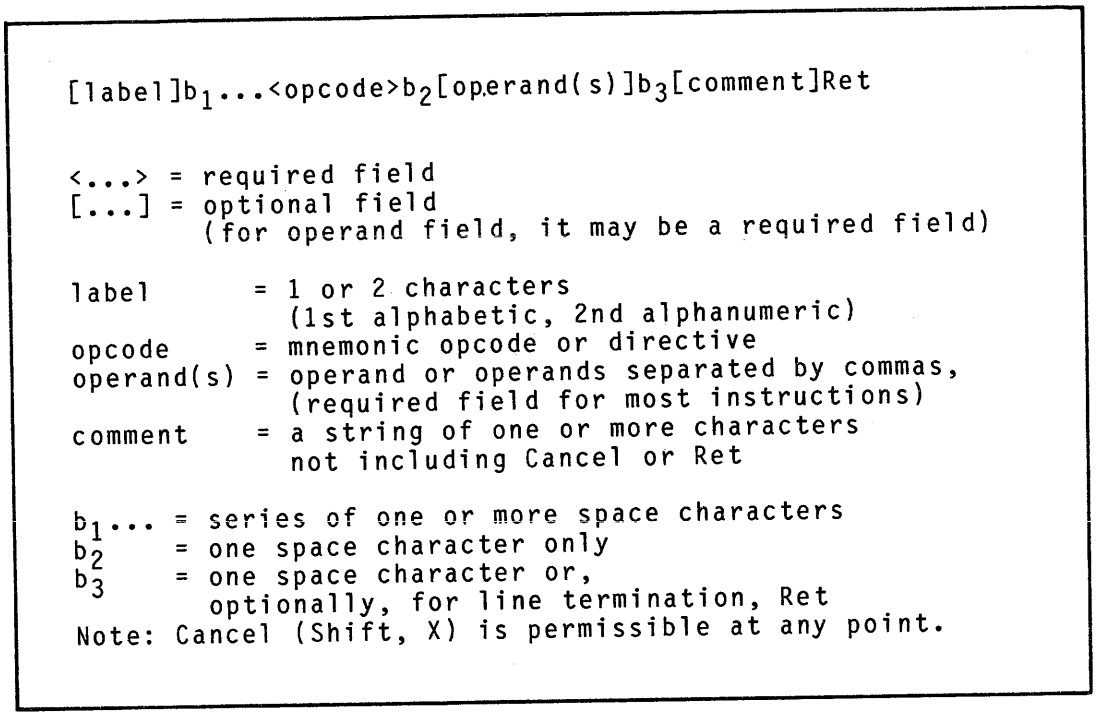


Figure 4-3. Instruction Syntax

and one wishes to change the label to CD, then it is corrected by simply entering CD before exiting the label field. In short, the assembler accepts the last two label characters entered in that field. If a single-character label is to be corrected, enter a one (1) indicating a single-character label and then the appropriate alphabetic character. These rules apply any place a label or a symbol is used. Not only can a label be corrected in the operand field, but a numerical constant also can be corrected in the operand field. For example, if a >1234 is entered but a >2234 is desired, it can be corrected immediately by entering 2234. The assembler accepts the last four characters of a hexadecimal constant. In contrast, an error in the opcode cannot be corrected without cancelling and re-entering the line.

Generally, it is not as easy to correct decimal operands. But if a programmer desires, this can also be accomplished in a similar manner, except that the assembler considers the last 16 decimal digits entered. This could mean that to do proper decimal correction on decimal constants one would need to enter 16 intervening zeroes. Consequently, it is probably best to cancel (shift X) and re-enter the line.

Instruction Syntax Review

Now, by way of review, the syntax for a particular line of code is restated (see Figure 4-3). The assembler expects, first,

either a space, or a label reference (1 or 2 characters, of which the first must be alphabetic, and the second, if any, alphanumeric) followed by a space. Between the label field and the opcode mnemonic, enter as many nonalphabetic characters as desired with the first being a space. The opcode field consists of one to four alphabetic characters. Between the opcode and the operand field, the assembler expects one space. This space actually may be any nonalphabetic character other than the CANC character. The operand field consists of one or two subfields each of which will be terminated by a nonalphanumeric character. If there are two operand subfields, it is generally expected that these will be separated by a comma, but, in fact, any nonalphanumeric character other than the CANC character is acceptable between the operand subfields. Between the operand field and the comment field there must be one space character. A CANC character means cancel a line. Following the operand field and in the comment field, a carriage return means terminate the line. The comment field consists of any character other than CANC or return (Ret). The comment field continues until the receipt of one of those two characters. Then, CANC and Ret have their normal meanings of line cancellation or line termination.

Concerning source-statement syntax, it is to be emphasized that:

- Labels on instructions must be entered without preceding spaces.
- Instructions without labels must be preceded by at least one space.
- Following the opcode only one space is allowed prior to the operand field.

Symbol Table and Unresolved Labels

The symbol table for the University Board Symbolic Assembler is always located in RAM starting at location >146. Its length is variable, with its current length in bytes maintained in RAM location >16. This explanation concerning the symbol table is for those interested in its details. Thus, some readers may desire to omit reading the remainder of this section.

In the following paragraphs of this section, the organization of the symbol table is documented, along with the necessary background for the user who desires to place entries in the symbol table directly.

It is emphasized that the length of the symbol table is variable and typically requires from four to eight bytes for each label or other symbol. It is actually a combination of three tables. It includes: (1) entries for defined labels and other symbols, (2) unresolved word references, and (3) unresolved jump references. Each entry consists of a label word and an address word, a total of four bytes.

If the entry is a resolved label or other symbol, then the label word will simply consist of the ASCII equivalent for the label. A single character symbol will be stored in its ASCII form preceded by an ASCII "1", (e.g., symbol A will be stored as >3141). The second word (the address word) will contain the address or constant equivalent or memory location corresponding to this label or symbol. For example, if label AC (ASCII >4143) is established to be at location >300, then the symbol entry is

```
4143     defined symbol reference entry
0300         (in the symbol table).
```

The unresolved word reference entry is similar, except that the label word will have a ONE in bit 0 (i.e., ASCII equivalent + >8000). In such a case, the address word points to the last location in which this label was used as a forward reference. For example, if the label AC is used as an unresolved forward reference in location >300, then the entry appears as

```
C143     unresolved word reference entry
0300         (in the symbol table).
```

Location >300 would either contain a pointer to the previous use of this label as an unresolved word reference, or a zero (to indicate the first use as an unresolved word reference).

The unresolved jump reference entry in the symbol table would appear similarly, except the label would contain a ONE in bit 8, (i.e., ASCII equiv. + >0080), and the address word would contain a pointer to the location of the last jump instruction using this unresolved label. For example, if location >300 had an unresolved jump reference to label AC, then the entry in the symbol table would be

```
41C3     unresolved jump reference entry
0300         (in the symbol table).
```

The right byte of the unresolved jump instruction will indicate the reach to the next previous unresolved jump instruction to the same label, or be assigned a byte value of -1, if there is no previous unresolved jump instruction to the same label.

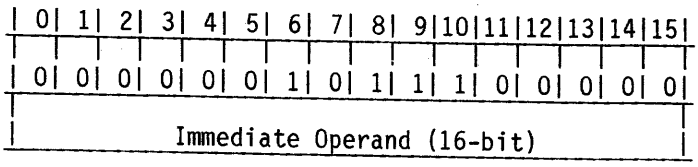
4.6 INSTRUCTION SUBSET 2

Initialization of the Workspace Pointer

The first instruction of Subset Number 2, summarized in the Instruction Summary 4-1, allows the programmer to set the workspace pointer to any desired value.

LOAD WORKSPACE POINTER IMMEDIATE LWPI

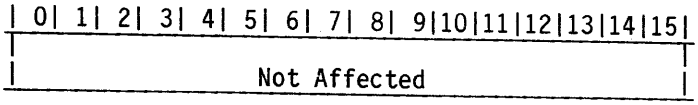
CODE: LWPI IOP



RESULT: IOP ⟶ (WP)

Length: 2 words

STATUS REGISTER:



OPERATION:

Replace the contents of the workspace pointer (WP) register with an immediate operand representing the beginning address of 16 contiguous words to be used as the "workspace." This changes the current workspace pointer.

NOTES:

The LWPI instruction is typically used as the initial instruction for a stand-alone program. A stand-alone program is one that does not depend upon any utility program or operating system for any of its functions. If the program is the only one in the machine, then one must set the workspace pointer upon initial execution to ensure that the workspace pointer is pointing to the proper place.

If programming in conjunction with an operating system, then typically the workspace pointer is set by the operating system. With the UNIBUG monitor, the user can set the workspace pointer using the W command prior to program execution, or he can set the workspace pointer using the LWPI instruction as the first instruction of the program. It should also be noted that the previous contents of the workspace pointer register is lost without recovery, unless it is saved prior to execution of this instruction.

Example:

```
LWPI >03B2      INITIALIZE WP TO >03B2
```

Byte Instructions--Manipulation and Arithmetic

There are six instructions which deal only with byte operands (see Table 4-1). The first three of these have counterparts in word instructions already discussed in Chapter 3. Their operation is the same as with their word counterparts, except the operation is on a byte basis. Each of the six byte instructions has the same machine code format as that discussed with the Move Word, Add Words, and Subtract Words instructions.

In fact, there are 12 instructions which share this same machine code format, six of which are word instructions and the other six byte instructions. Each byte instruction has several distinctive features which should be carefully noted. These instructions occur in functional pairs. The first pair consists of the Move Word and the Move Byte instructions. The Move Byte instruction is the byte counterpart to the Move Word instruction. See Instruction Summary 4-2.

The next byte instruction to be considered is the Add Bytes. This instruction allows one to add bytes, and its operation is similar to that of the Add Words instruction. Its highlights are given in Instruction Summary 4-3.

The third byte instruction to be considered is the Subtract Bytes instruction. It is a counterpart to the Subtract Words in struction and is summarized in Instruction Summary 4-4.

With one exception, the OP status bit is affected only by what one may call "true" byte instructions. All instructions which are only for byte operations end their name with the word byte or bytes, and the corresponding mnemonic opcode always ends with a B to indicate this. Furthermore, the reader should note that byte instructions work on "byte-tight" compartments.

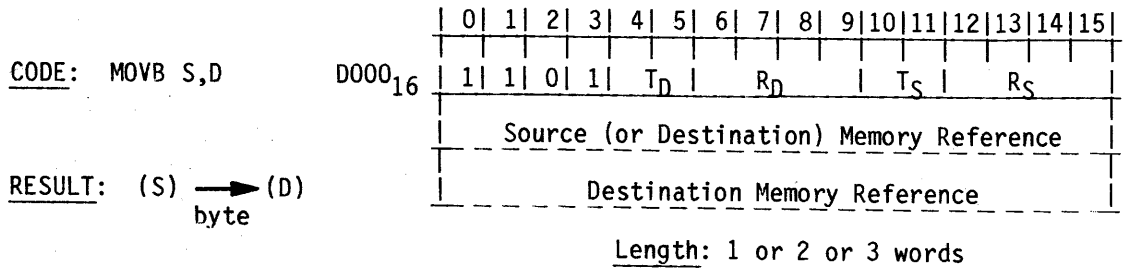
Table 4-1. Byte Instructions

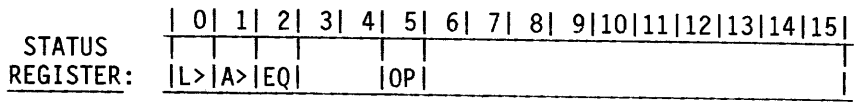
(Byte operands only)

MOVB	CB
AB	SOCB
SB	SZCB

MOVE BYTE

MOVB





OPERATION:

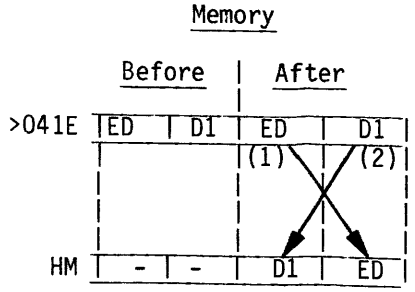
The destination operand (8 bits) is replaced with a copy of the source operand (8 bits). The 8-bit operand is compared with zero and the L>, A> and EQ status bits are correspondingly affected. The number of ONE's in this operand are counted and the odd parity (OP) bit is set to ONE if this count is odd and to ZERO if even.

NOTES:

The MOVB instruction is used to move or copy a byte from one general memory location to another general memory location, and is specifically needed when one needs to manipulate byte operands rather than word operands. One should also note in conjunction with this byte instruction that a particular status bit is affected which was not affected by the corresponding word instruction, i.e., the odd parity status bit (OP). The odd parity status bit is specifically related to byte instructions. The OP status bit resulting from this instruction can be used to generate and check odd and even parity of ASCII characters.

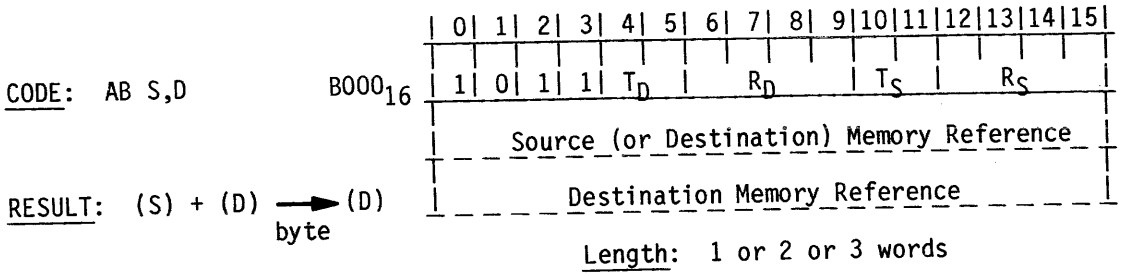
Example:

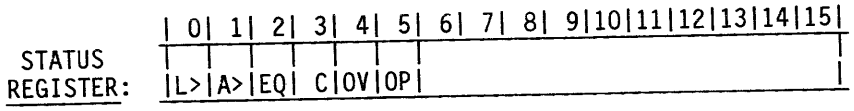
(1) MOVB @>041E,@HM+1
(2) MOVB @>041F,@HM

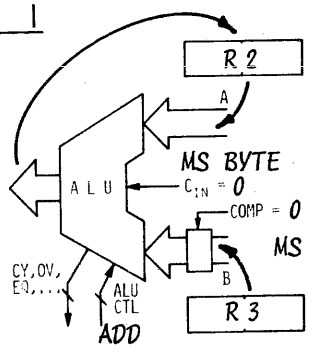


ADD BYTES

AB







OPERATION:

The source operand (8-bits) is added to the destination operand (8-bits) and the resulting sum (8-bit) replaces the destination operand.

The 8-bit result is compared with zero and the L>, A>, and EQ status bits are correspondingly affected. The addition operation based upon 8-bit values affects the C and OV status bits.

The number of ONE's in the resulting byte are counted and the odd parity (OP) bit is set to ONE if the count is odd and to ZERO if even. if even.

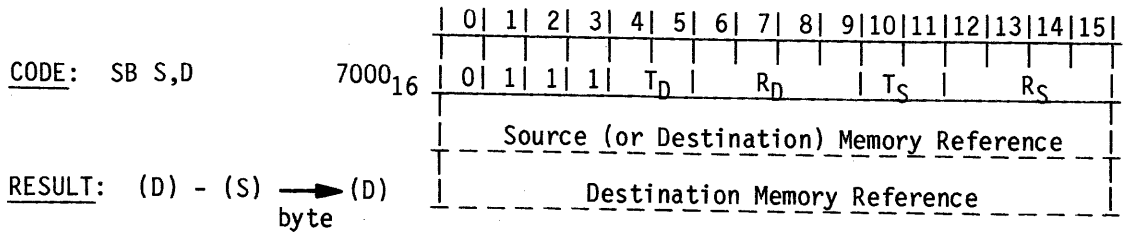
NOTES:

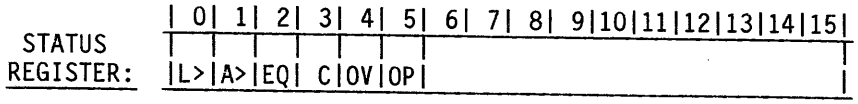
This instruction allows one to add signed 8-bit numbers (in the range -128 to +127), rather than 16-bit numbers as used with the Add Words instruction. Furthermore, when addressing bytes, reference to a workspace register causes the 8 leftmost bits of that workspace register (which is the even byte) to be used as the operand. This is true because a register number is simply a pseudonym for an even memory address. For example, suppose the workspace pointer contains >4200, and one refers to R0 in a byte instruction. Since R0 is a pseudonym for >4200, then the reference must be to the memory byte addressed >4200 and not to the byte >4201. This applies to all byte instructions.

Example:

```
AB R2,R3              ADD LEFT BYTES OF R2 AND R3, PUT RESULT IN LEFT
                      BYTE OF R3.
AB @>4200,@>4201      ADD LEFT BYTE OF MEMORY LOCATION >4200
                      TO RIGHT BYTE OF MEMORY LOCATION >4200 (i.e.
                      >4201); AND PUT RESULT IN >4201
```

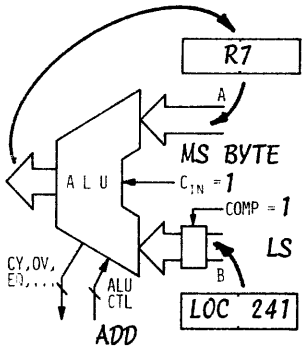
SUBTRACT BYTES SB





OPERATION:

The source operand (8-bits) is subtracted from the destination operand (8-bits) and the resulting difference (8-bits) replaces the destination operand. The 8-bit result is compared with zero and the L>, A>, and EQ status bits are correspondingly affected. The subtraction operation affects the C and OV status bits based upon 8-bit values. The number of ONE's in the resulting byte are counted and the odd parity (OP) bit is set to ONE if the count is odd and to ZERO if even.



NOTES:

The Subtract Bytes instruction is used to subtract signed 8-bit integers. These numbers are in the range of -128 to +127. One can use the subtract instruction to clear an operand. For example, to clear both bytes of register 0 one could write: S R0,R0 which would cause both bytes of R0 to be set to 0. The Subtract Bytes instruction can be used similarly. For example, to clear the left byte of R0 only, one may write: SB R0,R0 which will clear only the left byte of R0 and will not affect the right byte. The subtract bytes instruction could be used similarly with any of the addressing modes to clear only one of the two bytes of a particular word. This is significant because there is a special clear instruction for word operands, but not for byte operands.

Example:

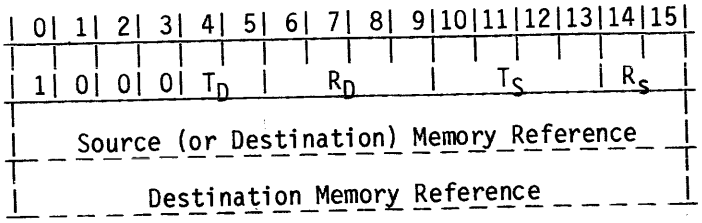
SB @>0241,R7	SUBTRACT RIGHT BYTE OF MEMORY WORD >240 FROM LEFT BYTE OF R7 AND PLACE RESULT IN LEFT BYTE OF R7
SB @>402,@407	SUBTRACT LEFT BYTE OF MEMORY WORD >402 FROM RIGHT BYTE OF MEMORY WORD >406 AND PLACE RESULT IN RIGHT BYTE OF MEMORY WORD >406.

Instruction Summary 4-4

COMPARE WORDS

C

CODE: C S,D

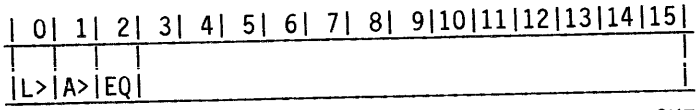


RESULT: Three status bits affected.
Related Truth Eqn's

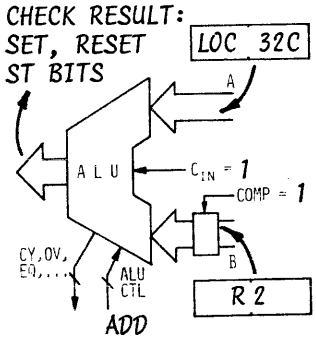
Length: 1 or 2 or 3 words

L>:=[(S)>(D)] (logical)
A>:=[(S)>(D)] (arithmetic)
EQ:=[(S)=(D)]

STATUS REGISTER:



OPERATION:



The source operand is compared to the destination operand and the L>, A>, and EQ status bits are set or reset accordingly.

The L> status bit is set to ONE if the source word is logically greater than the destination word; otherwise it is reset to ZERO.

The A> status bit is set to ONE, if the source word is arithmetically greater than the destination word; otherwise it is reset to ZERO.

The EQ status bit is set to ONE, if the source word is equal to the destination word; otherwise it is reset to ZERO.

The result can also be represented by indicating that the destination operand is equivalently subtracted from the source operand and the result compared with zero (as usual) and three status bits affected with the result being discarded (assuming overflow does not occur).

(S) - (D)
(3 status bits affected)

NOTES:

The Compare Words instruction is a very important one, if the programmer needs to compare two 16-bit operands to each other rather than to zero. Normally, the next instruction is a conditional jump.

Example:

```
C R2,@>032C      COMPARE (R2) WITH (>032C) AND
JEQ CK           JUMP IF EQUAL TO LOCATION CK
```

Compare Instructions

There are five different compare instructions in the TMS 9980A instruction set, three of which are of the general type, and two of which are of a special type. Previously, the CPU compared the result of an instruction with zero and appropriately affected the status register. With compare instructions (of the general type), the CPU compares the value of one operand with that of another and ascertains whether they are equal, or which one is greater (logically and arithmetically).

Compare Words Instruction. The first of these three compare instructions of a general type is the Compare Words instruction which is presented in Instruction Summary 4-5.

In order to ensure success in using the Compare Words instruction, the programmer should remember the following three questions in relation to how the status bits are set.

Is the source operand logically greater than the destination operand (i.e., in absolute terms)?

A yes answer to this question means the instruction will set the logical greater than status bit (L>) to ONE.

Is the source operand arithmetically greater than the destination operand?

A yes answer to this question means the instruction will set the arithmetic greater than status bit (A>) to ONE.

Is the source operand equal to the destination operand?

A yes answer to this question means the instruction will set the equal status bit (EQ) to ONE.

It is essential for the programmer to get the order of the two operands correct, if this instruction is to be used successfully. Several examples of results from the operation of the Compare Word instruction are given in Table 4-2 to illustrate the setting of the status bits.

The machine code format for the Compare Words instruction uses the same format as the Move Word, Move Byte, Add Words, Add Bytes, Subtract Words, and Subtract Bytes instructions.

Compare Bytes Instruction. The next instruction, Compare Bytes, allows the programmer to compare bytes, rather than words, from two general memory locations. This is helpful for byte-oriented data. See Instruction Summary 4-6.

Table 4-2. Results of Compare Words Instruction

Operands		Resulting Status Bits		
(Source)	(Destination)	L>	A>	EQ
7FFF	0000	1	1	0
FFFF	0000	1	0	0
8000	0000	1	0	0
8000	7FFF	1	0	0
7FFF	8000	0	1	0
7FFF	7FFF	0	0	1

Compare Immediate Instruction. Another compare instruction, Compare Immediate, allows the programmer to indicate a specific number (IOP) to be compared to the contents of a register without using an additional register. This instruction is the third of five compare instructions. Futhermore, it is fourth of five instructions with "Immediate" as part of its name indicating an immediate operand as the second word of the instruction. See Instruction Summary 4-7.

More Jump Instructions

In the previous chapter, jump instructions were introduced and the JMP and JGT instructions were discussed in detail. As was previously indicated, there are 13 different jump instructions (see Table 3-2). Three more jump instructions: JEQ, JNE, and JLT will now be introduced.

Recall that jump instructions use PC relative addressing. The jump instruction contains an 8-bit signed field called a displacement. This displacement is in words. When the jump is taken, this displacement (in bytes) is added to the contents of the PC (in bytes) which is then pointing at the next instruction, and the result placed as the new value for the PC. It is important to note that when the displacement value is added to the contents of the PC, the machine code displacement value (in words) must first be doubled (converted to the number of bytes).

COMPARE BYTES

CB

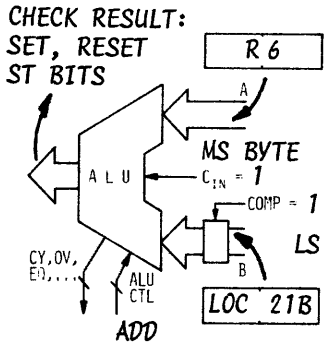
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: CB S,D 9000_{16}	1	0	0	1	T_D		R_D				T_S		R_S			
RESULT: Four status bits affected. Related Truth Eqn's	Source (or Destination) Memory Reference															
	Destination Memory Reference															

L>:=[(S)>(D)] (logical)
A>:=[(S)>(D)] (arithmetic)
EQ:=[(S)=(D)]
OP:=(S)(odd number of ONE's)

Length: 1 or 2 or 3 words

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	L>	A>	EQ			OP										

OPERATION:



The source operand (8-bits) is compared to the destination operand (8-bits) and the L>, A>, and EQ status bits are set or reset accordingly. The OP status bit is also affected based on the source operand.

The L> status bit is set to ONE if the source byte is logically greater than the destination byte; otherwise, it is reset to ZERO.

The A> status bit is set to ONE if the source byte is arithmetically greater than the destination byte (considering these as signed 8-bit integers); otherwise, the A> status bit is reset to ZERO.

The EQ status bit is set to ONE if the source byte is equal to the destination byte; otherwise, it is reset to ZERO.

The OP status bit is set to ONE if the number of ONE's in the source byte is odd; otherwise, it is reset to ZERO.

NOTES:

The Compare Bytes instruction is the eighth of the twelve instructions discussed thus far which have the same format of general source and general destination. Normally, the CB instruction is followed by a conditional jump.

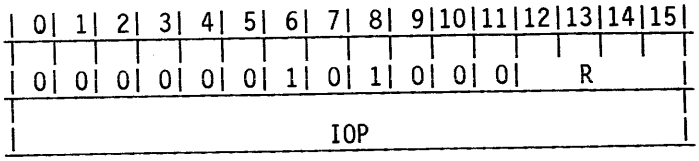
Example:

```
CB @>021B,R6    COMPARE LS BYTE OF (>21A) WITH MS BYTE OF R6
JGT $+6         SKIP NEXT 2 WORDS IF GREATER
AI R1,10        OTHERWISE ADD 10 TO (R1)
```

COMPARE IMMEDIATE

CI

CODE: CI R,IOP 0280_{16}

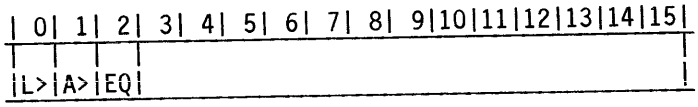


RESULT: Three status bits affected.
Related Truth Eqn's

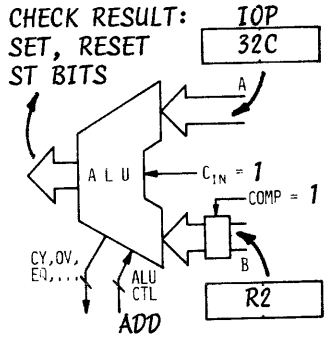
Length: 2 words

L>:=[(S)>(D)] (logical)
A>:=[(S)>(D)] (arithmetic)
EQ:=[(S)=(D)]

STATUS REGISTER:



OPERATION:



The contents of the indicated register are compared to the 16-bit immediate operand and the L>, A>, and EQ status bits are set or reset accordingly.

The L> status bit is set to ONE, if the contents of the register are logically greater than the 16-bit immediate operand; otherwise it is reset to ZERO.

The A> status bit is set to ONE, if the contents of the register are arithmetically greater than the 16-bit immediate operand; otherwise it is reset to ZERO.

The EQ status bit is set to ONE, if the contents of the register are equal to the 16-bit immediate operand; otherwise it is reset to ZERO.

The result can also be represented by indicating that the immediate operand is equivalently subtracted from the contents of the register and the result compared with zero (as usual) and three status bits affected with the result being discarded (assuming overflow does not occur).

(S) - (D)
(3 status bits affected)

NOTES:

The Compare Immediate instruction is a very important one, if the programmer needs to compare the contents of a register to a specific number. Its use avoids the requirement to load the specific number into a register and the specific number is visible as part of the instruction. Generally, the next instruction is a conditional jump.

```
Example:    CI R2,>032C     COMPARE (R2) WITH >032C  AND
            JEQ CK          JUMP IF (R2) EQUAL TO LOCATION CK
```

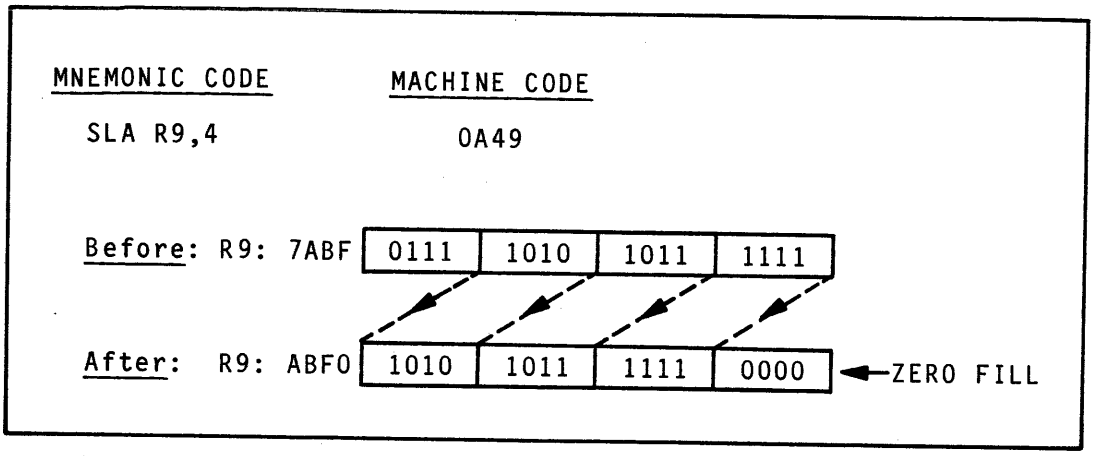


Figure 4-4. Illustration of SLA Operation

The Jump if Equal (JEQ). The Jump if Equal (JEQ) instruction allows the programmer to test the EQ status bit. It causes a modification in the program counter if the EQ status bit is ONE; otherwise, the transfer of control will not occur and the next instruction will be fetched and executed. Typically, JEQ is used to test if a number is zero. Details are presented in Instruction Summary 4-8.

The Jump if Not Equal (JNE). The Jump if Not Equal (JNE) instruction also allows one to test the EQ status bit. However, as shown in Instruction Summary 4-9, the transfer of control occurs when the EQ status bit is ZERO.

The Jump if Less Than (JLT). The Jump if Less Than (JLT) instruction completes the pair of arithmetic status bit tests (JGT was discussed in Chapter 3). It allows one to test for negative numbers. It causes a jump to occur when the arithmetic greater than (A>) status bit is zero, and the EQ status bit is zero. In effect, it means jump if negative. The JGT and JLT instructions are the only two jump instructions which test the A> status bit. Instruction Summary 4-10 provides the detailed description for the JLT instruction.

Addition Instructions

Shift Left Arithmetic. The next instruction, Shift Left Arithmetic, is a very powerful instruction. This instruction allows the programmer to do multiplication of signed numbers by powers of two. Plus, it allows him to shift bits to the left, thus rearranging "digits," viewing them logically. See Figure 4-4. Details are provided in Instruction Summary 4-11.

JUMP IF EQUAL — JEQ

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: JEQ Location 1300_{16}	0	0	0	1	0	0	1	1	Displacement (words)							

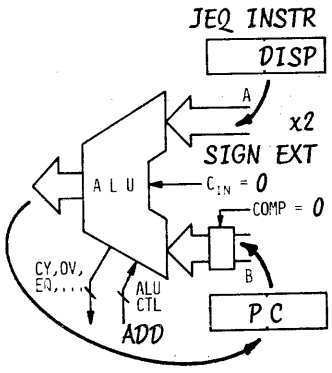
Length: 1 word

RESULT: If EQ = 1, (PC) + Displacement in bytes ⟶ (PC)
If EQ = 0, (PC) unchanged

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not affected															

OPERATION:

When the equal status bit (EQ) is set to ONE, the signed displacement (in bytes) in the instruction (8-bit field) is added to the PC and the result is placed in the PC; otherwise the PC is not affected by the instruction. The status register is not affected by the instruction operation.



NOTES:

The Jump if Equal instruction is used any time one desires to jump when the equal status bit (EQ) is set to ONE. See the example below.

Example:

```
C R4,R5     COMPARE (R4) TO (R5)
JEQ TP      IF EQUAL, JUMP TO LOCATION TP
            OTHERWISE, EXECUTE NEXT INSTR
```

JUMP IF NOT EQUAL

JNE

CODE: JNE Location 1600_{16}

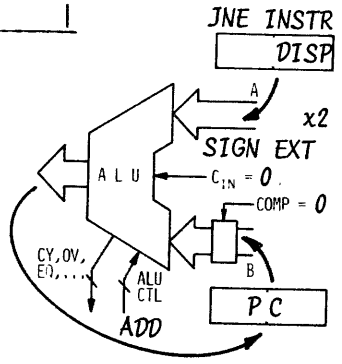
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	1	0	Displacement (words)							

Length: 1 word

RESULT: If EQ = 0, (PC) + Displacement in bytes ⟶ (PC)
If EQ = 1, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															



OPERATION:

If the EQ status bit is ZERO, then the signed displacement (in bytes) of the machine code instruction is added to the contents of the PC and the sum placed into the PC; otherwise, the PC is unchanged.

NOTES:

Use the JNE instruction to test the EQ status bit for ZERO. For example, it can be used to implement a loop. A counter (memory location or register) can be loaded with a number indicating the number of times the loop is to be executed. Each time through the loop, the number in the counter is decremented. The JNE instruction is used to test the number in the counter and continue the loop if it is non-zero. When the counter value becomes zero, execution "falls" through to the next block of instructions.

Example:

```
    LI R1,1500      INITIALIZE DELAY COUNTER R1 WITH 1500(10)
LP  DEC R1          DECREMENT COUNTER
    JNE LP          JUMP BACK TO LP IF (R1) <>0
    NEXT            OTHERWISE, DELAY COMPLETED, EXECUTE NEXT
NX  -------         INSTRUCTION
```

JUMP IF LESS THAN

JLT

CODE: JLT Location 1100_{16}

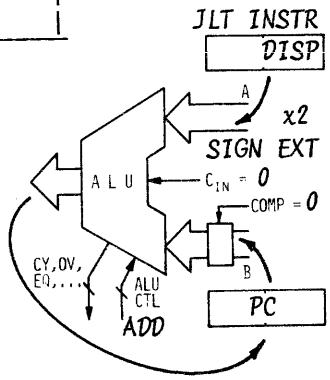
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	0	0	1	Displacement (words)							

Length: 1 word

RESULT: If A> = 0 and EQ = 0, (PC) + Displacement in bytes ⟶ (PC)
Otherwise, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															



OPERATION:

If both the A> status bit and the EQ status bit are ZERO, the signed displacement (in bytes) is added to the contents of the PC and the sum placed into the PC; otherwise, the PC is unchanged.

NOTES:

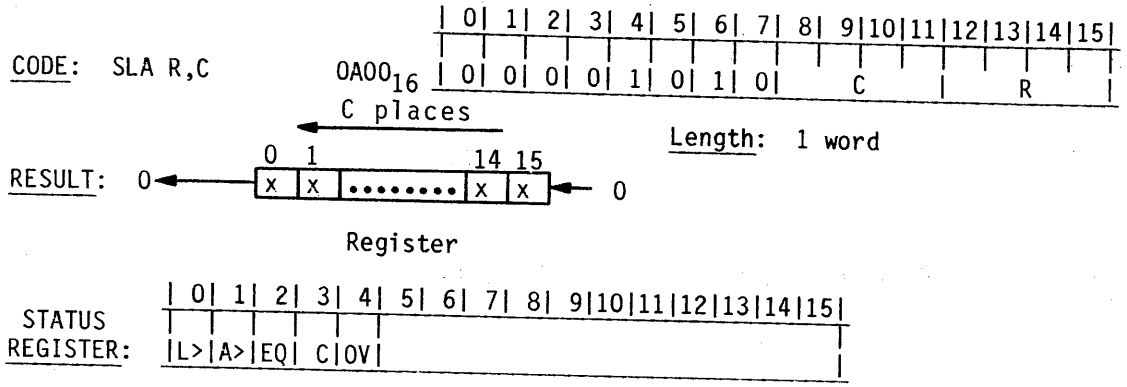
The JLT instruction is used to test for negative numbers. It can be used to implement a loop by setting the initial count to some negative number, then continuing until the counter goes to zero. Thus, the JLT instruction can cause the loop to continue, and allows execution to go to the next block when zero is reached.

Example:

```
    LI R1,>FF01     INITIALIZE DELAY COUNTER TO -255(10)
LP  INC R1          INCREMENT COUNTER
    JLT LP          JUMP BACK TO LP IF (R1)<0
    NEXT            OTHERWISE, DELAY COMPLETED, EXECUTE
NX  -------         NEXT INSTRUCTION
```

SHIFT LEFT ARITHMETIC

SLA



OPERATION:

The contents of the specified workspace register (including the sign bit) are shifted left the number of bits specified by the count (C) with ZERO's filling the vacated bit positions. The last bit shifted out is retained in the carry status bit.

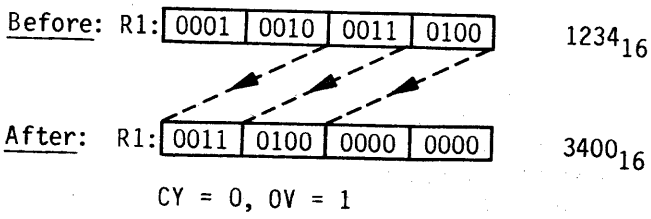
Any sign change during the shift process causes the OV status bit to be set to ONE.

If the count (C) is equal to zero, then the right 4 bits of R0 are used as the shift count. If these 4 bits are also zero, then the shift count is 16.

NOTES:

The SLA instruction can be used to multiply numbers by powers of two or simply to accomplish necessary bit shifting. It also can be used to introduce desired short delays since shifting takes a relatively long amount of time (microseconds).

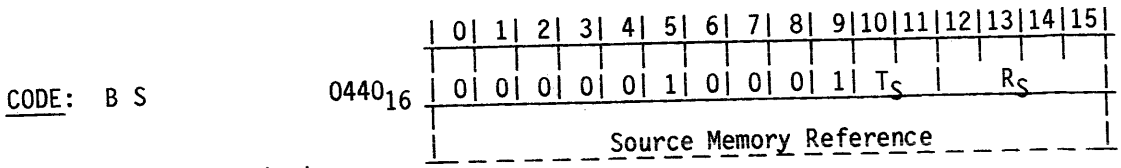
Example: SLA R1,8 MULTIPLY CONTENTS OF R1 BY 256 (2 TO THE 8TH POWER)



CY = 0, OV = 1

BRANCH

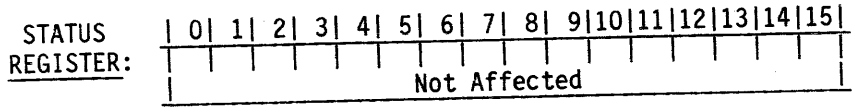
B



RESULT: SA ⟶ (PC)

Length: 1 or 2 words

[Source Address]



OPERATION:

Replace the contents of the PC with the source address (not the contents). This transfers control to the specified source address.

NOTES:

The Branch (B) instruction (also known as Branch Unconditional) allows one to transfer control when the desired next instruction to be executed is out of reach of the JMP unconditional instruction.

The Branch instruction also provides a more versatile type of modification to the program counter since all five of the general modes of addressing can be used. One can branch to a particular memory address by memory symbolic addressing (or absolute addressing rather than relative addressing). It also allows one to branch to a location pointed to by the contents of the register (i.e., register indirect addressing). These two are the primary modes of addressing used with this particular instruction. For example, if one needed to jump a long distance on the equal status condition and the reach were too great, then one could write a JNE instruction to jump around the succeeding branch instruction. This is equivalent to a "branch on equal."

Example:

```
    JNE NX      COMBINATION TO PERFORM
    B @DT       BRANCH ON EQUAL OPER.
NX  -------
```

Note that in the TMS 9980A instruction set, there are no conditional branch instructions. Further note, that a branch means long range and jump means short range. Another example is the use of a branch with register indirect addressing. If one were to store a return address in a particular register prior to executing a block of code, then a branch indirect on that register will return control to the desired location. This is generally a method employed to implement subroutines, as is discussed in Chapter 8.

With the SLA instruction, the sign bit is shifted along with the other bits and might not be preserved. The above illustration (SLA R9,4) will shift the contents of workspace register 9 four places to the left filling the vacated bits with zero. This will, in effect, multiply its contents by 16. If this arithmetic operation is not successful, i.e., the resulting number is too large, then the OV bit will be set to ONE to indicate this. Register 0 (with its right four bits) can be used to implement a variable shift count. This occurs when the count in the instruction is indicated as zero, then the right four bits of register 0 are used to determine the appropriate shift count. If these four bits contain the value zero, then sixteen is the shift count. To the programmer this means that register 0 has a special use. Take note.

It is interesting to observe what occurs when the shift count of the SLA instruction is zero and the right four bits of R0 are all ZERO's. The contents of the shifted register will go to 0. The least-significant bit will be placed in the carry bit. The OV status bit will indicate if any of the 16 bits shifted were a binary ONE. The instruction requires several microseconds to execute, so it may be useful in this mode to introduce a slight delay, if desired.

Try a program segment to illustrate the execution of the SLA instruction. Enter the assembler with the A command followed by: 200. This causes the assembler to display 0200. Enter the following lines of code and terminate each line with a Ret (comments optional). Respond to each object code display with an Ret.

```
ST  LI R9,>ABCD     LOAD R9 WITH >ABCD
B   SLA R9,4        SHIFT LEFT 4 PLACES
C   JMP C           SPIN
D   END ST          END DIRECTIVE
                    (START AT ST)
```

The resulting object will be:

Location	Object
0200	0209
0202	ABCD
0204	0A49
0206	10FF

Now after a return to UNIBUG as a result of the END directive, set WP to 280_{16} with the W command and verify that PC is at >0200 with the P command. Inspect R9 with the R command. Now execute the program with the E command, and return to UNIBUG by activating the LOAD switch and depressing the (Ret) key. Verify the execution of the SLA instruction by inspecting R9 and observing its contents which should be >BCD0 (with WP set to >280).

Branch Instruction. The Branch instruction allows the program to branch directly to any location in memory. It corresponds to

the JMP instruction, in that the change in the program counter occurs regardless of the contents of the status register. There are no conditional branch instructions. It differs from the JMP instruction in that the instruction points directly to the specific destination rather than changing the program counter relative to the current location. For short-range program counter changes, both B @A and JMP A accomplish the same result, except B @A is a two-word instruction and JMP A is a one-word instruction. For long-range changes in the program counter, the B instruction is the only option (long-range means beyond the reach of the jump instructions).

If the current instruction is at location >280 and it is necessary to transfer program control to location >300, then it is in range of a jump instruction.

Example:

Location	Object	Instruction
0280	103F	JMP >300

If in contrast the need is to transfer control to >388 then the required reach is >108 bytes (LC) or >83 words (PC) with the limit being >7F. Since the reach is too great, it would need to be implemented with a B instruction:

Location	Object	Instruction
0280	0460	B @>388
	0388	

See further highlights of the B range instruction in Instruction Summary 4-12.

4.7 MEMORY MAP

The memory map for the TM 990/189, shown in Figure 4-5, indicates where the various segments are located in memory. A key location to remember is the user entry (or return) point to UNIBUG which is: >3000 (for System ROM REV.B). This may vary with other revisions of the System ROM.

The University Board's memory is laid out with areas for

- System RAM
- Assembler Symbol Table*
- User RAM
- On-board Expansion RAM
- On-board Expansion ROM
- Off-board Expansion ROM and RAM
- System ROM.

*The assembler Symbol Table requires additional system RAM during the operation of the symbolic assembler, but this space is available to the user's at other times (indicated by the line at >146), and it should be noted that the length of the Symbol Table is purely a function of the user's source program.

4.8 FTN1 PROGRAM REVISITED

The program discussed in Chapter 3 to calculate $[4(N1-N2)]^2$ can be used to illustrate the use of some of the instructions and concepts presented in this chapter. The LWPI instruction can be used to set the WP during program execution. The SLA instruction can be used to multiply by 4. The AORG, BSS, and END directives can be included to set the location to start assembly, to set aside space for a workspace, and to exit the assembler. Furthermore, the program can be revised to initialize the program data at the time of program assembly using the DATA directive.

The FTN1 program in Chapter 3 is listed below.

```
   MOV  R1,R4
   S    R2,R4          FORM DIFF: N1-N2 IN R4
   ABS  R4             FORM ABS VALUE OF DIFF
   A    R4,R4          MULT DIFF BY 4 BY ADDING
   A    R4,R4            TO ITSELF TWICE
   MOV  R4,R5          SET UP LOOP CTR
LP A    R4,R0          ADD 4 |DIFF| TO RESULT
   S    R3,R5          DECR LOOP CTR
   JGT  LP             JUMP IF LOOP CTR +
OU JMP  OU             STOP
```

This FTN1 program is slightly revised and placed in assembler format as shown below. Recall that assembler syntax requires that only one space is used between the mnemonic opcode and the operand.

```
   AORG >200
UB EQU >3000           UNIBUG RETURN
GO LWPI WS             INIT WP
   CLR R0              CLEAR RESULT ACCUM.
   MOV R1,R4
   S R2,R4
   ABS R4
   SLA R4,2            MULT. DIFF BY 4
   MOV R4,R5
LP A R4,R0
   DEC R5              DEC LOOP CTR
   JGT LP
   B @UB
WS DATA 0,5,2          (R0)=0, (R1)=5, (R2)=2
   BSS 26              26 BYTES FOR R3-R15
   END GO
```

This program can be assembled and executed. It performs the same function as the earlier program in Chapter 3. Different data items can be used by simply changing the contents of R1 and R2. (The W command should be used to set the workspace pointer to the address of WS.)

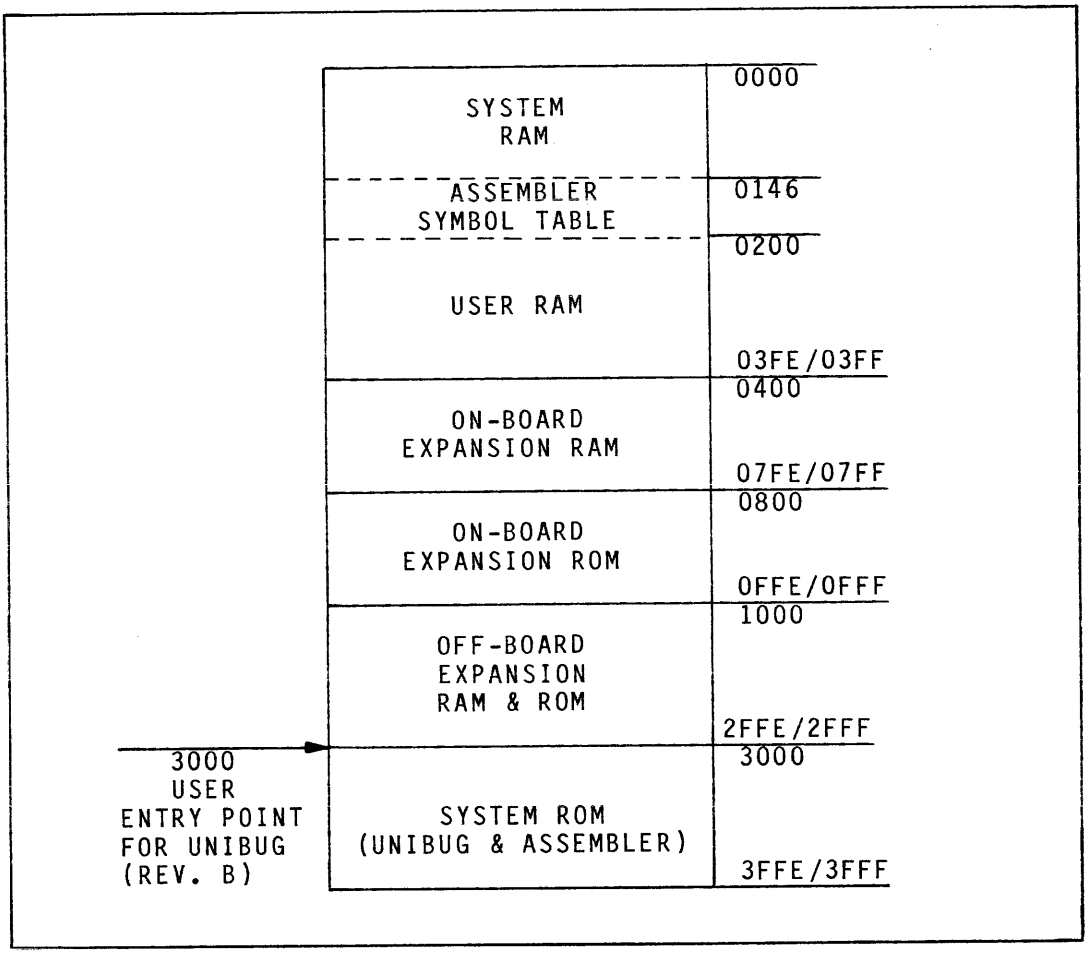


Figure 4-5. TM 990/189 Board Memory Map

4.9 PROGRAM EXAMPLE: SORT ALGORITHM

Program Idea

The program idea is to sort a list of numbers in memory into arithmetic (or algebraic) order so that the greatest number is first, and the second greatest is second, and the least value is last.

The Sort Algorithm

The algorithm demonstrated for this sort is the replacement type. The approach takes the first item on the list, compares it with the second item and determines which is greater. If the first is not greater than the second, swap the two. Then compare the first with the third; and swap, if necessary. Continue to compare

the first with the fourth, etc. Finally, after a comparison is made between the first and the last numbers in the list (and swapped if necessary), the greatest value will be first. The second greatest value is determined by comparing the second value in the list with the third and swapping, if necessary. The second is compared to the fourth, etc., until the second greatest is in the second slot. This continues until, finally, the next-to-the-last slot is compared with the last slot and swapped, if necessary. At this point, the program terminates.

Program Specification

The program specification can be stated as follows.

Write a program to implement the replacement sort algorithm. The word file to be sorted is located from memory location >0300 to >031E. Sort this file so that the greatest arithmetic number resides in location >0300. Use register indirect autoincrement addressing within the program wherever possible. Conclude the program with a branch to the UNIBUG monitor entry point. Use the following list of numbers to test the sort (enter into locations >300 through >31E).

1, 2, >30, >400, 8, >1600, 6, >8000,

20, >2000, 7, 10, 15, 17, >E, >FFFE.

Location	Number
>300	1
>302	2
.	.
.	.
.	.
>31E	>FFFE

Program Flowchart

To develop the program, a flowchart is constructed as shown in Figure 4-6. This flowchart indicates that the first step is to initialize the workspace pointer, since this program is to be implemented as a stand-alone program. Furthermore, the flowchart reflects the concept of using two pointers to implement the algorithm. The master pointer points to the current slot where the greatest value will be placed in this particular "pass," while the inner pointer points to the next and subsequent items to be compared with this initial item. The master pointer is initialized to the address of the beginning of the file, and on each pass the inner pointer is reinitialized to equal the master pointer. There is no need to compare the initial item with itself; therefore, the inner pointer is incremented to the next item at the beginning of each pass.

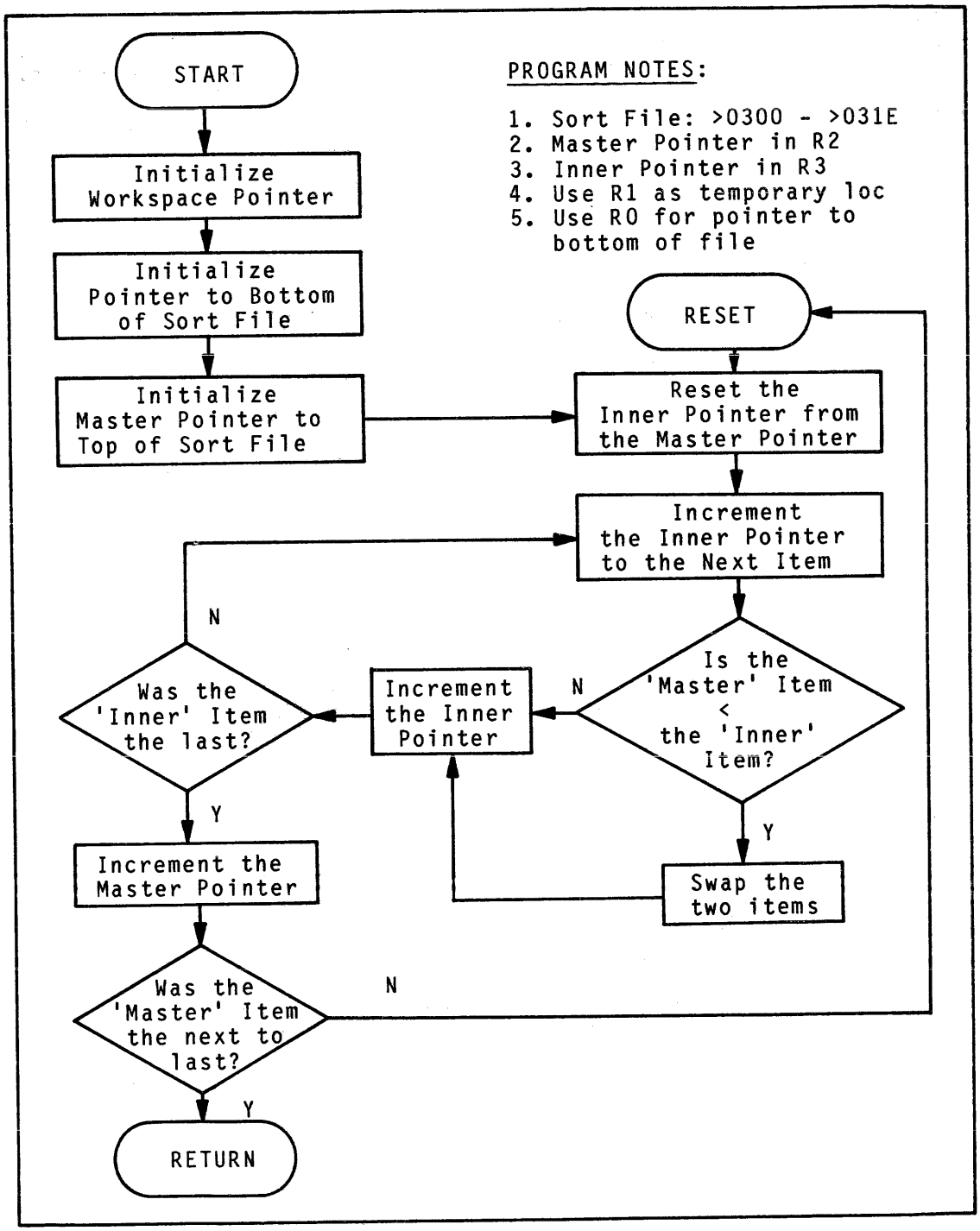


Figure 4-6. Replacement Sort Program Flowchart

The comparison can be accomplished by the compare instruction. The incrementing of the inner pointer can be accomplished using register indirect autoincrement.

To swap two values, three locations must be used. In this program, register four is used as an intermediate storage point. In the swap routine an autoincrement has already been performed on the inner pointer; therefore, it is necessary to index back with an offset of -2 to be able to pick up the appropriate item to be swapped. After each comparison of the items being sorted, the inner pointer value must be compared with the address of the last item in the file, to determine if the inner pointer is now greater than the address of the end of the file. If it is, then the current pass has been completed and the highest item in the string of comparisons has been determined. At this point, the master pointer is moved (by incrementing) to the next slot in the file where a check is made to determine if the master pointer is pointing to the end of the file. If so the program terminates. Otherwise, the inner pointer is reinitialized and another pass is begun.

Source Code

The source code for implementing the replacement sort is given in Figure 4-7. Observe that the initialization of the workspace pointer is directly implemented by the LWPI instruction. The master pointer is initialized simply by using a Load Immediate instruction. The inner pointer is initialized by moving the value in the master pointer to the inner pointer. The test for one number being arithmetically less than the other is accomplished by using the Compare Words instruction with the Jump If Greater Than instruction. The inner pointer is incremented by using register indirect autoincrement addressing with the Compare Words instruction. The inner pointer and the master pointer are compared with the limit value (stored in a register) by using the Compare Words instruction.

Generation of Object Code

To begin the assembly of the source code, enter the letter A on the terminal while under UNIBUG monitor control. This causes UNIBUG to transfer control to the assembler at which point, the programmer can enter AORG >200 to set the location counter.

Each line of source code is then entered in succession. Remember there is only one space between operation and operand fields. The last line of the source code consists of the END directive followed, optionally, with the start label (BG) if the programmer wants the program's starting address to be placed in the PC slot of the UNIBUG monitor. The resulting object code listing from this assembly (using an off-board printing terminal) is given in Figure 4-8.

```
LABEL   OPCODE/OPERAND      COMMENT
UB      EQU >3000           RETURN TO UNIBUG
        AORG >200           SET PROGRAM ORIGIN
BG      LWPI WS             INITIALIZE WP
        LI RO,>31E          POINTER TO BOTTOM OF FILE
        LI R2,>300          INITIALIZE MASTER POINTER
RS      MOV R2,R3           RESET INNER POINTER
        INCT R3             INCREMENT INNER POINTER
CP      C *R2,*R3+          COMPARE MASTER ITEM & INNER ITEM
*                           AND INCREMENT INNER POINTER
        JGT CK              NO SWAP IF MASTER ITEM GREATER
        JEQ CK              NO SWAP IF EQUAL
SW      MOV *R2,R1          MOVE MASTER ITEM TO R1
        MOV @-2(R3),*R2     REPLACE MASTER ITEM WITH INNER ITEM
        MOV R1,@-2(R3)      REPLACE INNER ITEM WITH MASTER ITEM
CK      C R3,RO             IS INNER POINTER JUST BEYOND LAST?
        JLT CP              *IF NOT,
        JEQ CP              *   GO FOR NEXT ONE
        INCT R2             INCREMENT MASTER POINTER
        C RO,R2             IS MASTER POINTER NOT YET LAST?
        JGT RS              IF SO, DO AGAIN
        B @UB               OTHERWISE, RETURN TO UNIBUG
WS      BSS 32              SPACE FOR WORKSPACE
        END BG
```

Figure 4-7. Replacement Sort Program Source Code

LISTING FOR REPLACEMENT SORT PROGRAM

```
OBJECT  |          SOURCE
LOC &   | LABEL   | OPCODE/OPERAND  | COMMENT
CONTENTS|         |                 |
--------|---------|-----------------|-----------------------------------
0000      UB        EQU >3000         RETURN TO UNIBUG
0000                AORG >200         SET PROGRAM ORIGIN
0200 02E0 BG        LWPI WS           INITIALIZE WP
0202R0000
0204 0200           LI R0,>31E        POINTER TO BOTTOM OF FILE
0206 031E
0208 0202           LI R2,>300        INITIALIZE MASTER POINTER
020A 0300
020C C0C2 RS        MOV R2,R3         RESET INNER POINTER
020E 05C3           INCT R3           INCREMENT INNER POINTER
0210 8CD2 CP        C *R2,*R3+        COMPARE MASTER ITEM & INNER ITEM
0212      *                           AND INCREMENT INNER POINTER
0212R15FF           JGT CK            NO SWAP IF MASTER ITEM GREATER
0214R13FF           JEQ CK            NO SWAP IF EQUAL
0216      *  SWAP ROUTINE
0216 C052 SW        MOV *R2,R1        MOVE MASTER ITEM TO R1
0218 C4A3           MOV @-2(R3),*R2   REPLACE MASTER ITEM WITH INNER ITEM
021A FFFE
021C C8C1           MOV R1,@-2(R3)    REPLACE INNER ITEM WITH MASTER ITEM
021E FFFE
0220 8003 CK        C R3,R0           IS INNER POINTER JUST BEYOND LAST?
0214*1305
0212*1506
0222 11F6           JLT CP            *IF NOT,
0224 13F5           JEQ CP            *   GO FOR NEXT ONE
0226 05C2           INCT R2           INCREMENT MASTER POINTER
0228 8080           C R0,R2           IS MASTER POINTER NOT YET LAST?
022A 15F0           JGT RS            IF SO, DO AGAIN
022C 0460           B @UB             OTHERWISE, RETURN TO UNIBUG
022E 3000
0230      WS        BSS 32            SPACE FOR WORKSPACE
0202*0230
0250                END BG   0000
```

Figure 4-8. Replacement Sort Program Listing (Source & Object)

Program Testing and Modification

To execute and test this program, it is necessary to enter the data items starting at location >0300, as indicated by the specification. Also be sure that the register workspace, the program, and the sort file do not overlap. After entering the data into the sort file, and ensuring the PC is set to the proper value, execute the program (with the E command). Check first to see if the program returns to UNIBUG. Then inspect the sort file to see if the data is actually sorted arithmetically. Note that the negative numbers will be on the bottom (higher addresses) and the positive numbers on the top (lower addresses). Also check to see that the data sorted extends strictly within the range of the sort file, not further and not less. For example, the item at location >0320 should not be included in the sort, while the one at location >031E should be included.

To correct this program or to modify it, use the assembler to re-enter the entire source code and any changes; or, if feasible, simply patch the machine code as required. For example, if it is desired to sort arithmetically such that the greater numbers are on the bottom (higher addresses) and the smaller numbers are on the top (the reverse of the way it was implemented in the program presented), then swap the numbers when the item at the "top" is greater. To implement this change, use the JLT instruction in place of the JGT instruction. The implementation of this modification is left as an exercise for the reader.

4.10 SUMMARY

In this chapter, an overview of the categories of software is presented. Emphasis is placed on the various functions of an assembler. The primary functions provided are translation of mnemonic codes, address housekeeping, generation of constants, error messages, and output control. The features and capabilities of the symbolic assembler are discussed. In particular, the use of two character labels for forward and backward references are presented. Entry, syntax, and correction of source statements are discussed along with details of the symbol table. Concerning source statement syntax, it is emphasized that

- Labels on instructions must be entered without preceding spaces
- Instructions without labels must be preceded by at least one space
- Following the opcode field, only one space is allowed prior to the operand field.

Directives are also introduced and defined.

Additional details on byte, compare, and jump instructions are covered in Instruction Subset 2. The problem of arithmetically sorting a list of numbers in a file is presented in the program example.

For those interested in a summary comparison of the advantages and disadvantages of assembly language versus high-level language, such is presented in Figure 4-9. The details presented in this figure may help the reader to gain a perspective and an understanding concerning the roles of these two levels of computer language.

4.11 EXERCISES

1. What would be the missing object code in this following program segment?

```
0200    0207        LI R7,>300
0202    0300
0204    81D8 CP     C *R8,R7
0206                JLT NX
0208    0608        DEC R8
020A                JMP CP
020C    0460 NX     B @>3000    RETURN
020E    3000
```

2. How many different labels and/or symbols are available with this assembler?

3. What result will the following directive produce in terms of object storage and resulting location counter? How else could the same result be accomplished with a directive?

Location	Source
0240	TEXT 'LAD'

4. Since this TEXT directive will not support entry of a single-quote character as part of a character string, how might one cause storage of the ASCII equivalent of a single-quote character?

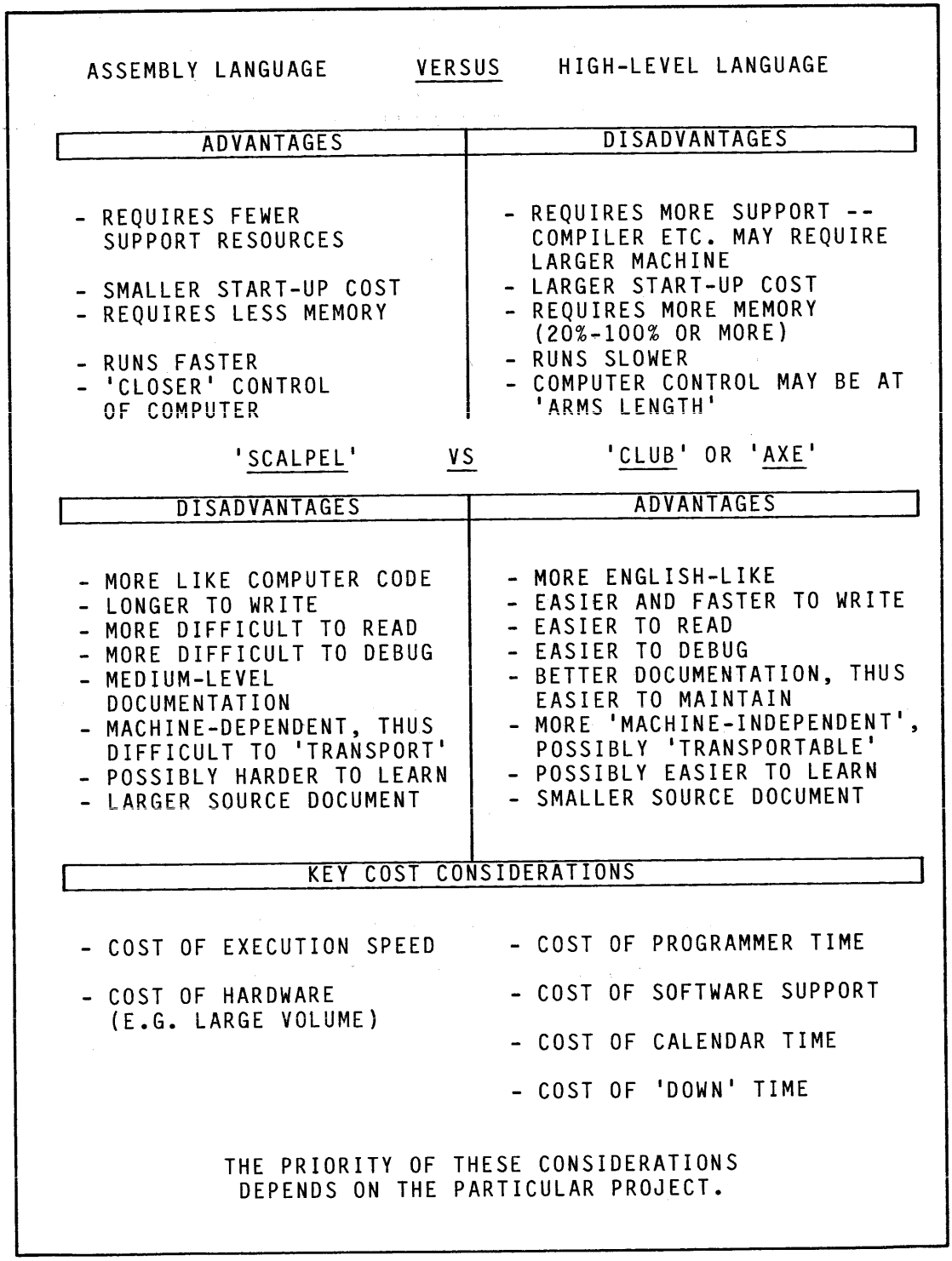


Figure 4-9. Summary Comparison of Assembly Language and High-Level Language

For Exercises 5-19, assume the following registers and memory locations contain the values shown for each exercise. Execute each of the instructions listed below. Indicate the result address and contents and the resulting status register changes.

(R0) = >0101	(222) = >2468
(R1) = >0F0F	(224) = >0101
(R2) = >0222	(226) = >4001
(R3) = >0225	(228) = >5511
(R4) = >0003	(22A) = >ABCD

		RESULT	STATUS REG. CHANGES
5.	AB R2,R0		
6.	AB *R2,R1		
7.	AB *R2,*R3		
8.	AB @>223,R0		
9.	AB @>224(R4),R1		
10.	SB R1,@>0006(R3)		
11.	SB *R3,*R2		
12.	SB @>22A,*R3+		
13.	MOVB R3,@8(R2)		
14.	MOVB R4,R4		
15.	SB *R2,*R2		
16.	CB @5(R3),*R2		
17.	C R0,@2-3(R3)		
18.	C *R3+,*R2+		
19.	CB R2,R3		

20. How big a program (in decimal words) can a user have in on-board RAM (as provided)?

21. Write the assembly language statements necessary to move the contents of memory locations >300 to >37E to the memory locations starting at >380 and then return to UNIBUG. Include AORG >200 and END directives. Use the LWPI instruction.

22. There is a byte table consisting of 50 bytes of data beginning at >3700. It is desired to copy this data to a byte table beginning at >380, with the bytes reversed (i.e., first byte last and last byte first). Write the assembly language program to do this.

4.12 LAB EXPERIMENTS

In the following experiments, it will be helpful to record on paper your entries and the corresponding displays.

1. In response to the UNIBUG prompt enter an "A" Command and 0 2 0 0 (Ret) and observe the assembler response. Enter several spaces and END (Ret). Observe the response indicating no unresolved references. Then enter a Ret and observe the return to the UNIBUG prompt.

2. Repeat #1 entering an AORG >300 (preceded by a space) while in the assembler and observe the change in the location counter.

3. Repeat #1 entering MOV R1,R0 (preceded by a space) while in the assembler and observe the acceptance of these predefined symbols and the generation of the correct object code (C001).

4. Enter the source statement A JMP A into the assembler and observe that it eventually produces the same object code as: JMP $ (10FF). Notice that A is at first considered an unresolved forward reference and then is resolved.

5. Enter a line of correct code but rather than terminating with a Ret, cancel it with a shift X. Observe the result.

6. Enter a line of code with a syntax error such as A MOX and observe the result.

7. Enter the source line: WS BSS 32 and observe the change in the Location Counter display. Then enter XY BSS -2 and observe the result.

8. Enter the source line: Z DATA 0 and observe the result. Then enter

```
IB  DATA 0,1,-1,2,-2
```

and observe the result. (Note each object display waits for character input before continuing.)

9. Enter these two source lines and observe the result.

```
CD  EQU >A55A
NX  LI R2,CD
```

10. Enter the source line: NM TEXT 'YOUR NAME' and observe the result. (One will be able to cause display of this text when introduced to instructions presented in later chapters.)

11. Assemble and execute the program loop indicated under the JLT instruction and presented here.

```
      AORG >200
WS    BSS 32
ST    LWPI WS
      LI R1,>FF01
LP    INC R1
      JLT LP
      B @>3000          RETURN TO UNIBUG
      END ST
```

Execute and inspect memory location >202 (i.e., R1).

12. Assemble and execute the revised FTN1 program specified in Section 4.8.

13. Assemble, execute, test, and correct, as necessary the replacement sort program indicated in Section 4.9 (Figure 4-7).

14. Modify the replacement sort program so that the smallest number ends up at location >300 and the largest at location >31E. (Hint: Only one instruction needs to be changed.)

15. Implement a sort program using the bubble sort algorithm. The bubble sort algorithm is one in which the program makes a pass on the entire file, comparing adjacent pairs and swapping them as necessary until a full pass is completed with no swaps. In other words, compare the first item with the second and swap, as necessary, and the second item with the third and swap, as necessary,etc. Continue until the pass is completed and repeat the process until no swaps are required for an entire pass.

CHAPTER 5

MEMORY SYSTEMS

5.1 INTRODUCTION

This chapter explains the various types of computer memory, defines the capabilities and limitations of each, and highlights several techniques which must be understood for effective memory usage. The emphasis is on specific applications of semiconductor memory components, especially in system design. The material deals with a technological area which is in constant and dynamic change and advancement; therefore, considerable effort is made to include any new technology which seems likely to affect memory usage patterns in the near future.

Chapter 1 reveals that a computer system consists of a CPU, memory, and an I/O (input/output) structure. A basic premise is that the processor operates from a program (a sequence of instructions). These instructions are executed sequentially by the processor; each instruction is completed before the next one is commenced. Except in the case of very specialized machines, the program is stored in memory.

Computer memory is also essential for the storage of data. Once data is available in memory, the CPU processes it when so instructed, and in the manner directed, by the program.

In the discussions to follow, computer memory is shown as one of two basic types: that which can be easily changed (Read-Write Memory) and that which is effectively permanent in nature (Read Only Memory).

Instruction Storage

Four basic kinds of programs exist in most computer systems: executive programs, I/O service routines, data-processing modules, and loaders.

The executive program, as the name implies, controls the operation of the entire computer, protecting against illegal operations. The executive controls operator communications with the machine, schedules various tasks, and transfers data between various subprograms.

Service routines are used to communicate with external data storage, computer terminals and other external devices. Service routines can be part of the executive, and are normally available for use by data processing modules.

Data-processing modules are programs which control individual functions of the computer system. With each data-processing module change, the computer performs a different task. In general, the new data-processing module operates under the same executive and calls on the same service routines. If the new tasks require different external machinery than the first task, new service routines may be needed.

The entire executive, all service routines, and data-processing modules can be stored in permanent memory. This type of storage is Read Only Memory, or ROM. When the processor system is powered-up, the programs already exist in memory and processing can begin immediately. In another case, all programs might routinely reside in nonpermanent or read/write memory. In such a case, a small permanent memory holds a special program called a bootstrap loader. The bootstrap loader supervises loading of the executive and other programs. Often a bootstrap loader is used to read into memory a more sophisticated loader program.

Work Areas

Another use for memory is that of a work area. A computer's work area acts like everything from scratch paper to a diary. One section of memory might be allocated for data or input buffers--a place to temporarily store incoming data to be processed. Similarly, output buffers hold data and/or operating commands to be sent to peripheral devices being controlled by the computer. Since data is written to work areas, these must be implemented as read-write memory.

Data Buffers. Consider the process of manipulating raw data. Data is taken from the input buffer, used in some form of computation, and the result transferred to an output buffer. In some cases, where computation is minimal and the incoming data rate is not too fast, the data is manipulated (perhaps revised or condensed) as it comes in, then stored in a large buffer. After being further processed, data is output in a continuous string as a block data transfer.

Data Manipulation. For review, consider this scenario of memory usage. The executive program supervises the data-processing module as it uses service routines to control external peripherals and to bring in data which is stored in the input buffer. Data is manipulated by service routines called from the executive or by special routines which are part of the data-processing module. Other service routines output the data to a peripheral device or to external

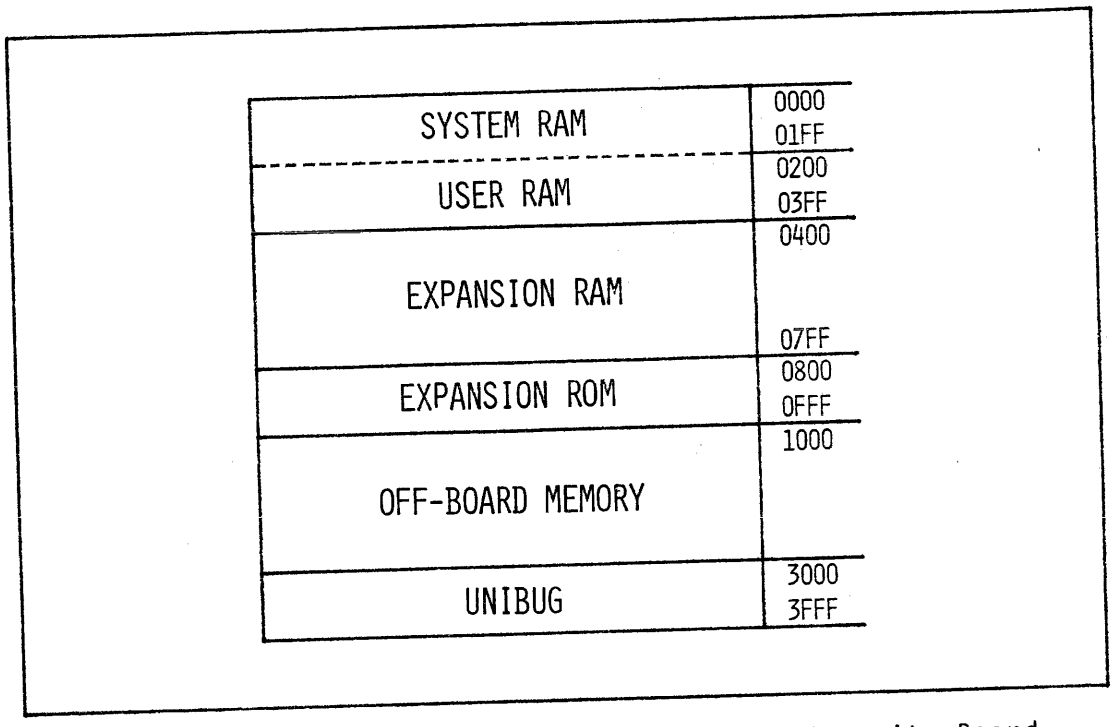


Figure 5-1. Memory Map of the TM 990/189 University Board

data storage from the output buffer. Decisions made by routines from the data-processing module will change the operation of external machinery, using a service routine to control the external equipment.

The computer can also sort data from several sources and merge those source files into one larger file. Another possible task is an assembler program to translate assembly language source code into machine code. Both of these tasks require a very large work area with relatively small input and output buffers.

Memory Map. No matter which of many possible memory arrangements is used, the final system memory design can be represented by a memory map. The memory map is a record of the starting and ending addresses of each block or type of memory used within a computer. Such a map also defines all blocks of unassigned memory locations. To a certain extent, the boundaries of various memory blocks will depend upon the memory organization of individual memory components, as discussed later. Figure 5-1 shows the memory map for the University Board.

5.2 MEMORY CHARACTERISTICS

Data Access

Memory is accessed in one of two ways: serially or randomly. Serial memory (e.g., paper tape, cassette tape, and reel-to-reel

magnetic tape) is arranged so that, as the tape passes through the record or playback mechanism, one bit or one character at a time is accessible (serial access). Physically, the tape medium must be examined along its entire length to find a particular block of characters.

In random-access memory, any unit of stored data can be directly accessed in random fashion. It generally uses a two-coordinate axis system for addressing as shown in Figure 5-2. Any byte or word in the entire memory can be directly addressed. Figure 5-2 depicts a matrix with dimensions 16 units x 4 units. Using the convention that locations vertically along the side represent the most-significant digits of the address, then cell X is at address 0006_{16} and cell Y is a $003C_{16}$. If this matrix were expanded to 64K locations (16-bit address bus) the vertical scale would identify 4096 rows (from 000_{16} through FFF_{16}). Just as Figure 5-2 shows the lowest four rows of memory, Figure 5-3 shows the highest four rows. Now, cell X is at $FFC6_{16}$ and cell Y is at $FFFC_{16}$.

In review, computer data can be stored and retrieved either serially or randomly. The on-board memory of a processor system is random access in nature. Common usage currently refers to the read-write memory (nonpermanent) as RAM (Random Access Memory). Since the RAM loses data when it is powered down, this kind of memory is sometimes referred to as volatile. The more permanent (nonvolatile) memory also allows random access addressing, but is referred to as ROM.

Memory Types

Memories can be divided into three main categories:

- Volatile
- Nonvolatile
- Mass storage.

Each of these is discussed briefly and subdivided further in the following paragraphs.

Volatile Memories. Volatile memories (except for charge-coupled devices discussed below) are subdivided into static RAM, dynamic RAM, and edge-activated static memory. The distinguishing factor is the type of memory cell used to store each data bit. The basic unit of storage for computer memory is the bit, and early memory IC's were organized one bit wide, i.e., 256x1, 1Kx1, etc. In Figure 5-4 a small section of random-access memory is expanded from the simple X-Y matrix to show that, if the memory location stores a byte (8 bits), this would require eight integrated-circuit packages (with bits located on the Z axis) to store whatever number of bytes are represented by the specified number of address locations. More recent IC memory designs feature four-wide and eight-wide outputs,

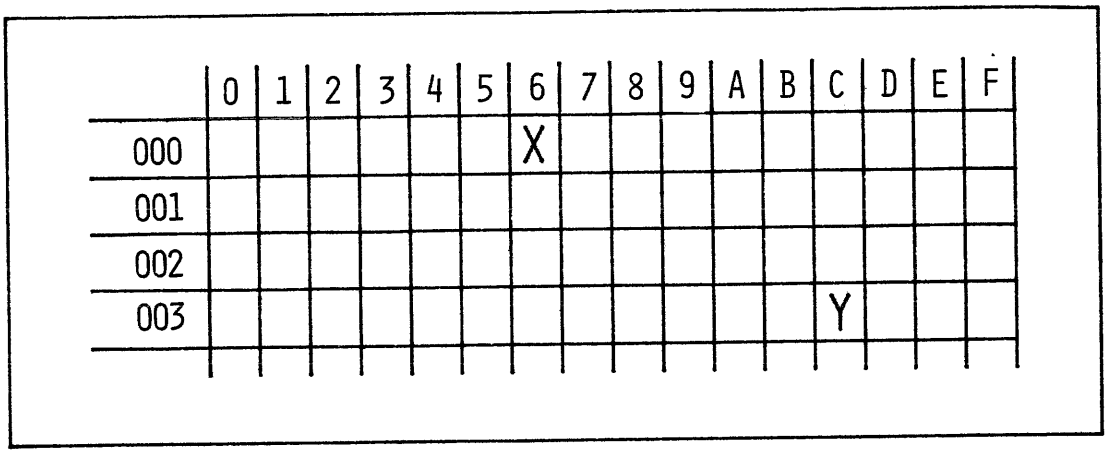


Figure 5-2. Memory Matrix --- "Low" Memory

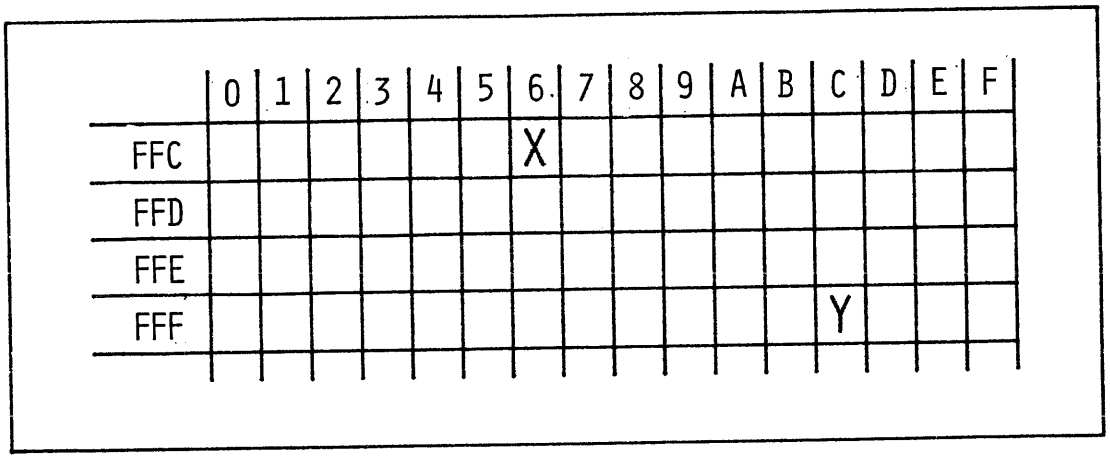


Figure 5-3. Memory Matrix --- "High" Memory

allowing small memories to be designed economically. This storage model is further refined later using various examples.

A static memory cell (Figure 5-5) is basically a conventional type of set-reset flip-flop. Although a number of different designs are used, the important idea is that, once stored, a data bit remains unchanged until the IC is powered-down or the cell is changed by writing a new bit into the cell.

On the other hand, the dynamic cell (Figure 5-6) "remembers" a logic ONE as a charge stored on a capacitor. Various factors including circuit leakage on the IC chip and the temperature of the IC cause this stored charge to leak away. To compensate for this leakage, it is necessary periodically to rewrite or "refresh" each logic ONE charge. Typically, this time period does not exceed two milliseconds. Depending upon the memory-controller architecture, it often is necessary to interrupt processor operation for brief

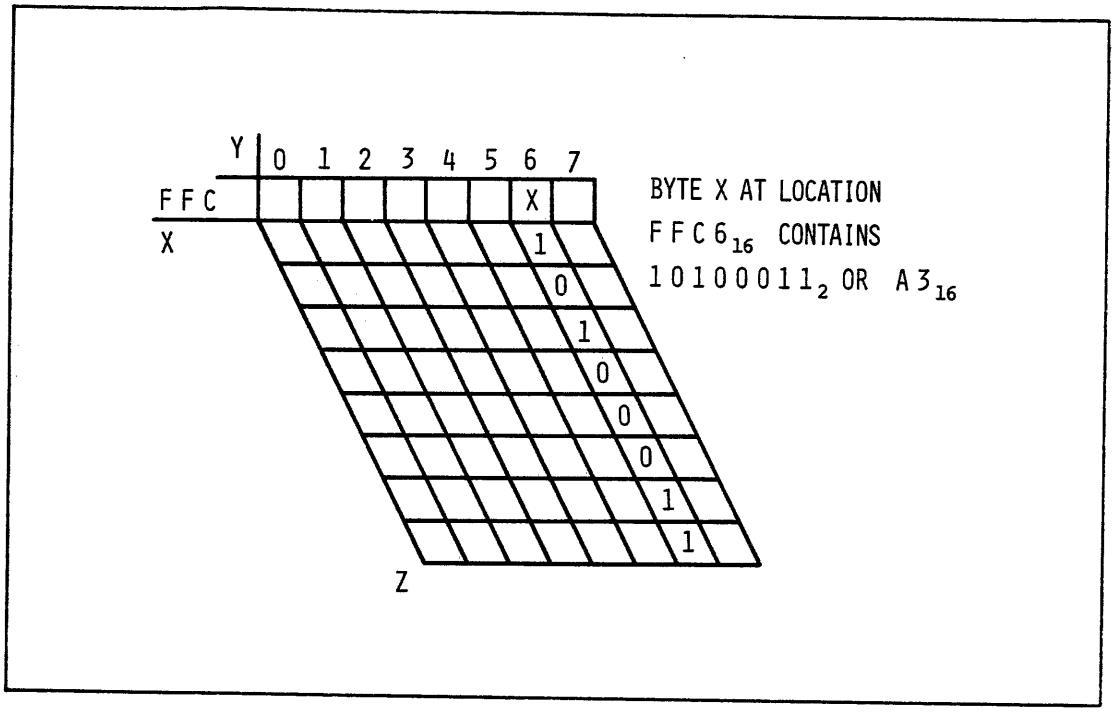


Figure 5-4. Expansion of Memory Location Showing Bit Storage

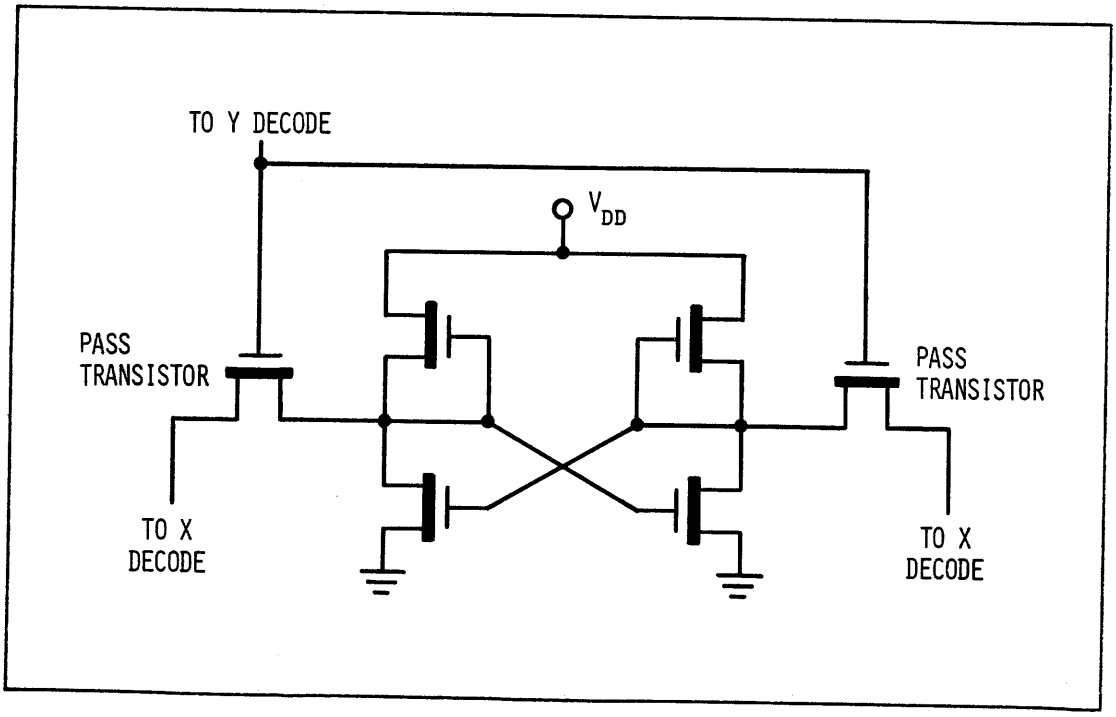


Figure 5-5. Example of Static Memory Cell

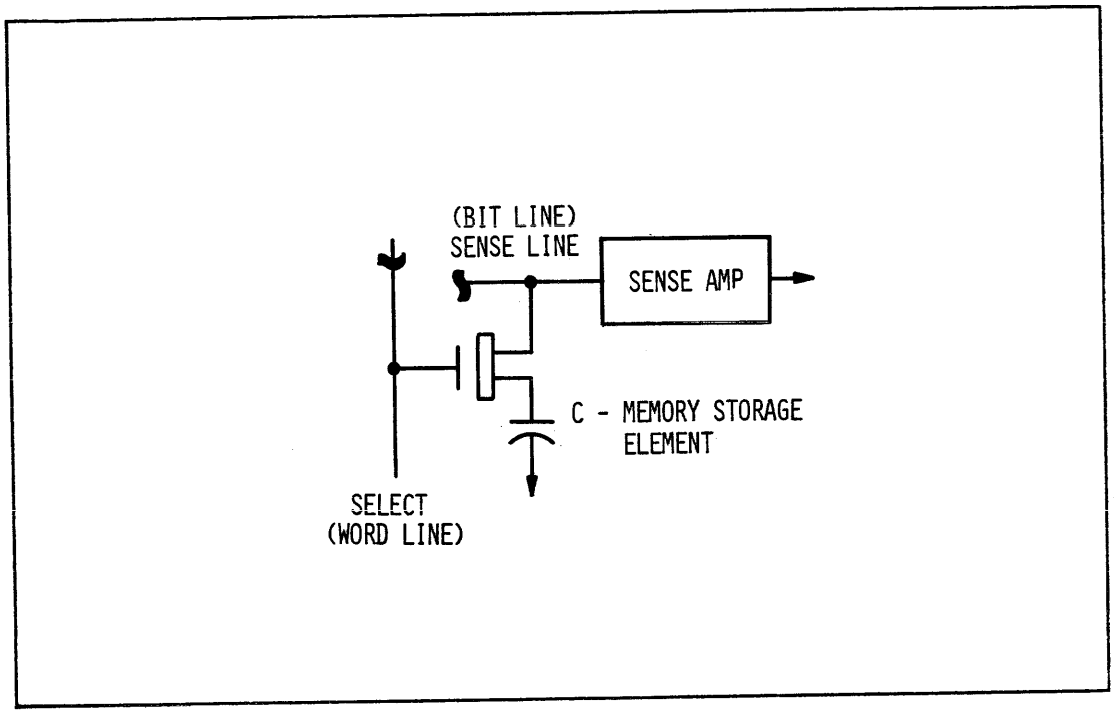


Figure 5-6. Example of Dynamic Memory Cell

periods to accomplish the refresh cycle. If the processor is not stopped, the refresh operation may require use of memory address lines at the same time the processor is trying to access memory. The resulting "tug of war" between refresh address drivers and processor address drivers is referred to as bus contention. Some recently introduced microprocessors and microcomputers arrange their architecture so that refresh cycles are "transparent" to (do not interfere with) processor operation. Currently, this feature is the exception rather than the rule.

Although the dynamic device is more troublesome, it is used for these two reasons:

- Lower power consumption
- Lower device cost.

The area of silicon required to implement a dynamic cell is considerably smaller than the area required for a static cell. Further, the power consumed by cells of equivalent access time (time required to retrieve stored data or change the data) is lower for dynamic cells than for static cells. Support circuitry (address decoders, data amplifiers, and output amplifiers) for the dynamic memory, illustrated in Figure 5-7, also uses lower power and requires a smaller area of silicon. IC cost is directly proportional to the amount of silicon used, and reliability is enhanced by cooler device operation resulting from lower power consumption. Thus, larger memory

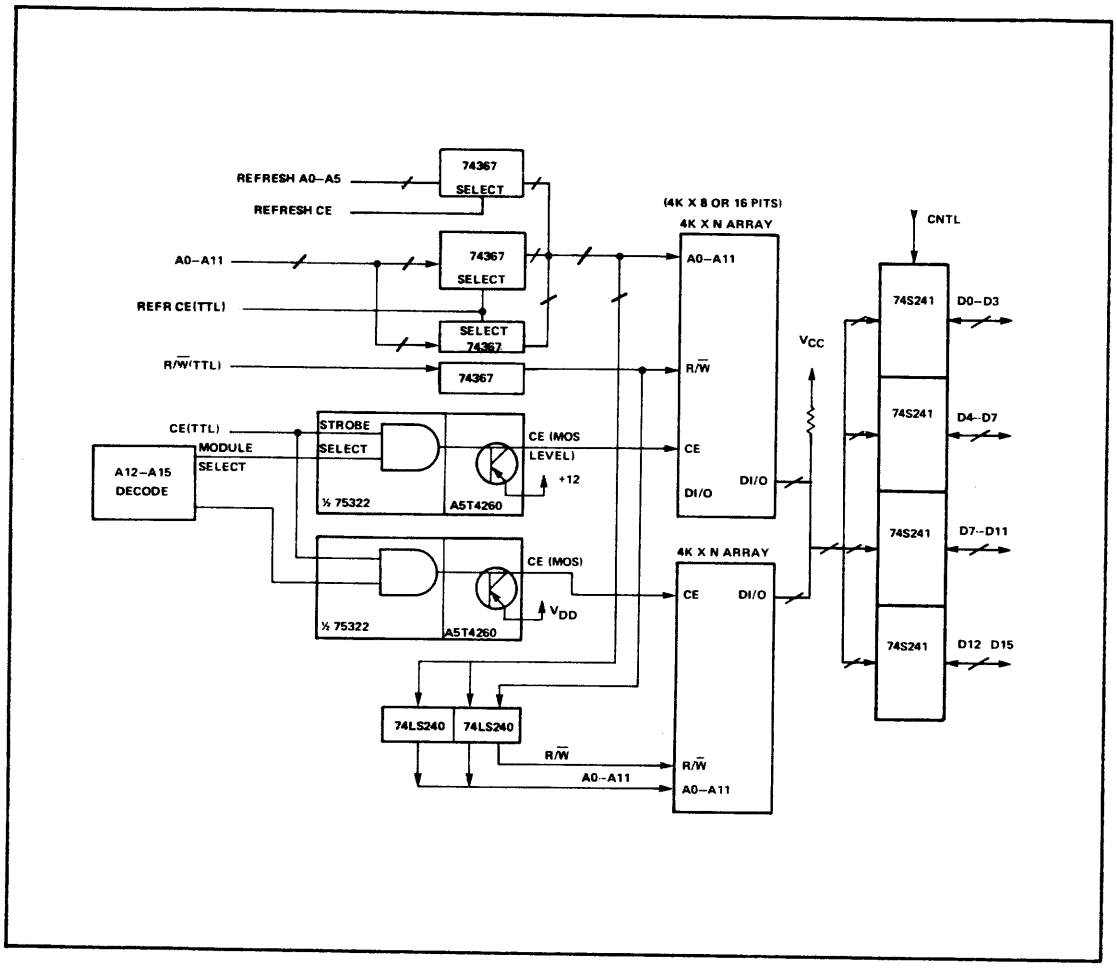


Figure 5-7. Example of Memory Support Circuits

arrays can be constructed less expensively using dynamic memory, even though the memory controller is more complex and is an additional expense for the dynamic memory.

Edge-activated memory is a recent development which combines the best features of static and dynamic memories. Static memory cells are used, with power dissipation about 20 microwatts per bit. The real saving in power comes from using dynamic support circuitry which effectively "powers down" automatically after a memory read or write operation. The operational difference between edge-activated devices and standard static RAM is essentially zero, so the internal differences are of more concern to the computer designer than to the user. If the memory control section of a computer is properly designed, the operational peculiarities of dynamic RAM are also transparent to the user. Only if the processor operation is suspended during refresh will dynamic memories impact the system's throughput, and then normally only if the computer is managing a real-time, high-speed data transfer or control operation.

Nonvolatile Memory. Although nonvolatile memory covers several categories of computer data storage, only semiconductor ROM is discussed at this point. Reviewing the earlier discussion in Chapter 1, the general ROM family is subdivided thusly:

- ROM (mask programmed Read Only Memory)
- PROM (Programmable Read Only Memory)
- EPROM (Eraseable PROM)
- EAROM (Electrically Alterable ROM).

The generic name ROM is often applied to all categories of nonvolatile solid-state memory.

Conventional ROM's are produced with the program information as a part of the production process of the integrated circuit. Integrated circuits are constructed by many successive operations involving photographic mask patterns and diffusion operations. Exact details of the process are beyond the scope of this discussion, but the process can be compared to production of a multicolored document on a printing press. Each pass through the printing press applies one color, using a pattern corresponding to the shape and location of that color on the finished page. In similar fashion, integrated-circuit components are constructed by "printing" successive layers of material corresponding to various circuit components. The pattern of 1's and 0's which define the machine code in a ROM is determined by the final printed layer ... actually the last layer of metalization in the production process.

Since the design of the 1/O pattern for a ROM is unique for each stored program and is irreversible after completion of the ROM, this type of nonvolatile memory is expensive unless a very large number of devices are produced from the same pattern. A typical high-density ROM contains from 8196 bits (1024 8-bit words) to 65,536 bits (8196 8-bit words). The creation of the mask (or pattern) is time-consuming (therefore expensive) and must be perfect. Since the creation of all the other masks associated with a particular ROM type are common, these other masks represent a one-time cost. Thus, the expense is amortized by spreading the cost over hundreds of thousands of units with different ROM patterns. The unique program mask similarly must also be amortized over a large number of units. Consequently, ROM's normally are practical only in applications requiring a large number of copies of a well-proven program. The time required to start production of a particular mask-programmed ROM can be eight weeks or longer, which normally is acceptable only in cases of continuing production of a proven device.

The production time lag and expense associated with ROM's inspired the development of PROM's. To understand the design of a PROM, consider first the ROM memory cell shown in Figure 5-8. Any particular memory address is decoded to the point where it represents a single data line which is then, say, driven high. Line No. 1 in Figure 5-8 behaves as if it has diodes connecting itself to output lines B and D. Therefore, when line No. 1 goes high (or is active), lines B and D follow. If D is the MSB and A the LSB,

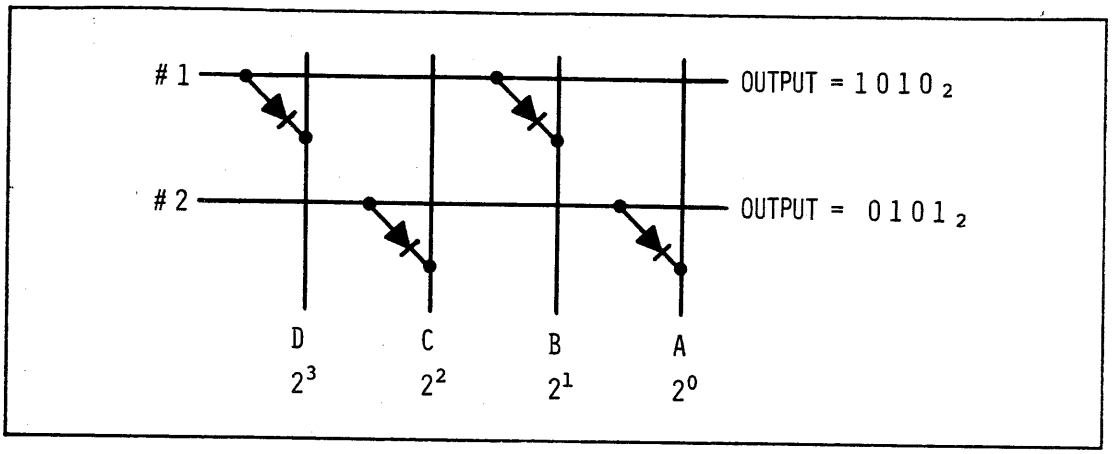


Figure 5-8. Example of Diode ROM

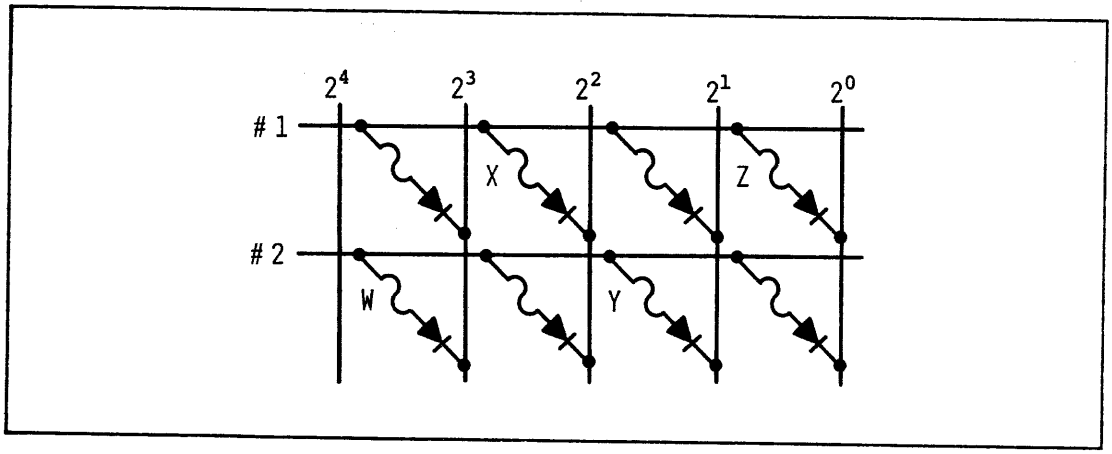


Figure 5-9. A Fusible Link Diode ROM

the output is 1010_2 or A_{16}. Similarly, activating line No. 2 will produce 0101_2 or 5_{16}.

A typical unprogrammed PROM is organized as shown in Figure 5-9. Note that since all possible diodes are hooked up, the output of any address will be 1111_2. Note also that each diode has a fuse in series with it. The act of programming a PROM consists of passing a high current through selected diodes and melting the fuse for each ZERO bit required. Therefore, passing a high current through all diodes except W, X, Y, and Z creates a section of ROM programmed like that in Figure 5-8. Just as with a standard ROM the program "burned" into a PROM should be considered permanent. Any programming error generally spoils the PROM.

IC's are designed around special characteristics of the material used in the IC's manufacture. MOS (Metal-Oxide-Semiconductor) devices are constructed in such a fashion that a highly insulated region

is created. If this region is the gate of an MOS transistor (Figure 5-10) and a charge is injected into the gate region, the transistor will be "on." If the voltage is then measured at the drain lead of the transistor (point A of Figure 5-10) the voltage is low, and that memory cell contains a logic ZERO. Another special property of the MOS material is that intense ultraviolet light directed against the IC's surface adds energy to the electrons trapped in the gate region causing them to leak away.

These peculiar characteristics of the MOS material are used to create another kind of ROM called EPROM. Selectively putting a high charge in the gate region programs a cell, while exposure to intense ultraviolet light de-programs or erases all cells in the EPROM at one time. Erasure then restores the original "new" condition, and allows a new or a modified program to be written or programmed into the device.

EAROM and EEPROM devices are nonvolatile read-write devices, however incongruent that may seem considering previous remarks. Again, these devices exploit special characteristics of the ROM material. Both are new technology devices expected to develop rapidly for special-purpose applications. EAROM's (Electrically Alterable ROM) use MNOS (Metal-Nitride-Oxide-Semiconductor) technology which has access times almost as fast as MOS EPROM's (500-1000 nanoseconds) but can be electrically erased in a selective manner. Erasure typically requires about 10 milliseconds per block of information, and writing new data requires about 1 millisecond per location. Such slow data-change times have caused these devices to be called "read-mostly" memory. EEPROM (Electrically Erasable PROM) devices are very similar to EPROM's in storage method and operation, except that erasure is accomplished by an electrical pulse which clears the entire component memory at one time.

In the past, almost all computer memory was nonvolatile and random access, and was known as magnetic-core memory. This technology consists of tiny ferrite toroid cores with various windings on each core. One winding allows the cell to be set to a ONE or ZERO by magnetizing the core to positive (logic ONE) magnetization or negative (logic ZERO) magnetization. Another winding drives the core with a demagnetizing pulse, which develops a pulse across a sense winding (third winding). The polarity of the sense pulse indicates whether a ZERO or ONE was stored, but the cell is erased in the process. This procedure is known as destructive read-out, and requires that the cell be rewritten in the same polarity as it was before being accessed. This seemingly cumbersome technique has undergone strong development over the years, and remains a viable memory technology for certain special applications. The major advantages of core memory is that it is nonvolatile memory with read-write random access at moderate CPU speed. Magnetic-core memory controllers are more complex than dynamic memory controllers, the memory plane is larger than semiconductor memory for a given storage capacity, data access time is longer than almost all other types of RAM, and power consumption is higher. For these reasons, core is only used where data must be retained even when line power fails.

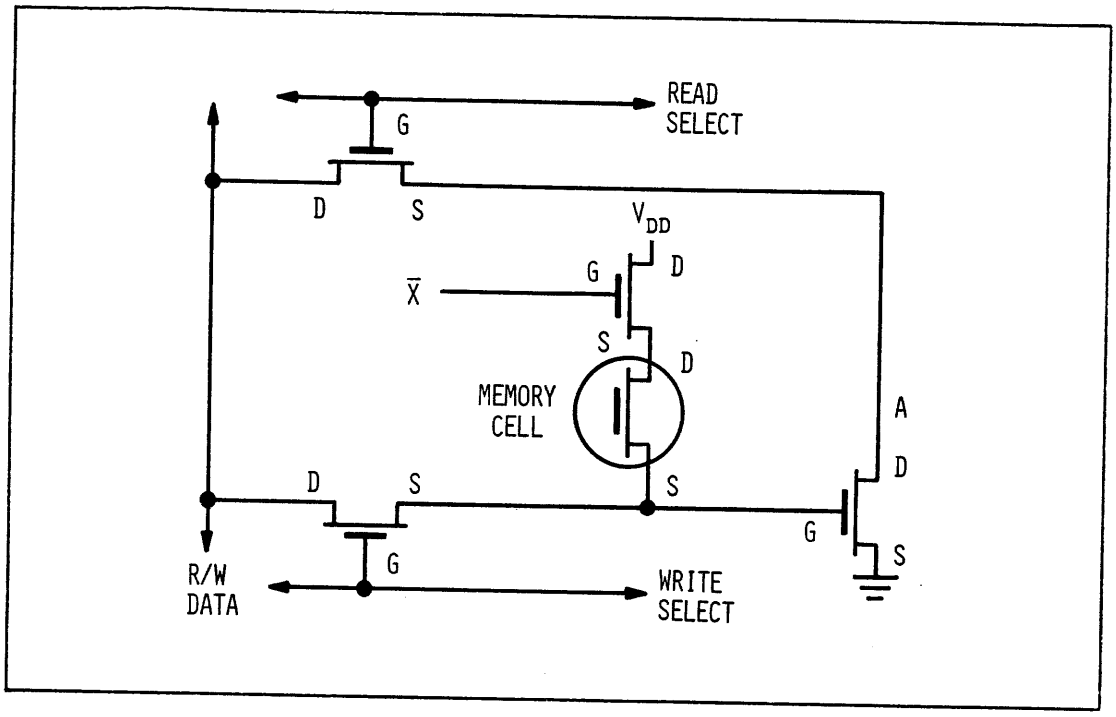


Figure 5-10. An EPROM Memory Cell

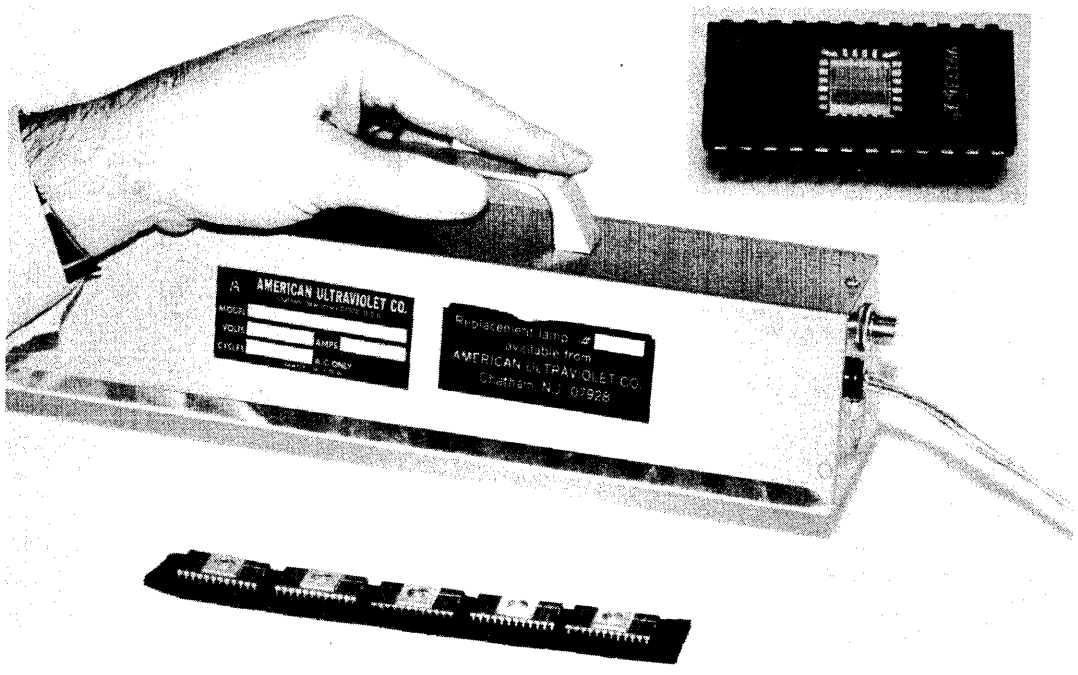


Figure 5-11. Typical EPROM TMS 2708; EPROM Erase Kit

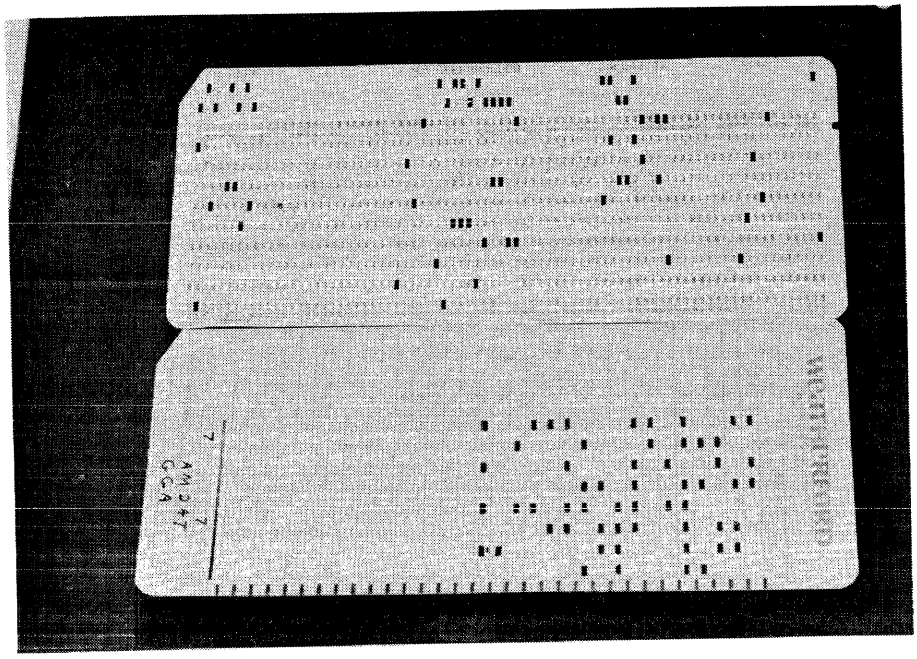


Figure 5-12. IBM-style punched card (top) stores 80 characters per card. Mark-sense card uses spots to designate ONE bits.

All the random-access technology discussed so far has been processor memory--memory which is actively used by the processor during operation and accessed by the processor without operator intervention. Typically, the maximum amount of memory "on-line" in microcomputer applications does not exceed 65,536 (2^{16}) bytes.

Mass Storage. All methods of storing large amounts of data for infrequent access--mass memory--is characterized as either sequential or random-access memory. For the most part, mass memory requires human intervention to allow the processor to access it. For example, an audio or digital cassette, diskette, or tape reel must be made ready to allow new blocks of data to be accessed by the processor.

Punched paper tape and punched cards (the familiar IBM card using Hollerith Code) were among the first mass storage media. Both are characterized by long access times and low storage density. A card stores 80 characters in a 2-of-12 code (see Figure 5-12), and data entry is accomplished by a keyboard-controlled punch. Developments in optical technology, particularly solid-state devices (LED emitters and silicon optical sensors) updated the mechanical pin arrangements which were used to read back the data from punched cards.

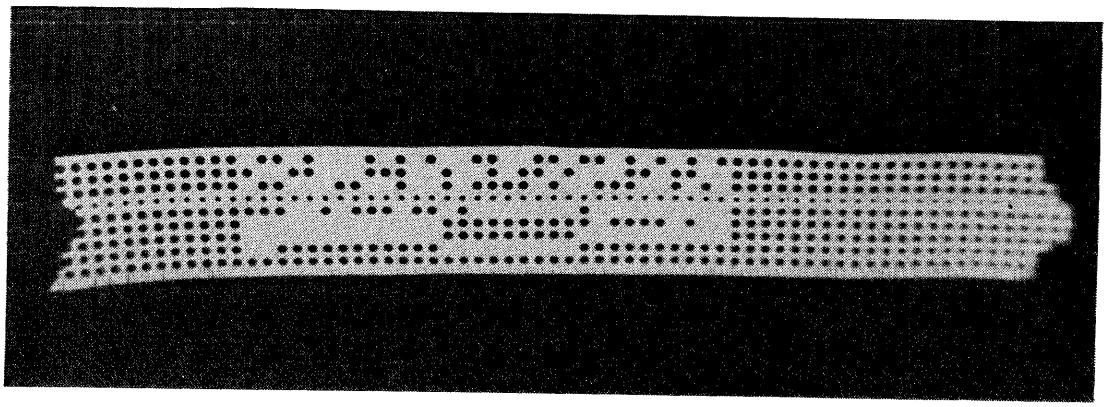


Figure 5-13. Paper Tape Format (ASCII Characters)

Further sensor refinement enabled the development of mark-sense cards. Data is entered by darkening spaces on the card. These marks are then recognized as logic ONE's just as are the holes on the punched card. The mark-sense card is usually reserved for small amounts of data, and is a common data entry method for stand-alone PROM and EPROM programming equipment.

Punched paper tape is another early mass-storage medium which, like punched cards, retains a surprisingly high use factor. Figure 5-13 shows the data pattern: an 8-bit parallel pattern represents a single character, with data in a character-by-character serial format. Either mechanical pin readers or much higher speed optical readers are used to retrieve the data.

A somewhat related technology is being developed which uses an optical bar code. This is an adaptation of the optical bar code appearing on grocery store items. The intent is to enable low-cost reproduction and dissemination of application programs. Data retrieval is accomplished via an optical wand which "reads" the bar code pattern. Software is used to decode the data. While any individual byte is directly accessible on a page, the data format is serial. Software constraints and the planned data format prohibits random loading of isolated bytes.

Magnetic tape is a long-standing mass-storage medium. The original concept used reel-to-reel magnetic tape with the data format essentially the same as on punched paper tape. Advances in technology, in magnetic material, in tape transport design (the mechanism which moves the tape), and in more sophisticated electronics, have steadily increased data-storage density while decreasing data-access time and decreasing the error rate.

Some time after the consumer audio industry introduced cassette recorders, this type of tape was applied to data recording. At present, audio cassette recorders are being used by many com-

puter hobbyists and some industrial computer users. Digital cassettes and digital recorders (much better and more expensive mechanical and electronic packages coupled with tested, high-quality digital tape) are being used throughout much of industry. The essential differences between digital and audio recording are

- Digital data on audio equipment is recorded as various tone combinations. The University Board uses one cycle of 1200 Hz as logic ZERO and two cycles of 2400 Hz as a logic ONE.
- The audio tones are recorded linearly (normal audio style recording) while digital recording utilizes saturation recording. Saturation recording requires different record/playback heads and drive electronics. Digital data is recorded as a series of flux reversals along the magnetic tape.
- If the same tape speed is used for both audio and digital recording, the digital record will have almost five times as much density--800 bits per inch of tape compared to about 170 bpi for audio recording. Also, digital tape usually runs faster than audio tape, with a further improvement in access time.

Magnetic-disk memory has long been a mass-storage medium. The basic format is a series of magnetized areas located in concentric rings or tracks on the surface of a rapidly spinning circular piece of magnetic material similar to the magnetic card mentioned earlier. The data format is bit-serial, similar to that of digital cassettes. Data is often stored in formatted fashion with the disk divided into a discrete number of wedge-shaped sectors. Each sector contains a number of data words, written around the disk in serial fashion. The effect is similar to a phonograph record, except that each data track has the same diameter. If the read/record head is not moved to a different track, the same data can be read repeatedly. A typical sector contains

- A sync field to synchronize the electronics
- An address field containing track numbers
- Head number (for units with multiple read-write heads)
- Record number and number of data bytes
- A CRC (cyclic redundancy check) word for error checking
- Address field to identify the record as either data or control
- Data record itself
- Two record gaps.

For each 128 bytes in a typical record block, all the other recorded "housekeeping" data totals an additional 59 bytes. Disks can also be unformatted, i.e., each record can be an arbitrary length. Unformatted disks have fewer identification and sync block (overhead), so total record storage is greater.

Magnetic-disk memories can be rigid, with single or multiple read-write heads, or "floppy." The floppy disk is thinner and much more flexible than a standard 45 RPM audio phonograph

record, and comes in two sizes: standard (8" diameter) or mini-floppy (5 1/4" diameter).

Although data is recorded in serial fashion on a disk, it can be randomly accessed. That is, the single-head machine can move the head to read the tracks in any order, and multiple-head drives can activate the heads in any order. Data-access time (for a particular record) is relatively long on the average, because the head must move to the correct track and the disk must rotate to the proper sector before data retrieval begins. After the record is accessed, data rates to the processor (or memory) are very high. The amount of data stored by a disk varies from about 250,000 bytes per diskette for a mini-floppy to over 20 million bytes for a rigid disk.

Charge-coupled devices (CCD) are a new type of semiconductor memory which, though storage is volatile, are planned to be low-cost bulk or mass memory. A CCD memory is basically a serial shift register made from MOS-type material. Packets of charge similar to those stored by a dynamic RAM are moved from location to location by a complex clocking scheme. The clocking serves to refresh each packet as it is moved along, similar to the required refresh in a dynamic RAM. Although each CCD device delivers data one bit at a time, parallel operation of a number of devices will produce bytes or even multi-byte words. The Texas Instruments' TMS 3064 CCD Memory is a 65,536-bit memory with an average data-access time of 400 microseconds and data rate of 5 bits every microsecond. Note that, since CCD's are dynamic devices, power must be applied continuously and the recirculate clocks must run at a rate sufficient to perform the required refresh. Even so, the power consumed by a well-designed CCD memory will be less than other volatile memory types during standby operation.

Magnetic-bubble memories (MBM) are nonvolatile, static memories using individual magnetic packets called bubbles as a storage medium. The bubbles circulate, shift-register fashion, in a thin magnetic film under the influence of a rotating magnetic field. Average data access time is 4 milliseconds with a 50-kilobit/second data rate. The targeted usage is to replace rotating magnetic media of medium capacity (mini-floppy and floppy disk) with lower price, lower power consumption, and faster data access nonmechanical memory. Nonvolatile operation of bubble memory is obtained at the expense of furnishing a permanent magnet to surround the memory with a magnetic field to hold the bubbles in place during times of power-down.

Configuration and Process Technology

A great variety of semiconductor memory components is available with memory component packaging affected significantly by the intended use. Indications of this diversity are given in the dicussion and in several figures which follow.

Configuration. Suppose that a small microcomputer-based controller needs a control program no greater than 1024 bytes of ROM and less than 128 bytes of RAM for workspace. A common memory component size is 1024 bits, organized as 1024 locations of one bit each. If 1K x 1 RAM devices were all that were available, eight such IC's would have to be used, so that 8-bit words could be formed. Total storage would then be 1024 bytes--eight times as much as is needed. Fortunately, a 128 x 8 memory device is available--thus saving greatly on power, space on PC boards and cost. For similar reasons, memory IC's are provided in a wide array of sizes as discussed below.

Process Technology. The semiconductor process technology used to build a memory component has a large effect on the device's cost and speed. The broad range of performance shown in Figure 5-14 summarizes memory performance (access time or speed) as a function of process technology (as of mid-1978). Similarly, memory organization versus process technology is summarized in Figure 5-15.

A number of terms used in these two figures are defined below.

- STTL - (Schottky TTL). Standard TTL (Transistor-Transistor Logic) was the first really popular digital logic technology. Typical gate circuits use only transistors, resistors, and diodes, with the transistors driven into saturation. A saturated transistor has a slower turn-off than an unsaturated one. Schottky TTL uses a special diode clamp (Schottky process) to intercept the excess drive, which prevents saturation and gains a 4:1 decrease in propagation delay.

- ECL (Emitter Coupled Logic). This type of device uses linear amplifiers for logic. This gains some speed over STTL at the expense of very high power consumption.

- MOS. Four kinds of MOS (Metal Oxide Semiconductor) logic appear in the tables: PMOS, NMOS, CMOS, and SOS/CMOS. These logic families are P-channel MOS, N-channel MOS, and Complementary MOS. An MOS transistor uses either N-type silicon with P-type emitters (PMOS) or P-type silicon with N-type emitter (NMOS), thus creating operation akin to PNP and NPN transistors respectively. Basically, PMOS and NMOS devices use less power than bipolar (TTL) devices, but more power than CMOS devices which use P-channel and N-channel transistors in series. In CMOS, both transistors are off except during switching, so CMOS draws effectively zero power except during switching. SOS means Silicon On Sapphire, and CMOS circuits built on a sapphire substrate instead of silicon gain almost an order of magnitude in speed over standard CMOS.

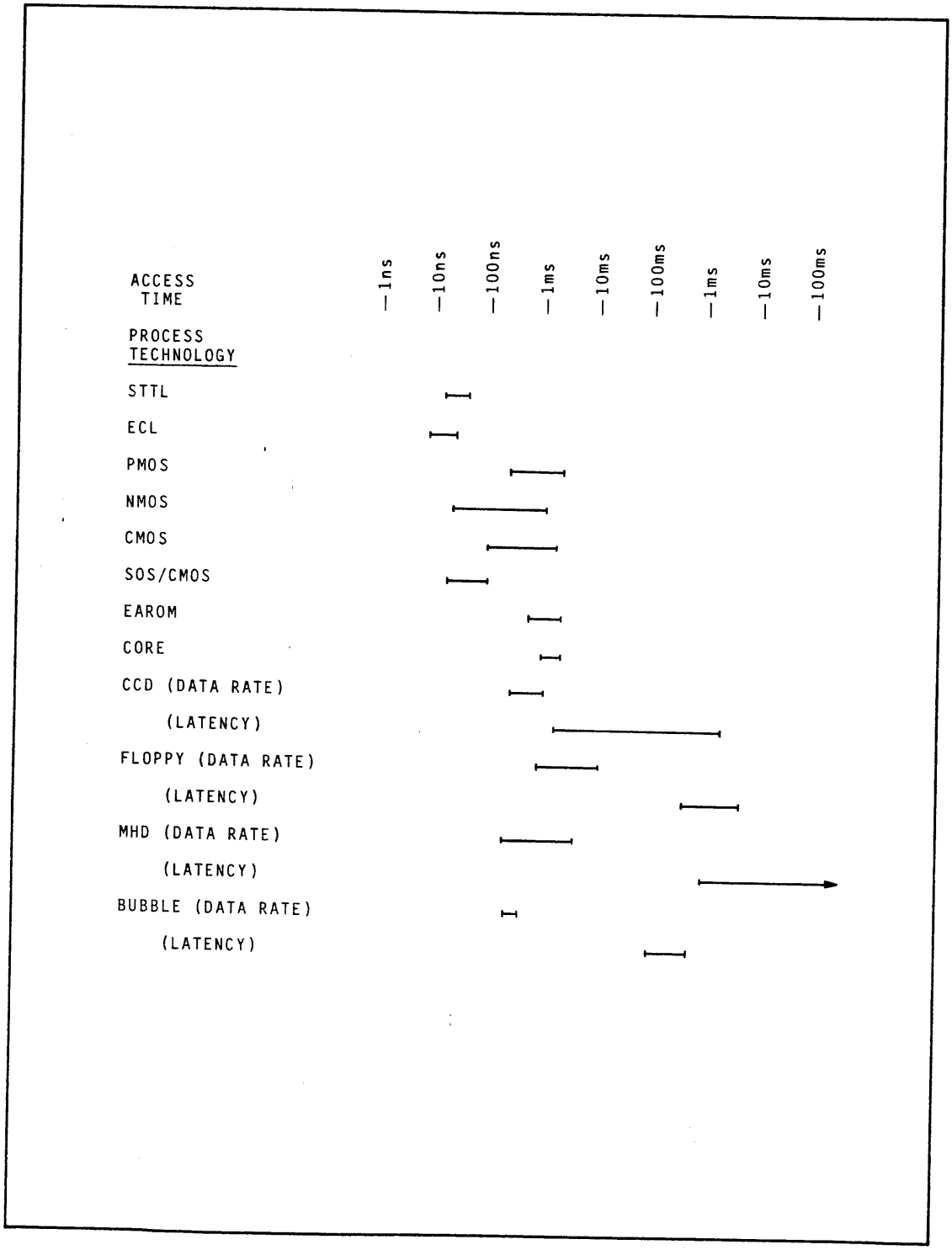


Figure 5-14. Relative Memory Performance by Process Technology

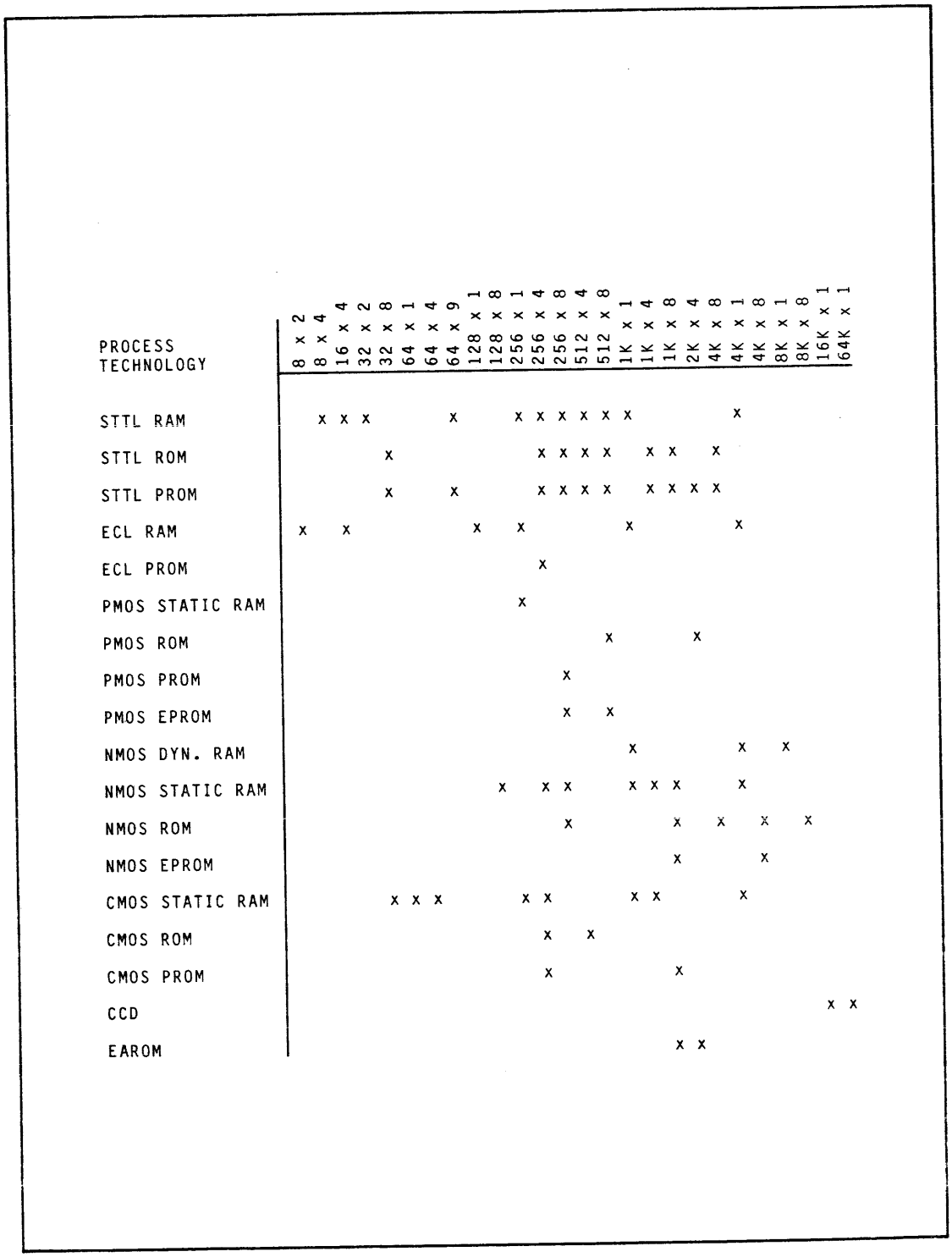


Figure 5-15. Memory Organization versus Process Technology

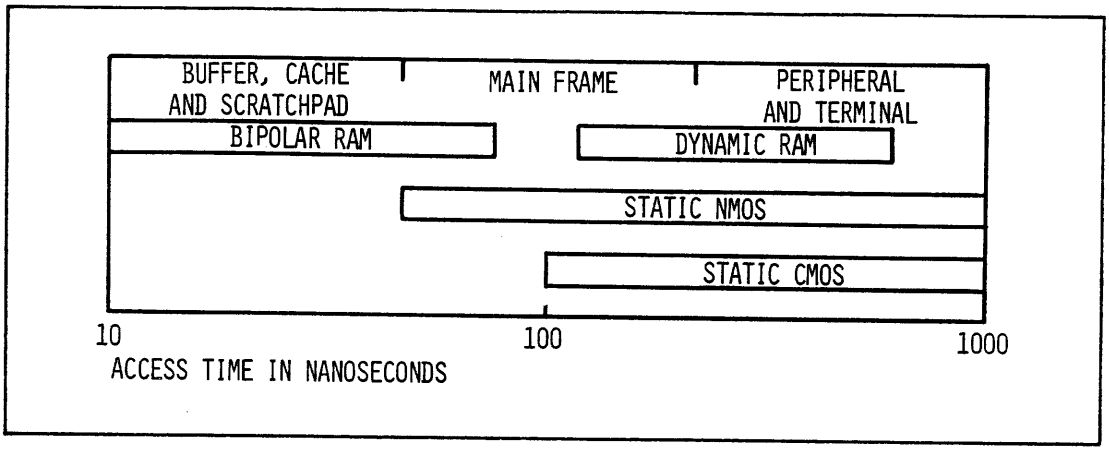


Figure 5-16. Uses of RAM's versus RAM Access Time

Figure 5-14 also includes CCD's, bubble memories, and disk systems. Note that two categories of time or speed are shown for the latter systems--data rate and latency. Latency is the time required for data to appear and is usually specified as the mean between the shortest and longest times possible on a given system. No tape systems are shown, since latency would be dependent upon the size of the reel of tape and how fast the tape moves, along with numerous other factors.

Applications

Figure 5-16 shows various uses for RAM, according to access time. A number of new terms are used in this figure and are defined below.

- Scratch pad RAM--a small amount of very fast memory used to hold intermediate computation results, thus saving time needed to fetch frequently used data.
- Cache high-speed buffer--very fast memory (larger than Scratch Pad) used to hold a program segment during execution of that part of the program.
- Mainframe RAM--resident within the computer, but loaded from mass storage.
- Peripheral and terminal memory--that which is located in terminals and other "smart" peripheral devices, where lower power and lower speed devices serve adequately.
- Intermediate working storage--slower serial memory such as CCD's or bubble memory. High density, low power storage is used to accumulate considerable data before transmittal to mass storage. Mass storage is used for long-term offline storage, such as data logging and data acquisition records.

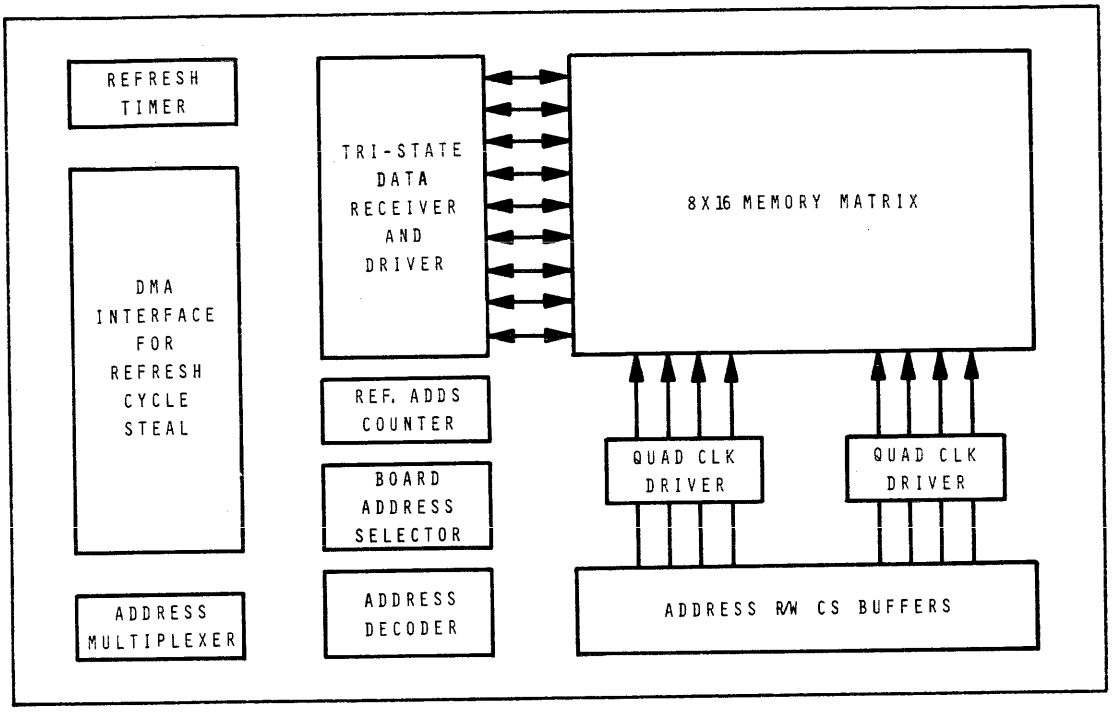


Figure 5-17. Typical Memory System Layout

5.3 MEMORY SYSTEMS

A memory system consists of much more than just memory IC's. For example, a large memory would require buffers to avoid overloading the CPU address and data lines. Also, the address lines must be decoded to uniquely enable individual blocks of memory. Some elements of a memory system are illustrated on Sheets 2, 3, 4, and 11 of the TM 990/189 Electrical Schematic Diagram.

The following paragraphs introduce material on the component, interface, control, and timing requirements for memory systems.

Components

If a memory is designed with address decoding, input/output buffers, timing, and control, the result is a simple board such as the University Board (TM 990/189). All the necessary control signals are furnished by the microprocessor and its associated circuitry. Add error-detection and correction circuitry, physical chassis, and cooling mechanism, and the machine begins to resemble a minicomputer (see Figure 5-17). Timing and control for static memories require fairly limited circuitry, whereas extensive additions are required to handle refresh of dynamic memory. A small static memory plane may fit on a single, convection-cooled PC board which plugs into a rack or stands alone. Input and output buffering are handled

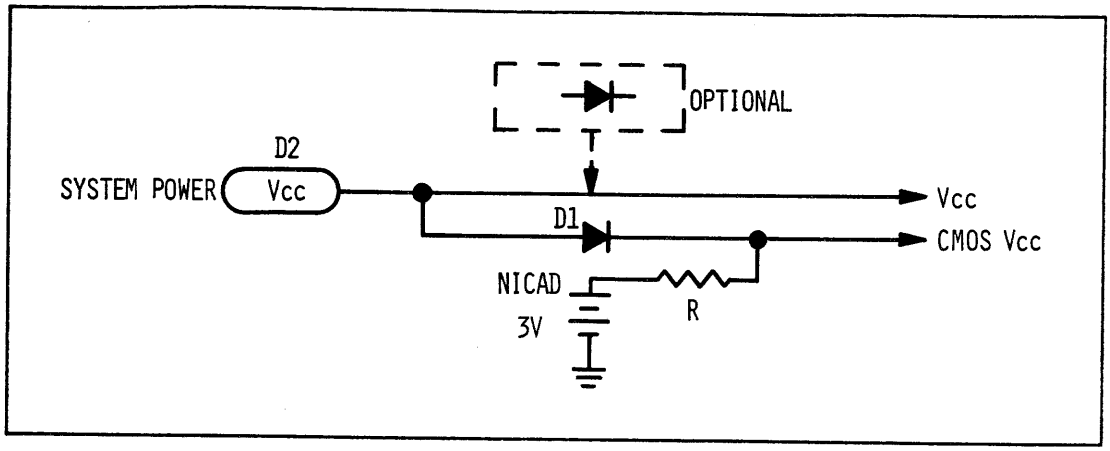


Figure 5-18. Basic Battery Backup System

by the memory IC's and processor lines, and address decoding requirements are generally minimal.

With respect to power supply requirements the trend is downward as the new 4K and 16K memory chips become available. New manufacturing processes result in devices which use much less power per bit stored. For example, two 1K x 4 RAM's will typically use one-third as much power as the eight 1K x 1 RAM's required to give the same data storage.

Computers which must keep running during a power failure obviously require some sort of fail-safe uninterruptable power systems to carry through a blackout.

The basic principle of a battery backup system is shown in Figure 5-18, where diode D1 is used to automatically switch in the battery if Vcc fails. Diode D2 may be used to equalize the voltage applied to the memory and the control circuitry. Such a simple system is really adequate only for CMOS memory, but relatively little more is required to furnish battery back-up for other static memories. If dynamic memory is used, the battery must also power the refresh circuitry, requiring a much greater energy reserve.

Control

Other action is required of a memory controller besides refresh. Input enable signals are either furnished by the processor or decoded from processor signals. A "wait-state" control must be used if the memory IC's require longer access time than allowed by the processor's operating speed. The wait signal is used to force the processor to wait on the memory until data is ready. On the TMS 9980A, this signal is called READY. Also, some types of memory IC's have a standby provision. On large memory banks, only a few memory IC's are accessed during each memory fetch. A standby control circuit

can keep all memory IC's in standby except when being accessed. This saves up to 90 percent of the power which would otherwise be used by the memory. Heat is reduced in the memory area and reliability and component lifetimes are greatly improved.

Interface

The TM 990/189 is a good illustration of memory interfacing principles. The memory space is limited to 16K bytes divided into RAM, ROM, and undefined areas which can be assigned to either function as indicated previously in Figure 5-1. For the discussion to follow, reference is made to specific sheets of the schematic diagram for the TM 990/189, as found in Appendix A of the TM 990/189 Microcomputer User's Guide. Sheet 4 shows the addressing and data lines for both system ROM and expansion ROM, while Sheet 3 shows the same connections for normal RAM and expansion RAM. Also on Sheets 3 and 4 are some auxiliary decoding circuits to control memory read and write operations. Sheet 2 shows the TMS 9980A processor with address lines (A0 through A13) and data lines (D0 through D7). Note that a signal name or signature followed by a minus sign (e.g., ENA-) is true or active when low (logic ZERO).

The two sections of U34 (see Sheet 2) sample the four most-significant address lines and decode the memory space into 16 1K blocks. The lower half of U34 decodes A0 and A1 to divide the memory space into four 4K blocks. LOMEMENA- is true for all addresses between 0000_{16} and $0FFF_{16}$, while HIMEMENA- is true for addresses between 3000_{16} and $3FFF_{16}$. Another block (addresses 1000_{16} through $2FFF_{16}$) is reserved for external (off-board) memory expansion. Similarly, the other half of U34 samples A2 and A3 to divide each 4K block into separate 1K blocks. Note that the RAM memory IC's are 1K x 4, so that each has ten address lines (A4 through A13). Portions of U14 and U15 (Sheet 2) derive DRE- by performing a logical OR between DBIN- and WE-. (Refer to the TMS 9980A/TMS 9981 Microprocessor Data Manual for an explanation of signals derived from the processor.) On Sheet 3, DRE- is ANDed with Block 0- and LOMEMENA- to produce RAMCE-; this signal selects memory IC's U20 and U22 when the processor must read or write to addresses 0000_{16} through $03FF_{16}$. Similarly, U21 and U23 (if installed) are selected when the processor needs memory addresses between 0400_{16} and $07FF_{16}$. Similar arrangements occur for EPROM's U32 and U33, except that U33 is a 4K x 8 block of memory, and U32 can be either a 1K x 8 or 2K x 8 block of memory, depending upon whether the jumper between E49 and E50 (Sheet 4) is in place.

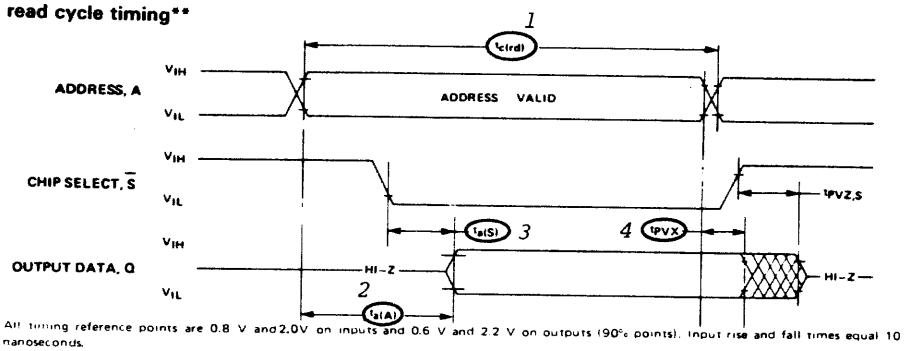
Note that the data lines for all the on-board memory devices are in parallel, so that the chip select signals (RAMCE-, EXPRAME-, ROMCE- and EPROMCE-) select only one block of memory at one time. This prevents bus contention as described earlier. If more memory were installed off-board, the drive capacity of the processor output lines would be exceeded. Therefore, Sheet 11 shows the circuitry needed to buffer (i.e., increase the drive capability of) the processor data and address lines.

TMS 4014, JL, NL **1024 WORD BY 4-BIT STATIC RAM**

	PARAMETER		MIN	MAX	UNIT
1	$t_{c(rd)}$	Read cycle time	450		ns
1	$t_{c(wr)}$	Write cycle time	450		ns
	$t_{w(W)}$	Write pulse width	200		ns
5	$t_{su(A)}$	Address set up time	0		ns
7	$t_{su(S)}$	Chip select set up time	200		ns
8	$t_{su(D)}$	Data set up time	200		ns
9	$t_{h(D)}$	Data hold time	0		ns
6	$t_{h(A)}$	Address hold time	20		ns
	$t_{T(A)}$	Address transition time	5	200	ns

	PARAMETER		MIN	NOM	MAX	UNIT
2	$t_{a(A)}$	Access time from address			450	ns
3	$t_{a(S)}$	Access time from chip select (or output enable) low			120	ns
4	$t_{a(W)}$	Access time from write enable high			120	ns
	t_{PVX}	Output data valid after address change	10			ns
	$t_{PVZ,S}$	Output disable time after chip select (or output enable) high			100	ns
	$t_{PVZ,W}$	Output disable time after write enable high			100	ns

read cycle timing**



All timing reference points are 0.8 V and 2.0V on inputs and 0.6 V and 2.2 V on outputs (90% points). Input rise and fall times equal 10 nanoseconds.

**Write enable is high for a read cycle.

early write cycle timing

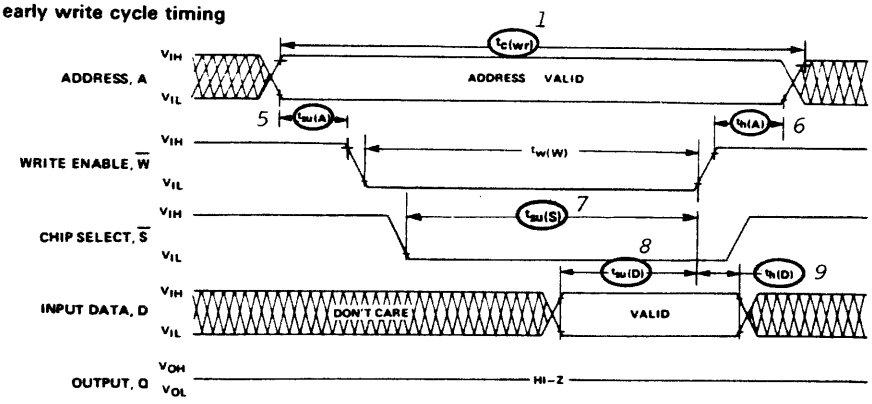


Figure 5-19. Memory Timing Diagram

In summary, memory interfacing involves careful consideration of total memory space allocation, and the decoding and buffering needed to implement a particular design so that no parts are overloaded and bus contention is prevented.

Timing

Memory timing requirements are fairly strict, and become more so as processor speed increases. Two basic memory data cycles are covered here--read cycle (data to processor) and write (data to memory). Note that only worst-case timing is specified; this is the only valid specification to use for design. Many data sheets show a "typical" specification which should never be used for design, since functional devices can have performance much different from the "typical" value.

Note the following timing sequence as described in Figure 5-19.

Read Cycle:

- Address lines come true from CPU.
- Block enable signal is decoded (Block 0- and Block 1-, from U34, Sheet 2 of University Board Schematic).
- Block enable (active low) enables specific IC.
- Data at the specified address is gated onto the data bus.

Write Cycle:

- Address lines come true.
- Block enable (chip select) is decoded.
- W- (Write Enable from CPU) comes active (low).
- Data from CPU comes true.
- W- goes high to enter data into memory.

The most critical parameters for proper memory timing have been circled and numbered on Figure 5-19 and are described below.

1. $t_{c(rd)}$--Read cycle time and $t_{c(wr)}$, Write cycle time, minimum time needed for either type of operation.
2. $t_{a(A)}$--Access time from address, indicates the earliest time after the addresses have settled that the output data is guaranteed to be available.
3. $t_{a(S)}$--Access time from chip select. Note that, in the example shown, this time is half of $t_{a(A)}$. This gives a generous time allowance for S (chip select or block select) to be decoded (general practice) from the address lines.
4. t_{pvx}--Output data valid after address change, define the maximum time data can be expected to be good as the processor finishes a data fetch. Note that if the RAM data is to be strobed into the processor, the strobe signal must be coincident with, or earlier than, the address change time.

5. $t_{su(A)}$--This parameter indicates the lead time (if any) required between address and Write Enable ($\overline{W-}$). The specifications shown imply that the two waveforms can be simultaneous.
6. $t_{h(A)}$--Address hold time, a positive value for this parameter indicates that the address lines must remain true longer than W-.
7. $t_{su(S)}$--Setup time for S-, minimum time this signal must be true before W- decays.
8. $t_{su(D)}$--Setup time for Data, minimum time the data must be true before W- decays.
9. $t_{h(D)}$--Data hold time, minimum time the data must be true after W- to ensure proper writing of data into memory.

The memory timing example above can be considered typical only in the basics. Each manufacturer uses different notation terminology for the various times. Sometimes the IC terminals and processor outputs also differ, so interpretation is often necessary.

5.4 PROGRAMMING AN EPROM FOR THE UNIVERSITY BOARD

As described in Chapter 2, the TM 990/189 has an executive program or monitor and symbolic assembler implemented in a masked ROM. The monitor manages all aspects of the microcomputer's operation in a manner specifically tailored to the classroom environment. Additional tasks can be specified by installing a supplemental program in expansion ROM area. If a larger control program is needed, a new executive can be designed. Designing a new executive program, as in any other program, should follow a certain sequence of operations:

- Define the problem in considerable detail.
- Decide what input/output structures will be needed, and define I/O addresses (see Chapters 6, 7, 8).
- Draw flowcharts of program action, subdividing each major portion into more detailed charts.
- Write assembly language code for each flowchart.
- Test and debug each code segment on the University Board taking particular pains with critical timing segments.
- Merge the program segments into one large program and enter this program into EPROM.
- Continue to debug the program as required and reprogram the EPROM.

The last steps in program development, merge, debug, and EPROM programming, can be handled very efficiently with another board in the TM 990 line, namely, the TM 990/302 Software Development Board (SDB). This system's memory map can easily be reconfigured to match that of the TM 990/189 microcomputer so as to utilize the new EPROM. On-board software for the SDB includes

° Text editor
° Symbolic assembler
° Relocating loader
° Debug monitor
° EPROM programming driver.

Once the program is entered, assembled, and debugged, special accessories are available to enable the user to program a number of different EPROM's, including the TMS 2708 and TMS 2716 which are usable on the University Board.

5.5 INSTRUCTION SUBSET 3

Both the instructions and data reside in computer memory. Some of the TMS 9980A instructions have been previously covered in Chapter 3 and 4. It is now appropriate to discuss the next subset of instructions. During this discussion, refer to the called-out Instruction Summaries.

Conditional Jump Instructions

It was previously noted that jump instructions modify the program counter and transfer control to another section of the same program. Conditional jump instructions incorporate a test of some condition of the bits in the status register. If the condition is met (test is true), then and only then is the control transfer (jump) made. A summary of the jump instructions (Table 3-2) appeared in Chapter 3.

JUMP ON CARRY and JUMP IF NO CARRY test only the CARRY status bit and jump only if the CARRY bit is a logic ONE or ZERO, respectively. See Instruction Summaries 5-1 and 5-2 for a discussion of these two instructions with an example of each.

Compare Instructions

Although some instructions (MOV, for example) automatically set status bits, other instructions are intended specifically only to set status bits and take no other action. This is the case with the COMPARE WORDS instruction covered previously in Chapter 3. In COMPARE ONES CORRESPONDING (COC) and COMPARE ZEROS CORRESPONDING, (CZC) a general source location contains a bit mask (bit positions identified by logic ONE's). These two instructions compare the contents of a specified workspace register with the bit mask in the source and set the EQUAL status bit accordingly. JUMP EQUAL or JUMP NOT EQUAL instructions would logically follow one of these compare instructions.

It is stressed that the EQUAL status bit is set to ONE in the COC instruction only when there is a corresponding ONE in the destination register for each ONE in the source bit mask. In a sense, if the COC instruction is followed by a JEQ instruction,

it is like a programmable AND gate where the programmer specifies with the source mask the exact bits in the destination register he wishes to check. If all such bit positions in the destination register are ONE's, the "gate" has an active output (EQUAL status bit set to ONE). On the other hand, if the programmer uses a JNE instruction after the COC instruction, then the action in the program is like that of a NAND gate--active if any corresponding bit in the destination register is ZERO.

In the case of the CZC instruction, the EQUAL status bit is set to ONE only when there is a corresponding ZERO in the destination for each ONE in the source-bit mask. Thus, operation of the CZC instruction when followed by a JEQ instruction is like that of a programmable NOR gate: only when all corresponding bits are ZERO's is the gate active, i.e., the jump is taken. The operation switches to a programmable OR gate if the CZC instruction is followed by a JNE instruction: if at least one of the corresponding bits is logic ONE, the gate is active and a transfer of control is made.

The essential characteristics of these two compare instructions are given in Instruction Summaries 5-3 and 5-4 along with operational examples for each.

Bit Manipulation Instructions

Most microcomputer control operations involve some form of bit manipulation. In a typical example, each bit represents on-off control of a single valve, light, motor, or other function. Even in computers which have bit-manipulation instructions, a bit mask is normally used to identify the bit or bits to be changed.

The TMS 9980A has a number of instructions which can manipulate any number of bits in a memory location. Instruction Summaries 5-5 and 5-6 explain how SET ONES CORRESPONDING and SET ZEROS CORRESPONDING utilize the source bit mask to set bits to ONE or ZERO, respectively. It is very important to note that the SOC instruction is the classical logical OR instruction while the SZC instruction is the same as the classical AND instruction except SZC uses the complement of the source, not the source itself. In similar fashion, the byte variants of these instructions (SOCB, SZCB) operate on specified bytes instead of on full words. Note in particular, however, that byte instructions which specify a register as source or destination operate on or with the even (lefthand) byte only. This shows up clearly in the example of SOCB. See Instruction Summaries 5-7 and 5-8.

AND IMMEDIATE, OR IMMEDIATE and EXCLUSIVE OR instructions also modify bits in a specified register by use of the bit map. The examples shown in Instruction Summaries 5-9, 5-10, and 5-11 all use the same destination and the same bit mask; note the different results in each case.

The special bit-map technique to be discussed next is a very powerful tool for those microcomputer applications which require

a considerable amount of bit manipulation or "bit diddling." The first step is to create a bit mask array equivalent that is shown in Figure 5-20. This array may be created at program assembly time by a succession of DATA directives as follows:

```
Label       Directive

B0          DATA >8000
B1          DATA >4000
B2          DATA >2000
B3          DATA >1000
B4          DATA >800
.           .
.           .
.           .
BF          DATA >1
```

Using the appropriate instructions, this bit mask array can then be used to manipulate and test selected bits as illustrated below:

```
Instruction         Comment

SOC @BA,R7          SET BIT 10 OF R7 TO LOGIC ONE
SZC @BD,@>3FC0      SET BIT 13 OF LOC >3FC0 TO LOGIC ZERO
XOR @B2,R7          CHANGE THE STATE OF BIT 2 IN R7
COC @B5,R7          IS BIT 5 OF R7 = 1?
CZC @B0,R2          IS MSB OF R2 = 0?
```

Shift Instructions

The SHIFT RIGHT ARITHMETIC, SHIFT RIGHT LOGICAL, and SHIFT RIGHT CIRCULAR instructions, Instruction Summaries 5-12, 5-13, and 5-14, move the location of all the bits in a register to the right a specified number of positions. The mathematical effect of shift instructions is to divide (shift right) or multiply (shift left) by 2^n, where n = number of shifts. Note that for unsigned numbers, the multiplication produces the correct answer, provided the programmer remembers to account for all ONE's shifted into the carry bit. For example, if >5A00 is shifted left three times:

```
1st shift results in >B400 with no carry
2nd shift results in >6800 with carry
        (new value = >16800)
3rd shift results in >D000 with no carry
        (new value = 2D000).
```

For signed numbers, the left shift on signed numbers will produce an incorrect value if the MSB (sign bit) ever changes during the shifting. Such a change in sign with the left shift instruction causes the OVERFLOW status bit to be set.

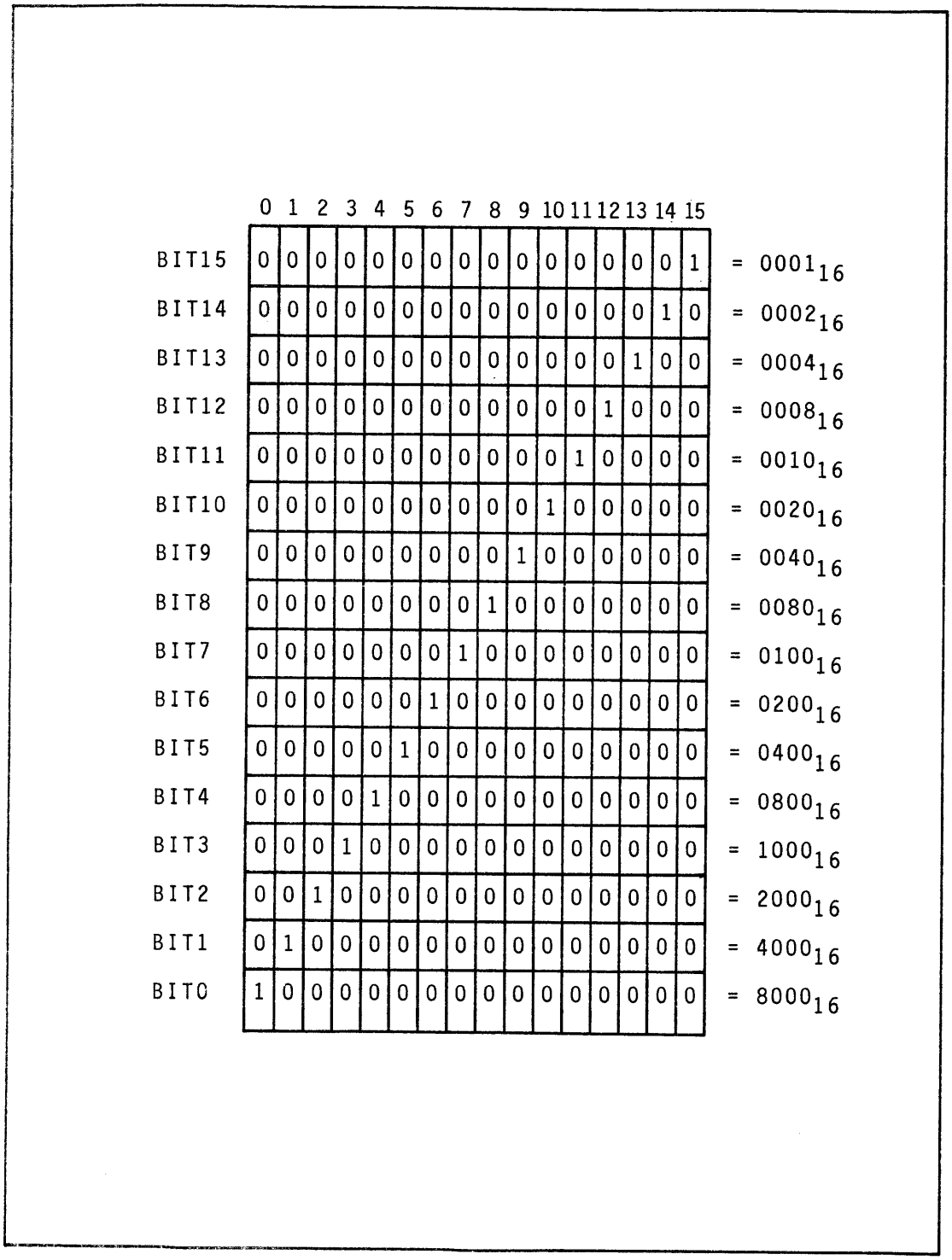


Figure 5-20. Bit Map Array

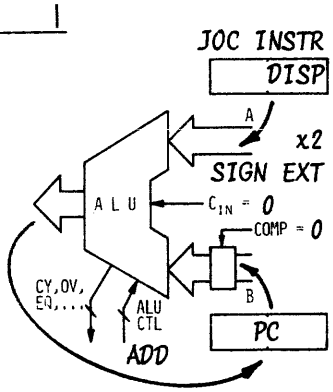
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: JOC Location 1800_{16}	0	0	0	1	1	0	0	0	Displacement (words)							

Length: 1 word

RESULT: If C = 1, (PC) + Displacement in (bytes) ⟶ (PC)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:



If the carry bit in the status register is set, add the signed displacement in bytes (of the machine code instruction) to the contents of the incremented program counter and place the sum in the program counter; otherwise, leave the PC unchanged. (The program counter is incremented after an instruction fetch.)

NOTES:

Used to transfer control to another part of the program (modify the contents of the program counter) if the carry status bit is set.

Example:

Location 3190_{16} Mnemonic JOC $+26 C = 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(PC) Before Instruction Fetch = 3190_{16}	0	0	1	1	0	0	0	1	1	0	0	1	0	0	0	0
$(3190_{16}) = 180C_{16}$	0	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0

$3190_{16} + 18_{16} + 2 = 31AA_{16}$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
New (PC) = $31AA_{16}$	0	0	1	1	0	0	0	1	1	0	1	0	1	0	1	0

Displacement (words) = (N/2) - 1
= (26/2) - 1
= $12_{10} = C_{16}$

New PC = (Displacement (words) x 2) + PC + 2
= (C_{16} x 2) + 3190_{16} + 2
= $31AA_{16}$

Instruction Summary 5-1

JUMP IF NO CARRY

JNC

CODE: JNC Location 1700_{16}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	1	1	Displacement (words)							

Length: 1 word

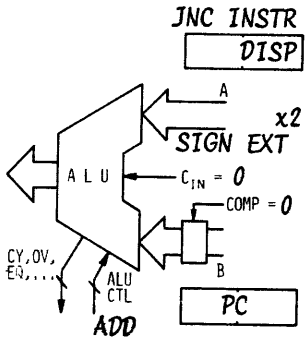
RESULT: If C = 0, (PC) + Displacement in bytes ⟶ (PC)

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

If the carry bit in the status register is ZERO, add the signed displacement in bytes (of the machine code instruction) to the contents of the incremented program counter and place the result in the program counter; otherwise, leave the PC unchanged. (The program counter is incremented after an instruction fetch.)



NOTES:

Used to transfer control to another part of the program (modify the contents of the program counter) if the carry status bit is cleared.

Example:

Location 3190_{16} Mnemonic JNC $-8 C = 0

(PC) Before Instruction Fetch = 3190_{16}

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

$(3190_{16}) = 17FB_{16}$

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

$3190_{16} + (-A_{16}) + 2 = 3188_{16}$

New PC = 3188_{16}

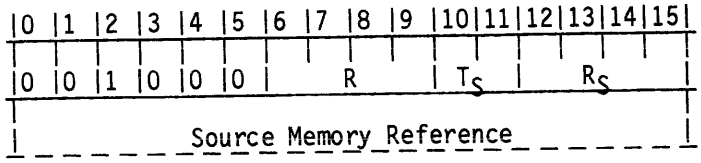
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	0	0	1	1	0	0	0	1	0	0	0

New PC = (Displacement (words) x 2) + PC + 2
= (-5_{16} x 2) = 3190 + 2
= 3188_{16}

Displacement (words) = (N/2) - 1
= (-8/2) - 1
= $-5_{10} = -5_{16}$

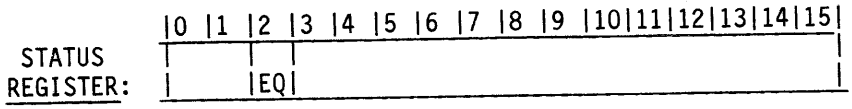
Instruction Summary 5-2

CODE: COC S,R



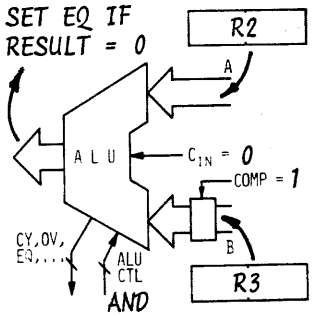
Length: 1 or 2 words

RESULT: (S) AND ($\overline{R}$); set equal status bit if result is zero.



OPERATION:

Set the EQ status bit if all the bits in the contents of the destination register that correspond to the logic ONE bits in the source operand are logic ONE bits.

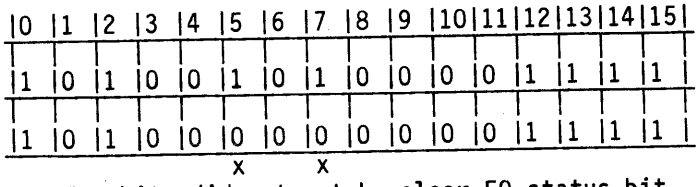


NOTES:

Used to test specified bits in the contents of any register. The bits to be tested are specified by logic ONE's in the bit mask (contents of the source).

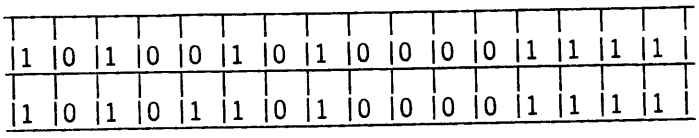
COC R2,R3

Example 1:



Two bits did not match; clear EQ status bit.

Example 2:



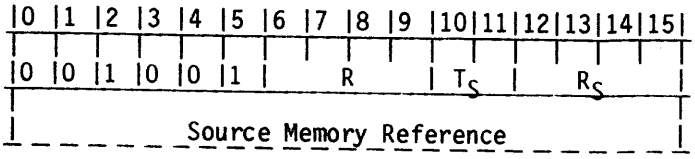
All ONE bits in mask have corresponding logic ONE bits in destination; set EQ status bit.

COMPARE ZERO'S CORRESPONDING

CZC

CODE: CZC S,R

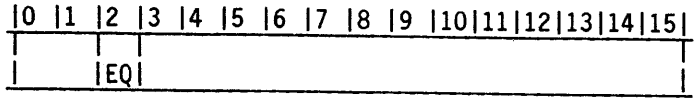
2400_{16}



Length: 1 or 2 words

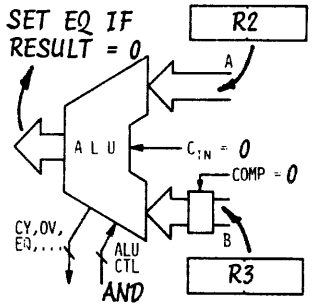
RESULT: (S) AND (R); set equal status bit if result is zero.

STATUS REGISTER:



OPERATION:

Set the EQUAL status bit if all the bits in the destination register that correspond to the logic ONE bits in the source operand are logic ZERO bits.

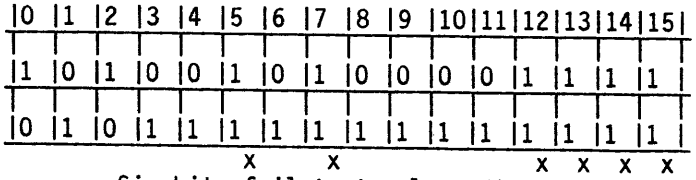


NOTES:

Used to test specified bits in the contents of any register. The bits to be tested are specified by logic ONE's in the bit mask (contents of the source).

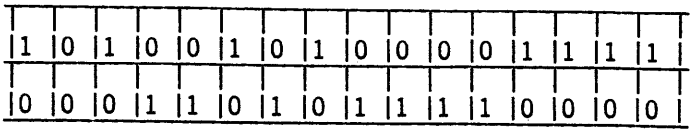
CZC R2,R3

Example 1:



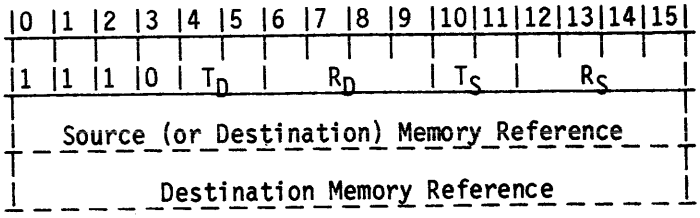
Six bits fail test; clear EQ status bit

Example 2:



All ONE bits in mask have corresponding logic ZERO in destination: set EQ status bit

CODE: SOC S,D $E000_{16}$



Length: 1 or 2 or 3 words

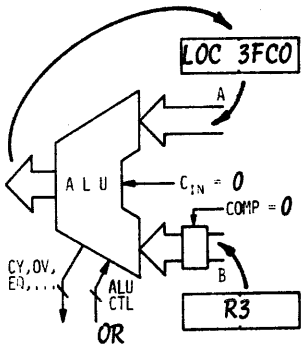
RESULT: (S) OR (D) ⟶ (D)

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ													

OPERATION:

Set to a logic ONE all bits in the contents of the destination that correspond to the same bit positions with a logic ONE in the contents of the source. Leave unchanged in the destination any bits that have corresponding ZERO's in the source. This is a logical OR of the contents of the source and the contents of the destination with the result placed in the destination. Compare the result to zero and set the status bits accordingly.



NOTES:

Used to set to a logic ONE a number of bits in the contents of the destination as specified by logic ONEs appearing in the bit mask (contents of the source). In short, this instruction is very useful for merging data fields into a single word or byte.

Example:

```
SOC  R3,@>3FC0
```

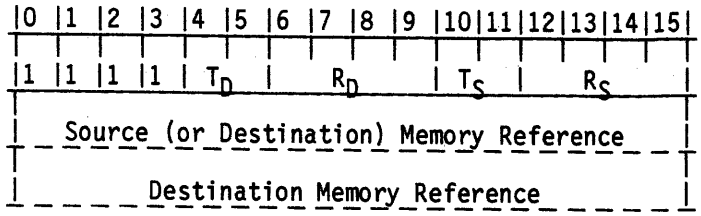
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Mask (R3)	1	0	1	0	0	1	0	1	0	0	0	0	1	1	1	1
(>3FC0) Before:	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
(>3FC0) After:	1	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1

Status: L> = 1; A> = 0; EQ = 0.

Instruction Summary 5-5

SET ONE'S CORRESPONDING (BYTE) — SOCB

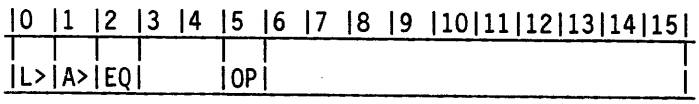
CODE: SOCB S,D — $F000_{16}$



RESULT: (S) OR (D) $\xrightarrow{\text{byte}}$ (D)

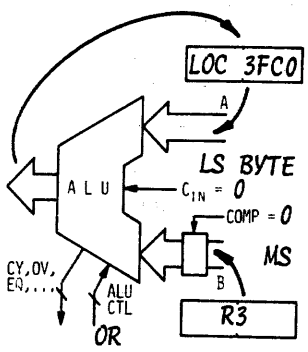
Length: 1 or 2 or 3 words

STATUS REGISTER:



OPERATION:

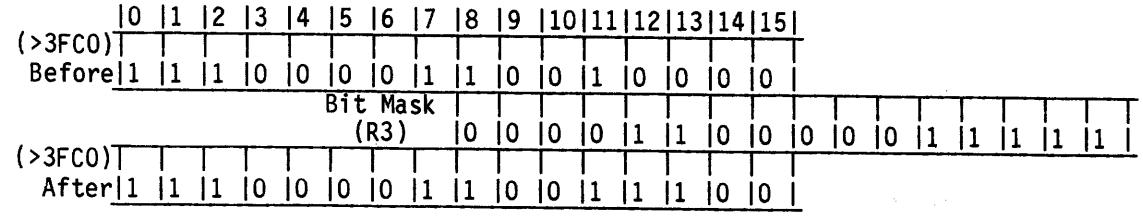
Set to logic ONE all bits in the specified byte of the contents of the destination that correspond to the same bit positions with a logic ONE in the specified byte of the contents of the source. This is a logical OR of the contents of the source and the contents of the destination. Leave unchanged in the destination any bits that have corresponding ZERO's in the source. Place the result in the specified byte of the destination. Compare the result to zero and set the status bits accordingly. The ODD PARITY status bit is set if the modified byte has odd parity.



NOTES:
Used to set to a logic ONE a number of bits in the contents of the destination as specified by logic ONE's appearing in the bit mask (contents of the source).

Example:

```
  SOC  R3,@>3FC1
```



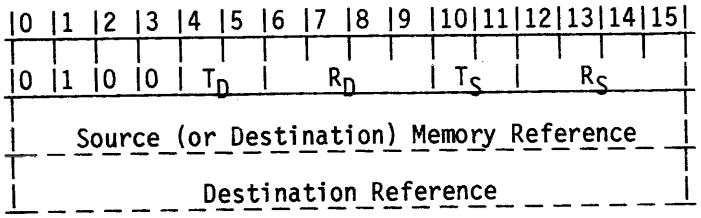
NOTE: >3FC1 is an odd byte. The byte instructions operate on or with the even byte of a workspace register. Thus, the bit mask must be displaced as shown to accomplish this instruction.

Status: L> = 1; A> = 0; EQ = 0; OP = 0.

Instruction Summary 5-6

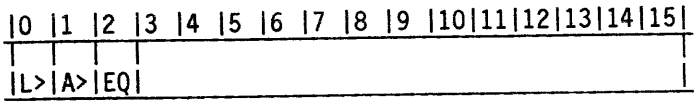
SET ZERO'S CORRESPONDING — SZC

CODE: SZC S,D — 4000_{16}



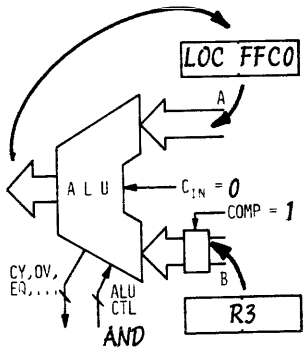
Length: 1 or 2 or 3 words

RESULT: $(\overline{S})$ AND (D) ⟶ (D)



OPERATION:

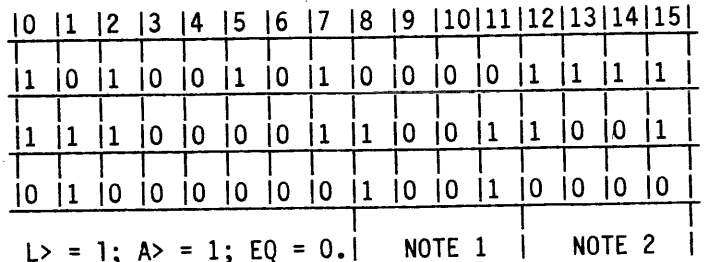
Set to a logic ZERO all bits in the contents of the destination that correspond to the same bit positions with a logic ONE in the contents of the source. Leave unchanged in the destination any bits that have corresponding ZERO's in the source. Place the result in the location specified as the destination. The contents of the source are not changed. This is similar to the AND instruction in other computers except that the complement of the source is AND-ed with the destination and the result placed in the destination. Compare the result to zero and set the status bits accordingly.



NOTES:

Used to set to a logic ZERO a number of bits in the contents of the destination as specified by the bit mask (contents of the source). This instruction is convenient for masking out or extracting fields of data as illustrated below.

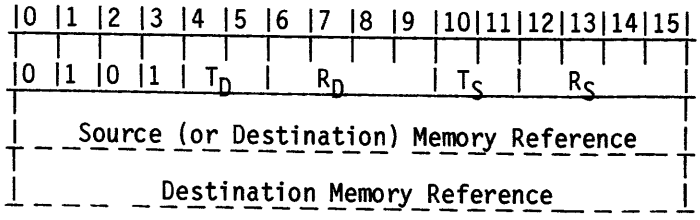
Example:
SZC R3,@>FFC0



1. All-ZERO nibble in bit mask extracts one nibble of data from original word.
2. All-ONE nibble in the bit mask clears the corresponding destination nibble.

SET ZERO'S CORRESPONDING (BYTE) SZCB

CODE: SZCB S,D 5000_{16}



Length: 1 or 2 or 3 words

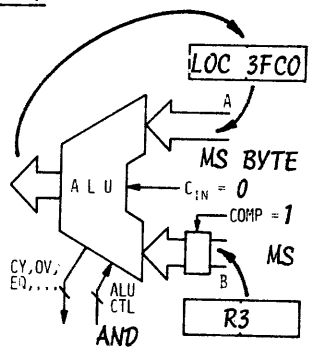
RESULT: $(\overline{S})$ AND (D) $\xrightarrow{byte}$ (D)

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L>	A>	EQ			OP										

OPERATION:

Set to a logic ZERO all bits in the specified byte of the destination that correspond to the same bit positions having a logic ONE in the same bit positions of the source byte. Leave unchanged in the specified destination byte any bits that have corresponding ZERO's in the source. Place the result in the location specified as the destination. The contents of the source are not changed. Compare the result to zero and set the status bits accordingly. The ODD PARITY status bit is set if the modified byte has odd parity.



NOTES:

Used to set to a logic ZERO a number of bits in the contents of the destination as specified by the logic ONE's appearing in the bit mask (the contents of the source).

Example:

SZCB R3,@>3FCØ

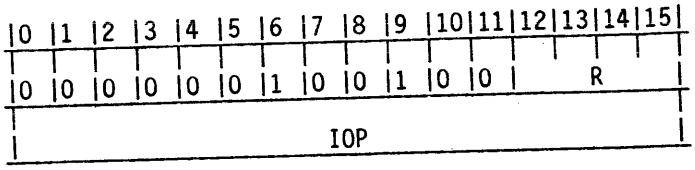
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Mask (R3):	1	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0
(>3FC0) Before:	1	1	1	0	1	1	0	1	0	0	0	1	1	1	1	1
(>3FC0) After:	0	0	0	0	1	1	0	0	1	0	0	1	0	0	0	0
	Changed								Unchanged							

Status: L> = 1; A> = 1; EQ = 0; OP = 0.

AND IMMEDIATE

ANDI

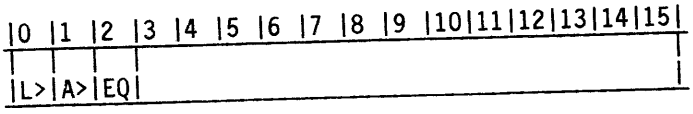
CODE: ANDI R,IOP 0240_{16}



Length: 2 words

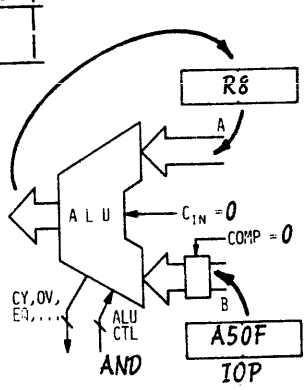
RESULT: (R) AND IOP ⟶ (R)

STATUS REGISTER:



OPERATION:

Perform a bit-by-bit logical AND operation between the contents of the specified register and the immediate operand (IOP). Place the result in the specified register. Compare the result to zero and set the status bits accordingly.

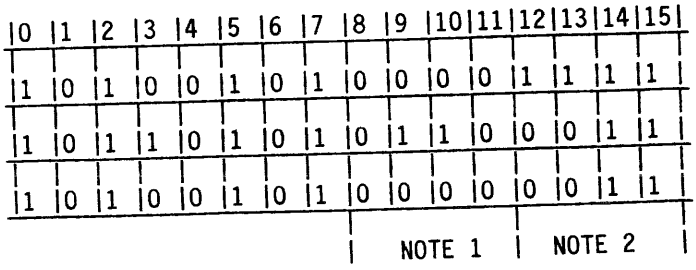


NOTES:

Used to "mask" off, or set to ZERO those bits in the contents of the specified register which are logic ZERO in the bit mask (IOP).

Example:

ANDI R8,>A50F



1. All-ZERO nibble masks out the corresponding destination nibble.
2. All-ONE nibble extracts one nibble of data from original word.

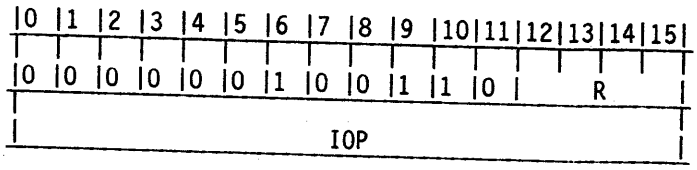
Status: L> = 1; A> = 0; EQ = 0.

Instruction Summary 5-9

OR IMMEDIATE

ORI

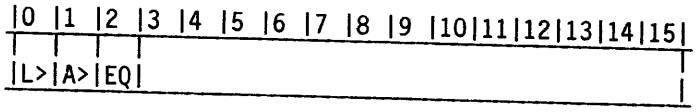
CODE: ORI R,IOP



Length: 2 words

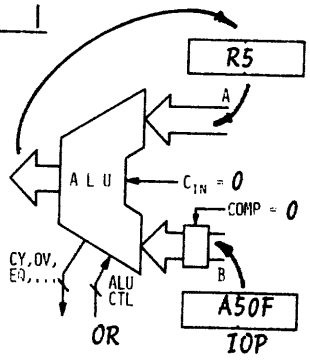
RESULT: (R) OR IOP ⟶ (R)

STATUS REGISTER:



OPERATION:

Perform a logical OR operation between the contents of the specified register and the immediate operand (IOP). Place the result in the specified register. Compare the result to zero and set the status bits accordingly.



NOTES:

Used to set to logic ONE those bits in the contents of the specified register which correspond to the logic ONE bits in the bit mask (IOP). Convenient for merging a data field from IOP into the designated register such as when converting from BCD to ASCII.

Example:

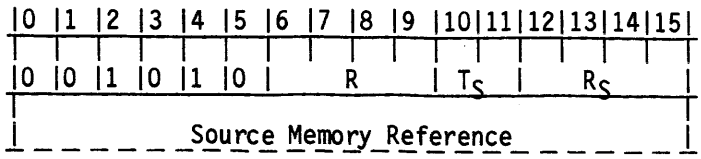
```
ORI  R5,>A50F
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Mask (>A50F):	1	0	1	0	0	1	0	1	0	0	0	0	1	1	1	1
(R7) Before:	1	0	1	1	0	1	0	1	0	1	1	0	0	0	1	1
(R7) After:	1	0	1	1	0	1	0	1	0	1	1	0	1	1	1	1

Status: L> = 1; A> = 0; EQ = 0.

EXCLUSIVE OR XOR

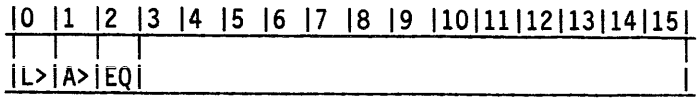
CODE: XOR S,R 2800_{16}



Length: 1 or 2 words

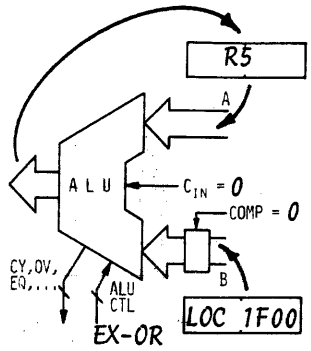
RESULT: (S) XOR (R) ⟶ (R)

STATUS REGISTER:



OPERATION:

Perform a bit-by-bit exclusive OR of the contents of the source with the contents of the specified destination register. Place the result in the destination register. Compare the result to zero and set the status bits accordingly.



NOTES:
Used to complement (invert) selectively those bits in the designated register which are specified by logic ONEs in the source (bit mask).

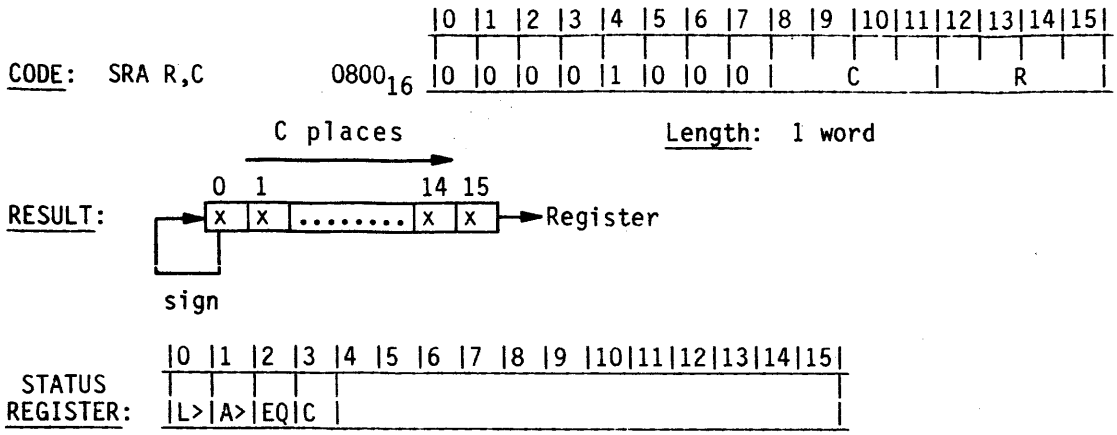
Example:

```
XOR  @>1F00,R5
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit Mask (1F00):	1	0	1	0	0	1	0	1	0	0	0	0	1	1	1	1
(R5) Before:	1	0	1	1	0	1	0	1	0	1	1	0	0	0	1	1
(R5) After:	0	0	0	1	0	0	0	0	0	1	1	0	1	1	0	0

Status: L> = 1; A> = 1; EQ = 0.

SHIFT RIGHT ARITHMETIC — SRA



OPERATION:

Shift the contents of the specified register to the right by the number of spaces specified by the count, C. Fill the vacated spaces with the sign bit (original Bit 0) value. The last bit shifted out is placed in the carry status bit. If C is 0, the shift count is defined by the lowest four bits of R0. If these four bits of R0 are also equal to zero, the shift count is 16. Compare the result to zero and set the other status bits (L>, A>, EQ) accordingly.

NOTES:

Used to shift the contents of a memory location to the right while preserving the sign bit. If the word to be modified is not in a workspace register, execute a MOV instruction to load the register, then SRA.

Example:

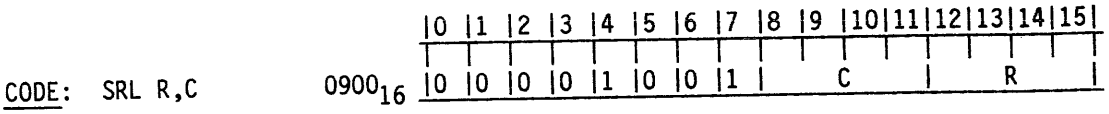
```
SRA  R9,5
```

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(R9)	Before:	1	0	0	1	1	1	1	0	0	1	1	1	0	0	0	0
(R9)	After:	1	1	1	1	1	1	0	0	1	1	1	1	0	0	1	1

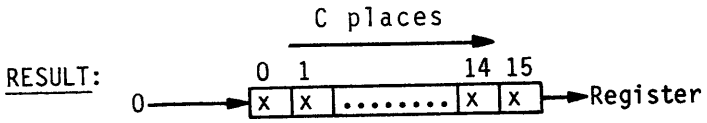
Status: L> = 1; A> = 0; EQ = 0; C = 1.

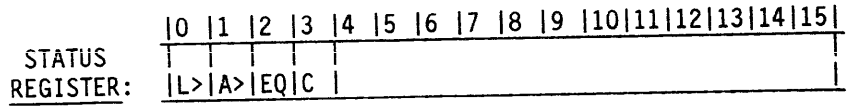
SHIFT RIGHT LOGICAL

SRL



Length: 1 word





OPERATION:

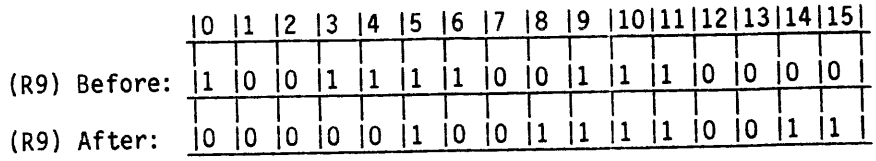
Shift the contents of the specified workspace register to the right by the number of bit positions specified by the count, C. Fill the vacated positions with logic ZERO. The last bit shifted out is placed in the carry status bit. If C is 0, the shift count is defined by the lowest four bits of R0; if these four bits of R0 are also equal to 0, the shift count is 16. Compare the result to zero and set the other status bits (L>,A>,EQ) accordingly.

NOTES:

Used to shift the contents of a register to the right. If the word to be modified is not presently in a register, execute a MOV instruction to load the register, then SRL.

Example:

```
SRL R9,5
```

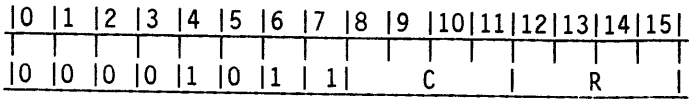


Status: L> = 1; A> = 1; EQ = 0; C =1.

SHIFT RIGHT CIRCULAR

SRC

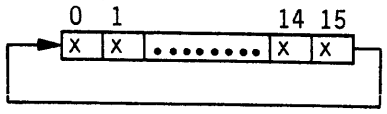
CODE: SRC R,C



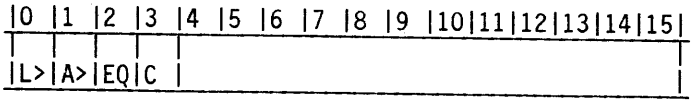
0800_{16}

Length: 1 word

RESULT:



STATUS REGISTER:



OPERATION:

Shift the contents of the specified register to the right by the number of bit positions specified by the count, C. Each bit shifted out of bit position 15 is shifted into bit position 0. The state of the last bit shifted out of bit position 15 is placed in the carry status bit. If C is 0, the shift count is defined by the lowest four bits of R0. If these four bits of R0 are also equal to 0, the shift count is 16. Compare the result to ZERO and set the other status bits (L>, A>, EQ) accordingly.

NOTES:

Used to shift the contents of any register to the right without losing any of the bits. That is, the original data can be restored by executing a second shift where the shift count is the 16's complement of the first shift count.

Example:

SRC R9,5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(R9) Before:	1	0	0	1	1	1	1	0	0	1	1	1	0	0	0	0
(R9) After:	1	0	0	0	0	1	0	0	1	1	1	1	0	0	1	1

Status: L> = 1; A> = 0; EQ = 0; C = 1.

5.6 PROGRAM EXAMPLE: MEMORY TEST

A program to verify correct operation of RAM is very useful in the operation of any computer system. Such a program is used routinely as a system check, or reserved for trouble-shooting time. One such program, called the memory test module, is presented in flowchart form in Figure 5-21. The flow chart graphically represents the program flow, showing an entry point (BEGIN block), an exit point, numerous operations (rectangles), and decision points (diamond-shaped blocks). With the program flow diagrammed this way, the logic is easier to follow.

The memory test module performs the following tests.

- Writes 0000_{16} to each memory location, beginning and ending at specified locations.
- Checks each test location to be sure that all locations contain 0000_{16}.
- Writes patterns 8000_{16}, $C000_{16}$, $E000_{16}$ ... $FFFF_{16}$ in succession to each location, checking for correct patterns at each test location and verifies correct operation.
- Reports each fault by displaying the faulty memory location, the pattern which was read back, and the test pattern.

Figure 5-22 shows a listing of the memory test module after it was assembled (translated from assembly language to machine language) on a 990 family minicomputer driving a line printer. A listing is a printed document produced by an assembler program which shows, among other things, a comparison of the assembly language statements input to the assembler and the machine code produced by the assembler during translation. The listing is produced with seven columns of data:

- Line numbers
- Address locations
- Machine code
- Labels (used for programmer convenience)
- Instruction nmemonics
- Operands
- Comments (very important in helping users to understand program flow)

It should be noted that different assembler programs produce different listings, according to the available equipment and operating system requirements. Line numbers (which identify each program statement and comment taken from the source listing) will not be produced by the TM 990/189 assembler. The student will utilize only the address locations and the machine code from this listing while entering the program. The listing should be studied as a whole to understand program operation.

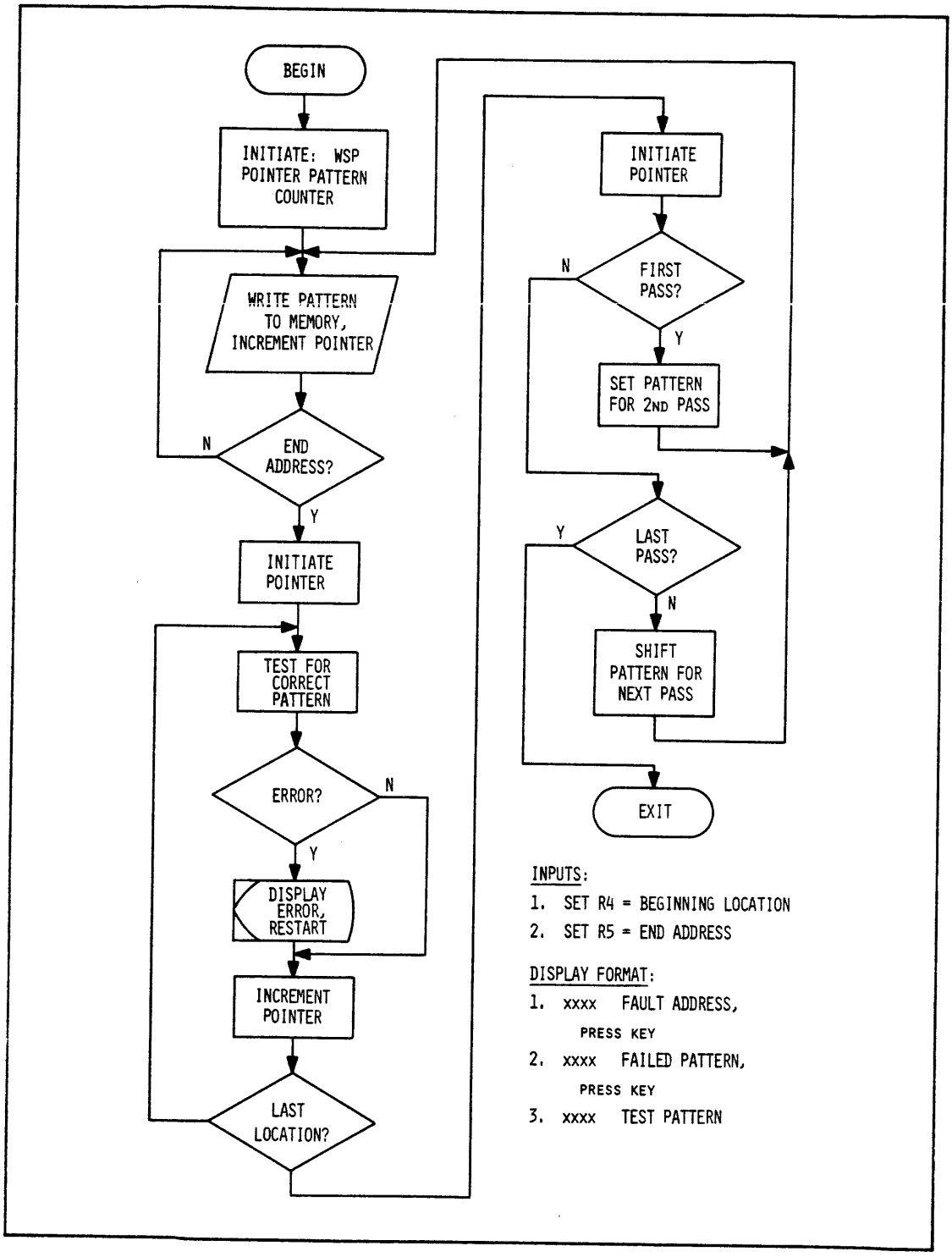


Figure 5-21. Flowchart for Memory Test Module

```
0001                *USER INSTRUCTIONS:  LOAD ADDRESS FOR TEST START IN R4 AND
0002                *                    END ADDRESS IN R5.
0003                      IDT   'MEMTEST'
0004 0200                 AORG  >200          DEFINE ORIGIN
0005 0200           WS    BSS   32            RESERVE MEMORY FOR WORKSPACE
0006 0220 02E0      ST    LWPI  WS            INITIALIZE WORKSPACE POINTER
     0222 0200
0007 0224 0206            LI    R6,>0D00      CARRIAGE RETURN AND LINE FEED
     0226 0D00
0008 0228 0207            LI    R7,>0A00      USED TO BLANK DISPLAY
     022A 0A00
0009 022C 04C2            CLR   R2            CLEAR ERROR COUNTER
0010 022E 04C3            CLR   R3            SET FIRST TEST PATTERN
0011 0230 C044            MOV   R4,R1         LOAD STARTING ADDRESS
0012 0232 CC43      OT    MOV   R3,*R1+       OUTPUT PATTERN TO MEMORY
0013 0234 8141            C     R1,R5         CHECK FOR END ADDRESS
0014 0236 12FD            JLE   OT            NOT LAST ADDRESS, RETURN
0015 0238 C044            MOV   R4,R1         SET STARTING ADDRESS AGAIN
0016 023A 80D1      IN    C     *R1,R3        TEST FOR CORRECT PATTERN
0017 023C 130D            JEQ   IC            NO ERROR, SKIP ERROR ROUTINE
0018 023E 0582            INC   R2            TALLY ERROR
0019 0240 2E81            XOP   R1,10         OUTPUT ERROR ADDRESS
0020 0242 2F49            XOP   R9,13         USER PAUSE TO RECORD DATA
0021 0244 2F06            XOP   R6,12         BLANK DISPLAY BY SENDING
0022 0246 2F07            XOP   R7,12         CARRIAGE RETURN AND LINE FEED
0023 0248 2E91            XOP   *R1,10        OUTPUT FAULTY PATTERN
0024 024A 2F49            XOP   R9,13         READ ONE CHARACTER, NO ECHO
0025 024C 2F06            XOP   R6,12         BLANK DISPLAY BY SENDING
0026 024E 2F07            XOP   R7,12         CARRIAGE RETURN AND LINE FEED
0027 0250 2E83            XOP   R3,10         OUTPUT TEST PATTERN
0028 0252 2F49            XOP   R9,13         USER PAUSE TO RECORD DATA
0029 0254 2F06            XOP   R6,12         BLANK DISPLAY BY SENDING
0030 0256 2F07            XOP   R7,12         CARRIAGE RETURN AND LINE FEED
0031 0258 05C1      IC    INCT  R1            TEST NEXT ADDRESS
0032 025A 8141            C     R1,R5         CHECK FOR END ADDRESS
0033 025C 12EE            JLE   IN            NOT END ADDRESS, RETURN
0034 025E C044            MOV   R4,R1         SET STARTING ADDRESS AGAIN
0035 0260 0283            CI    R3,0          CHECK FOR FIRST PASS
     0262 0000
0036 0264 1603            JNE   NX            IF NOT, TEST FOR LAST PASS
0037 0266 0203            LI    R3,>8000      SET UP PASS TWO PATTERN
     0268 8000
0038 026A 10E3            JMP   OT            TEST WITH NEW PATTERN
0039 026C 0283      NX    CI    R3,-1         CHECK FOR LAST PASS
     026E FFFF
0040 0270 1302            JEQ   DN            CHECK FOR TEST DONE
0041 0272 0813            SRA   R3,1          CHANGE PATTERN FOR NEXT PASS
0042 0274 10DE            JMP   OT            TEST WITH NEW PATTERN
0043      0276      DN    EQU   $
0044 0276 0340            IDLE
0045      0220            END   ST
NO ERRORS
```

Figure 5-22. Listing for Memory Test Module

The program begins operation by initializing the workspace pointer, the initial pattern to be written to the memory and the error counter. The first program loop begins by writing the first pattern (0000) to the first memory location to be tested, incrementing the pointer (test address) and then checking for the end address. Until the end address appears in the pointer location, the loop will continue. Next, the pointer is set again to the starting address, and each location is tested (read) for the correct pattern. If any location does not match the test pattern, the address which failed is reported, and the a program waits for the operator to press any key. When a key is pressed, the program resumes operation briefly to display the pattern contained by the memory address which failed. Another pause for a keystroke occurs while the operator records the faulty pattern. Finally, a second keystroke causes the test pattern to be displayed until it is terminated by a third keystroke. Note that XOP instructions are used to implement the interactive display error reporting. XOP's are utilized for this special purpose and are discussed in detail in Chapter 8.

Assuming the error found was an isolated instance, the program will continue testing memory until all specified locations have been tested. If a contiguous block of memory is faulty, each successive location will be reported as faulty. The record/keystroke sequence discussed above must be repeated for each location.

After the first pass (a pass consists of writing a pattern and then testing for errors), the pattern is changed from 0000_{16} to 8000_{16} and the whole write-and-test section is performed again. After each successive pass, the pattern is shifted (SRA) so that the pattern progresses thus: 8000_{16}, $C000_{16}$, $E000_{16}$ etc., and the test is repeated with each new pattern.

Operate the test in the following manner. Load the program beginning at 0200_{16}, then put the address of the first location to be tested into workspace register R4, and the end address + 1 in R5. Start the test, and record the errors found, if any. Select as the starting address the first free address after the end of the test program, and set 0400_{16} as the ending address. This tests all the remaining RAM in the normal RAM area, and should yield one error at location 0400_{16} if the TM 990/189 is operating properly. See the section on LAB EXPERIMENTS below for further tests with this program.

5.7 SUMMARY

The field of data storage for computers is very diverse. The technology has advanced to the point where a wide variety of semiconductor memory technologies is available to facilitate the cost-effective adaptation of microcomputer control to a wide variety of tasks.

Mass memory has also kept pace and is being addressed by a number of semiconductor technologies.

The University Board is treated as an example of a microprocessor system design using the latest available static memory technology. Also, the latest development in buffers, byte wide drivers, are employed for off-board expansion.

This chapter discusses the third subset of TMS 9980A instructions. This subset expands the programming versatility by adding to the available shift and jump instructions, and introduces the concept of bit manipulations and testing.

Finally, a memory diagnostic tool is furnished which allows easy checkout of memory reactions. The diagnostic is useful both for trouble-shooting the University Board and for checkout of new memory designs.

5.8 EXERCISES

1. The TMS 9980A has 14_{10} address lines.

   (a) How many locations in memory will it address directly?

   (b) If a program requires only two workspaces and a 60-byte buffer is required, how many bytes of read-write memory is available on a fully populated University Board? (Assume 146_{16} bytes are used for monitor buffer.)

2. Specify the lowest cost design of a memory system for applications detailed below, assuming the following expenses.

   - PC board area - $0.50 per memory component
   - Power supply - $15 per voltage required
   - IC costs as detailed below in Table 5-1.

Table 5-1

IC	Power Supplies	Process	Access (ns)	Cost
32 x 8 PROM	+5v	STTL	50	$2.75
256 x 4 PROM	+5v	STTL	50	$3.63
256 x 8 PROM	+5v	STTL	50	$6.35
1K x 8 PROM	+5v	STTL	50	$32.20
128 x 8 RAM	+5v	NMOS	450	$4.50
256 x 4 RAM	+5v	NMOS	450	$2.08
1K x 1 RAM	+5v	NMOS	450	$1.92
1K x 4 RAM	+5v	NMOS	450	$13.50
64 x 9 RAM	+5v	STTL	50	$22.00
1K x 1 RAM	+5v	STTL	50	$17.60
1K x 8 EPROM	+5v,-5v,+12v	NMOS	450	$13.95

(a) A bootstrap loader for a particular microprocessor needs 48 bytes, and the intended application will use 1K bytes of program and a maximum of 128 bytes of scratch-pad RAM. Select components for the lowest cost memory, assuming that the bootstrap loader will be used to enter program code. Required memory access time--450 ns.

(b) Select components for a memory with 500-ns access time, 1800 bytes of program in PROM or EPROM, and a minimum of 128 bytes of scratch-pad RAM.

(c) Repeat (B) above for a memory with 100-ns access time.

3. Refer to Figure 5-19 and assume S comes true on a TMS 4014-120 ns after the address is valid. How long before the end of the read cycle will the data become valid, if $t_{c(rd)}$ is 650 ns?

4. In exercise 3, what is the maximum time one can expect data to be valid?

5. If register R5 contains >4800, what bit mask is required to change the contents of R5 to >4867 when using either the SOC or the ORI instruction? Write the source code two different ways to achieve this change?

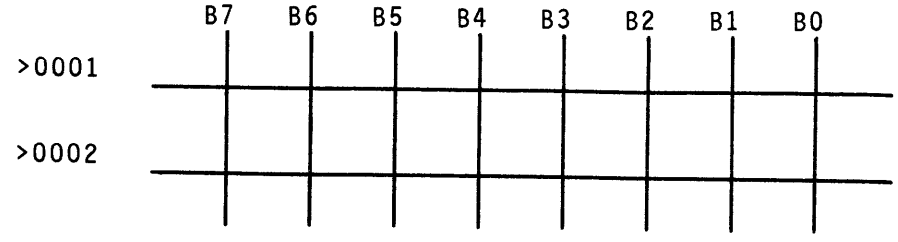
6. Sketch the arrangement of address and data lines for a 4K x 8 memory using

(a) 1K x 4 IC's
(b) 4K x 1 IC's

7. Assume that register R5 contains >A800 before the execution of each instruction listed below. Enter the R5 contents of R5 which will result from executing each instruction.

```
ANDI R5,>A200      (R5) = _____
ORI R5,>B000       (R5) = _____
XOR @>1000,R5      (R5) = _____
  (>1000) = >100F
```

8. Sketch a diode ROM which contains >B2 in location >0001 and >C3 in location >0002.



5.9 LAB EXPERIMENTS

1. When the memory test module (Figure 5-22) was used to test on-board RAM with the instructions provided, the result should have been one error reported at >0400. Run the test again, specifying >0405 as the ending address (value in R5). Record the results and explain why the failure reporting pattern changed.

2. Run the memory test again, with >3000 in R4 and >3002 in R5. Record and explain the results.

3. Modify the test so that R3 is cleared by using SZC, then repeat Experiment 1 above as a test of program validity. The program originally used 28 instructions. How many are now needed?

4. Report the number of bytes required for both the original program and for the modified program developed in Experiment 3.

5. Modify the test so that SRL is used to change the test pattern. Then test the new program as in Experiment 3. What additional program steps must be used so that the same test patterns will be used?

CHAPTER 6

INPUT/OUTPUT CONCEPTS

6.1 INTRODUCTION

In preceding chapters, two of the three major parts of a complete microprocessor system are explored: the processor and a system's memory. This chapter begins a detailed study of the third part of a system: the input/output section (I/O).

The main categories of I/O operation are discussed along with the advantages and disadvantages of each. These main categories are then related to I/O options available with the TMS 9980A on the University Board. One of these options, an I/O port called the "Communications Register Unit," is a unique feature of the TI 9900 microprocessor family architecture. This port, called the CRU for short, and its operation receive special attention. The TMS 9980A instruction set includes a group of instructions associated with the CRU. These instructions and their operation are explained.

A program example at the end of the chapter illustrates the operation of the CRU in an application.

Additional UNIBUG commands are introduced to allow the operator to store and retrieve data to and from tape cassettes using the I/O ports on the TM 990/189.

6.2 COMPUTER SYSTEM REVIEW: I/O FUNCTION

Refer to Figure 6-1 and recall that a microprocessor system is composed of three main parts: the processor, memory, and I/O. Discussions thus far have dealt with the characteristics and functions of the processor and memory. Now the focus is on the I/O subsystem.

The function of the I/O section is to communicate with and interact between the processor (including its programs and data in memory) and the world outside the processor system. As depicted in Figure 6-2, the I/O section is composed of devices with which the processor interacts or communicates, interface logic components used to condition or coordinate interaction between those devices and the system, and those portions of a processor's architecture that provide for interaction.

The interaction between a processor system and peripheral I/O devices includes the passage of data, including commands and status

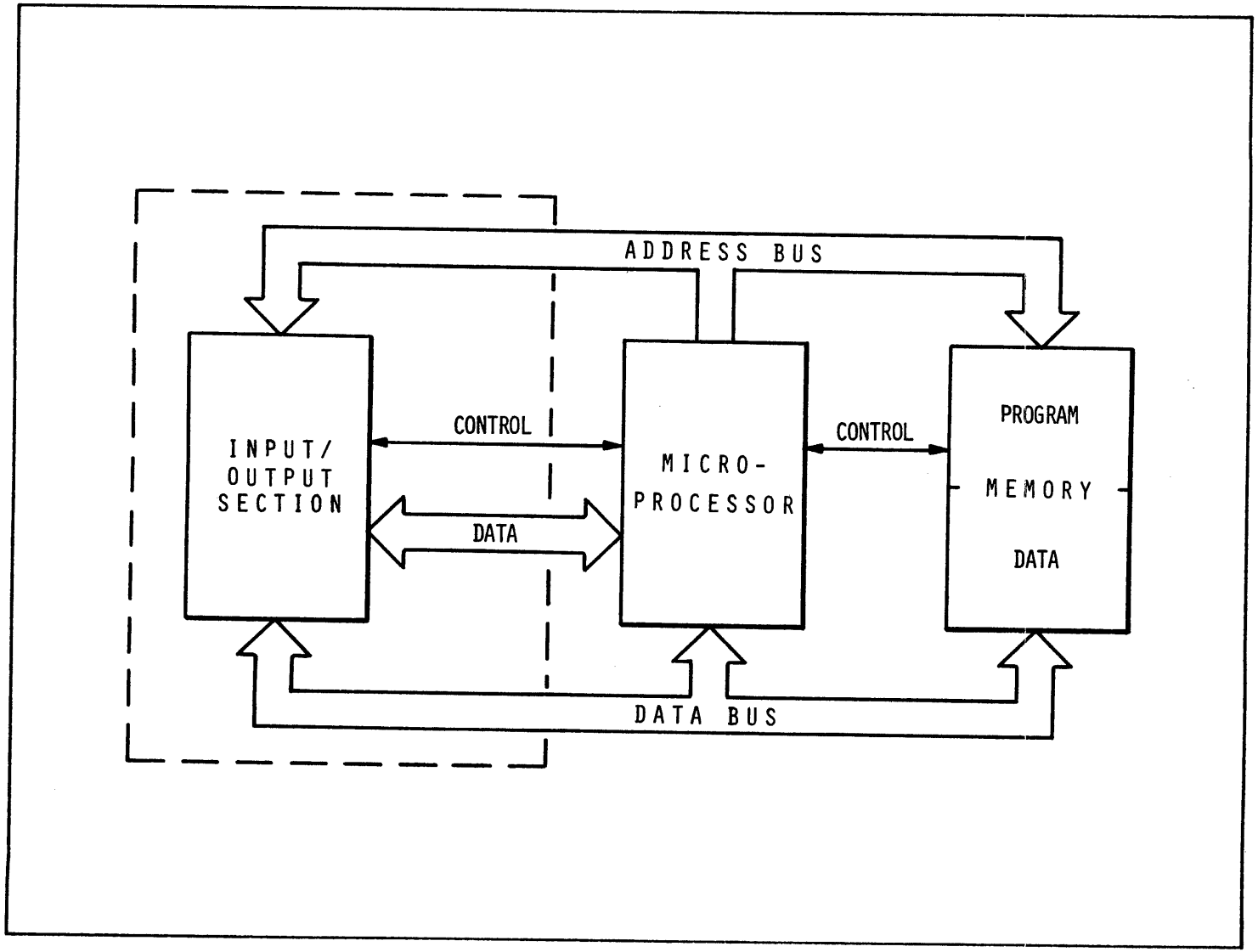


Figure 6-1. The Input/Output Section in a Microprocessor System

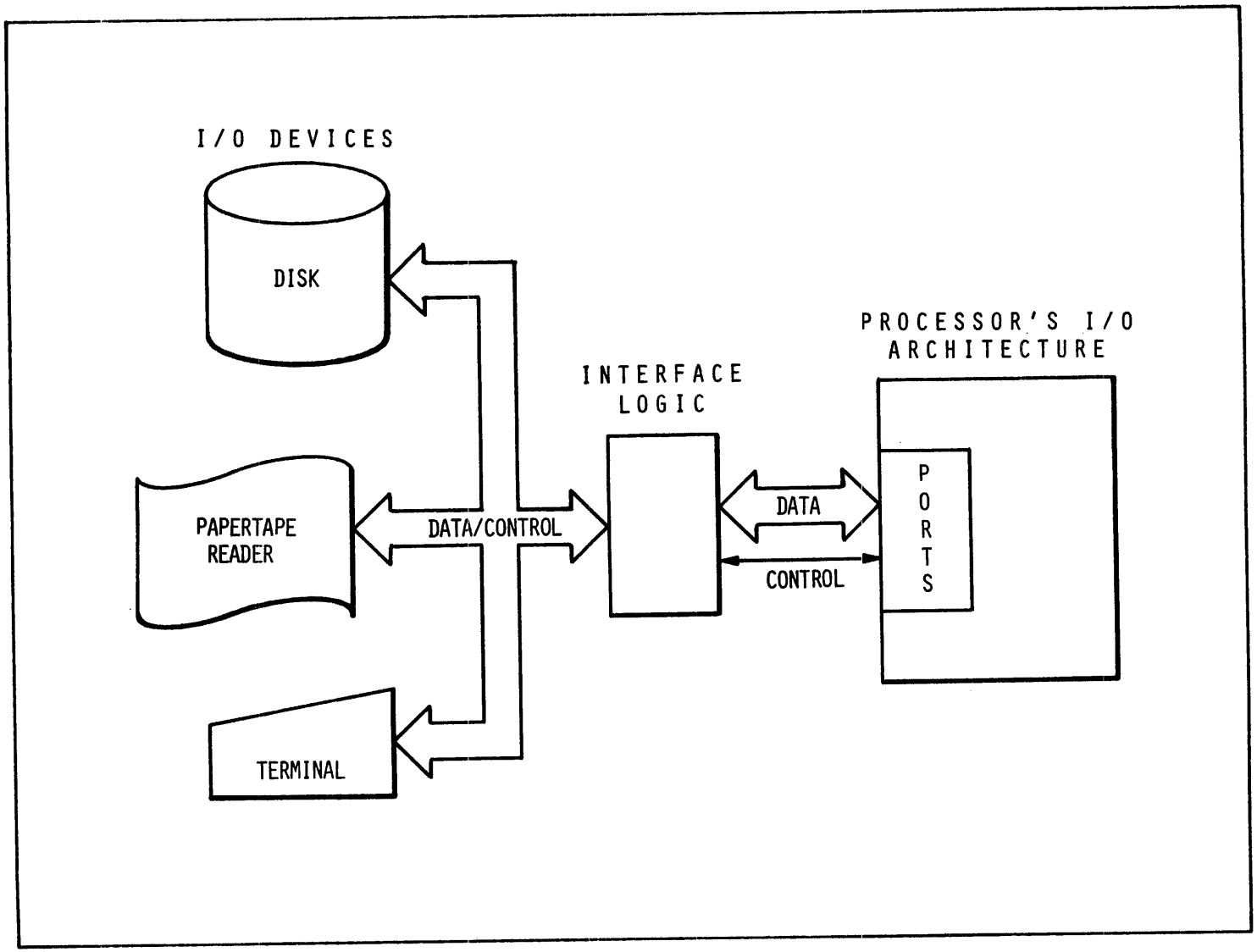


Figure 6-2. The Input/Output Section

requests. In addition, the I/O section is normally designed to alert the processor through the use of interrupts when a device needs servicing. As with the memory section, the I/O section performs under direction of control signals from the processor.

In a complete system, it is not uncommon for the I/O section to be the most complex part.

"Throughput" is often used as a prime measurement of the performance of a processor system. Throughput, which is the amount of data that can be input, processed, and output by a system in a given period of time, is largely influenced by the percentage of I/O operation compared to the precentage of processing operation and, within the I/O section, by the speed, efficiency, and architecture of the I/O ports.

"Port" is the name given to an interface between I/O devices and the processor system. It is composed of one or more I/O lines.

Interface circuitry is normally used between a port and an I/O device to allow a device to recognize when it is being addressed, to allow a device to alert the processor when it needs servicing, and to coordinate the transfer of information between a device and the processor's ports.

The interface circuitry may consist of several discrete small- or medium-scale integration (SSI or MSI) components. It is becoming more common, however, to combine the interface functions of address decoding, control, and data transfer within a single LSI component. The trend is also toward intelligent, programmable LSI peripheral components the I/O functions of which can be configured upon command from the processor and even dynamically reconfigured by the processor as a program is running.

The devices in the I/O periphery are as varied as the applications. They can be used to store data "offline;" that is, data that does not have to be "online" or held continually in the system's memory components. Examples of such devices were mentioned in the last chapter: tape transports, disks, card readers, and other mass-storage devices. Other examples of I/O devices include those which control actions or sense conditions, such as relays and switches. Devices can also be used to measure magnitude, distance, speed, or time. A thermistor, for example, can measure the temperature within a furnace. A thermistor is a device the resistance of which varies inversely with the temperature of its environment. It converts temperature into an electrically measureable quantity. However, such a measurement is part of a continuous or "analog" range of values, not a digital value. Before this quantity can be used directly by the processor, it must be digitized (converted to a binary value). Figure 6-3 shows a representative analog signal produced by a thermistor and a converted digital signal that approximates the analog signal. As shown in Figure 6-4, a common method of altering a signal is to employ an analog-to-digital converter.

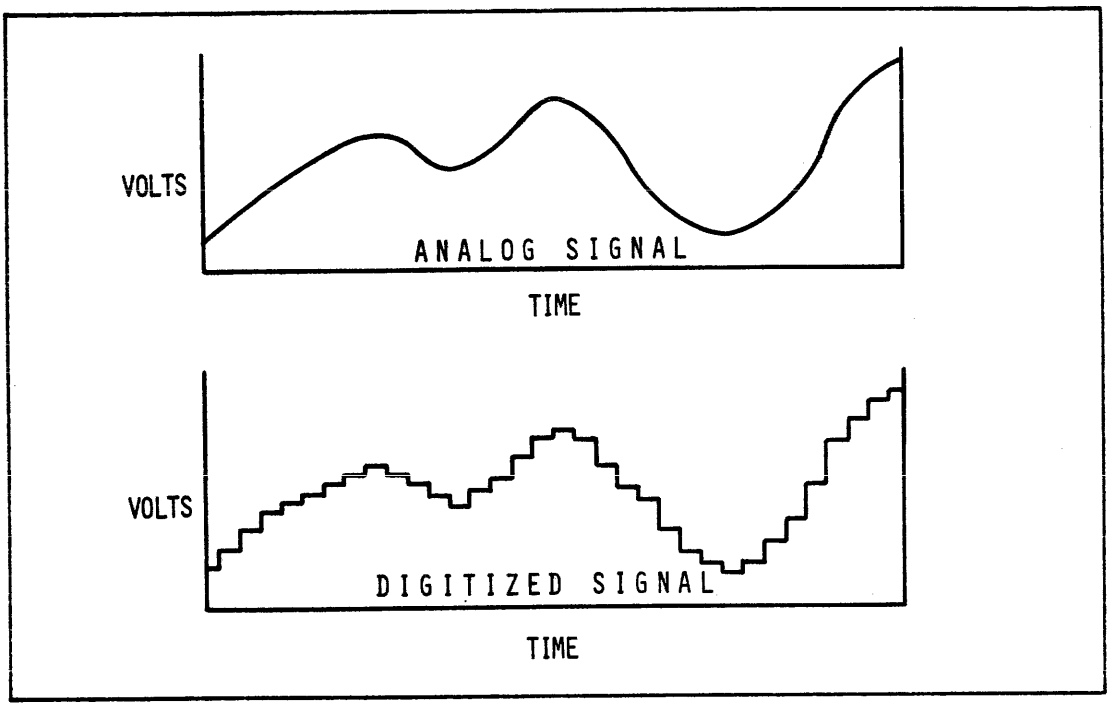


Figure 6-3. An Analog Signal Versus a Digital Signal

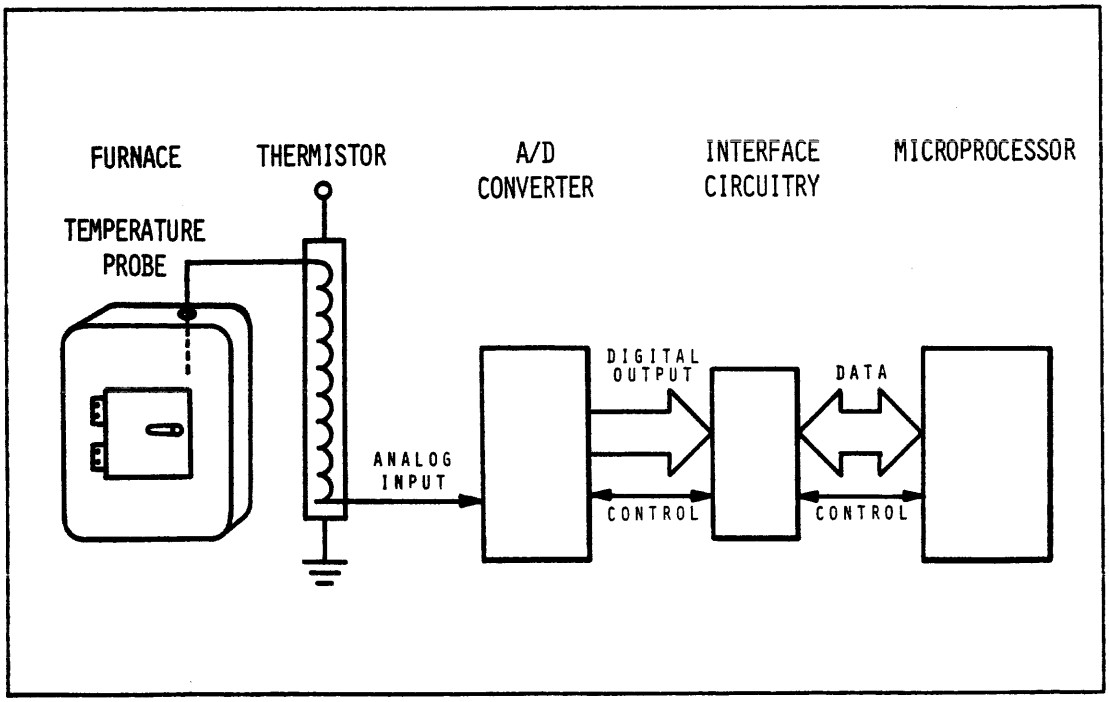


Figure 6-4. Digitizing an Analog Measurement with an A-to-D Converter

This electronic device converts an analog signal to a digital value suitable for input to a processor.

Conversely, if a processor is to output a value to an analog device, it is necessary to convert the digital output from the processor to an analog value. Another I/O device, a digital-to-analog converter, does this. As the name implies, this device takes a binary value either in parallel or in series (depending upon the D/A device) from the processor and quantifies the binary value into an analog value.

As an example, refer to Figure 6-5 where a processor is controlling an electric motor. The speed of the motor is controlled by a varying current. The processor outputs a binary quantity to indirectly express the amount of current to be fed to the motor. The binary value is converted by a digital-to-analog converter into an analog value to directly control current to the motor.

Even with binary data, I/O periphery is often necessary to limit or expand the voltage range of the signals or to otherwise condition the data.

Now that the function of the I/O section in a system has been determined, the various approaches to implementing the I/O will be examined.

6.3 OVERVIEW OF I/O CATEGORIES

Generally speaking, there are three categories of I/O operation:

- Program-controlled I/O
- Interrupt-driven I/O
- Direct memory access I/O.

Each category has its own unique characteristics, advantages, and disadvantages.

Program-Controlled I/O

Program-controlled I/O is the simplest method of operation. A program in the memory initiates all communication and data transfers between the processor and peripheral devices. Normally, the program causes the processor to put a device address on the address bus, and then outputs a control signal to the I/O interface to alert the I/O section that a device is being addressed. The I/O interface decodes the address on the address bus to determine which device is being addressed, and in the same operation alerts the device.

Included within the data on the address bus, or included in a subsequent piece of data, is a command from the processor allowing the device to determine what is needed from it. This command may

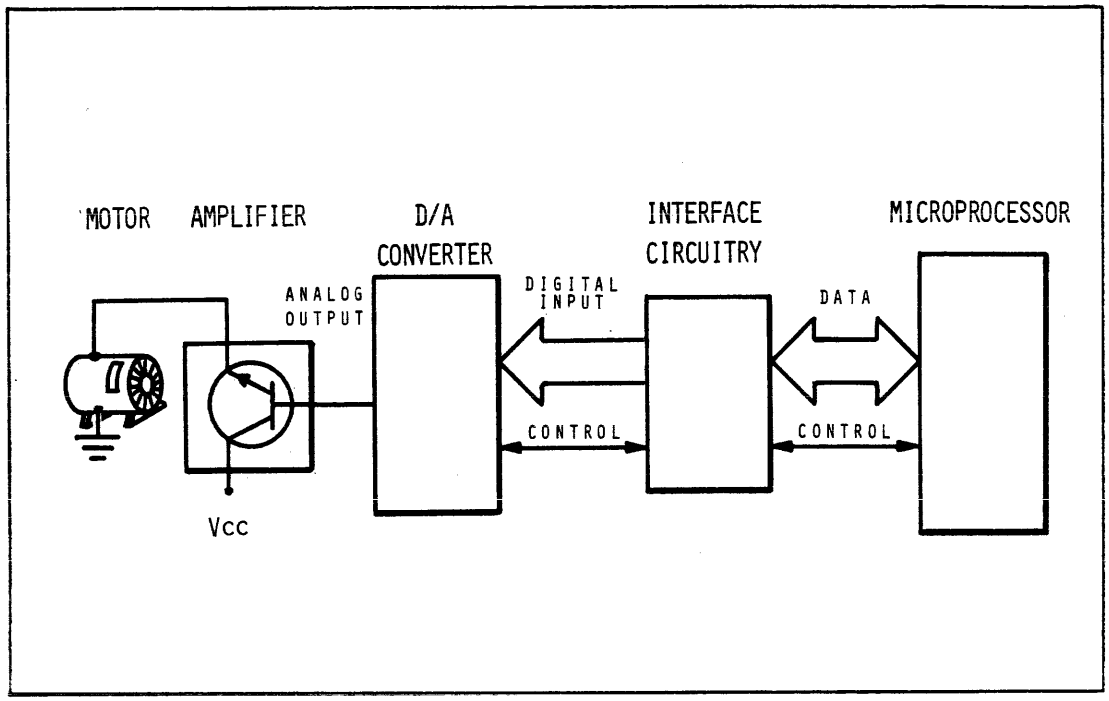


Figure 6-5. Converting a Digital Control Signal to an Analog Control Signal

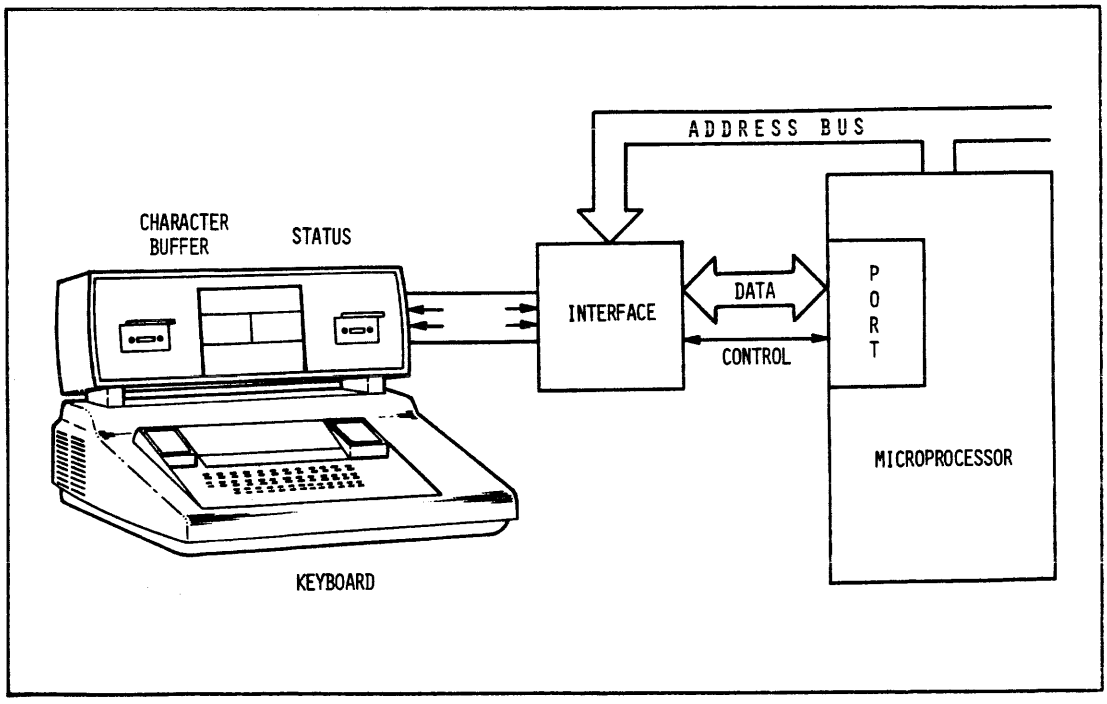


Figure 6-6. A Keyboard Input Device

tell the device to perform some action such as prepare to receive data or to turn on a connected device. The command might, instead, ask the device if a motor connected to it is on, or if a device has data to send. The answer to such questions is usually returned in the form of a device "status": one or more bits indicating the state of the device.

Once this "handshaking" has been completed, data can then be passed under program control according to a predefined protocol.

To better understand the interaction of an I/O device and the processor, take the example of a keyboard illustrated in Figure 6-6. Assume that whenever a key is struck, the ASCII code for the character is stored in a buffer at the keyboard. Whenever there is at least one character collected in the buffer, a status bit is set indicating the buffer is not empty. More than one character can be collected in the buffer and can be removed from the buffer in the same order as entered. Such an arrangement is called a FIFO or a "first-in, first-out" arrangement.

As the program in the system's memory is running, the program will periodically check to see if the keyboard has a character in its buffer (a process called "polling"). To make this check, the program must be able to refer to the device. Usually the system designer assigns an I/O port address to each device by which the program can reference the device. Whenever the program causes this unique address to appear on the address bus in conjunction with a given control signal, the interface circuitry completes the connecting link between the processor and the device. The instruction that causes this connection to take place may be a special I/O instruction in the processor's instruction set or it may be a type of instruction which is also used for memory-to-processor data transfers. In any case, the important point is that the program, and not the device, inititated the interaction.

The advantage of program-controlled I/O is that a minimum of interface circuitry is usually required. It also allows for a wide range of I/O speeds in the low- to moderate-speed range.

The disadvantages are that it is limited to the slower speed ranges and that the transfer of data is dependent upon the program initiating the interaction. High-speed devices or devices having data of immediate urgency for the system may not be able to tolerate these limitations.

Interrupt-Driven I/O

In the previous example of employing program-controlled I/O for the keyboard input, there is the requirement that the processor's program must ask for a character before the keyboard can send one. The possibility exists that the program may become involved in other tasks while numerous characters are being entered from the keyboard and collected in the buffer. There is the hazard that, if the program

fails to ask for input often enough, the buffer will become full and characters will be lost. A careful system analysis might prove this risk to be nonexistent. On the other hand, if the danger proves to be real, a solution could be the use of interrupt-driven I/O.

With interrupt-driven I/O, a device does not have to wait for the processor to poll it before it can indicate characters are ready to be sent. Now, the keyboard device can immediately alert the processor whenever a character is in the buffer. No matter what the processor's program is doing at the time when a character is entered on the keyboard, the device can directly ask for the processor's attention through an interrupt.

Most processors receive interrupt signals through one or more control lines into the processor. These interrupts are sent by external devices to request the processor to suspend what it is currently doing in order to service the device initiating the interrupt.

Figure 6-7 shows a typical interrupt operation. When an interrupt request is detected by the processor, it normally saves enough information to allow a return to the current (interrupted) program and then transfers control to an interrupt service routine to process the interrupt. When the interrupt processing is completed, the interrupt service program allows the processor to return control to the interrupted program, at the point of interruption.

During transfer of control to an interrupt service program, the TMS 9980A processor saves information to provide for an orderly return to the interrupted program when the interrupt processing is completed.

With nearly all processors that have interrupt capability, the logic of the processor automatically saves the contents of the program counter before transferring control to the interrupt service program. (The transfer of control is made by putting the start address of the interrupt service program into the program counter.) It is usually necessary, however, to save more than the contents of the program counter. Such things as the contents of the status register and the contents of the working registers must also be saved for later restoration, because the interrupt service program will be using these registers for its own data and the status register will change as the interrupt service program is running. To return control to the interrupted program at the point of interruption, the program must have the same contents in its status register and working registers as before the interrupt, or the results will be erroneous. It must also resume execution with the same program counter value it had when it gave up control.

Depending upon the processor used, the saving of information in addition to the contents of the program counter may be the duty of the interrupt service routine, or, this can be performed by the logic of the processor.

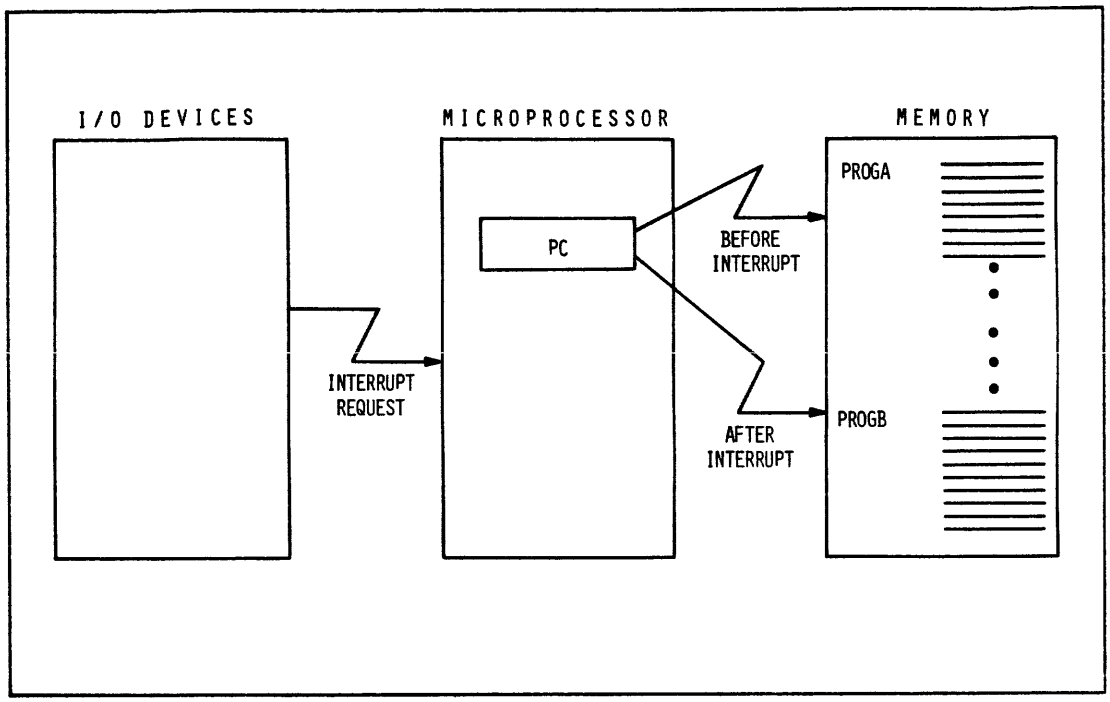


Figure 6-7. Interrupt Initiation by an I/O Device

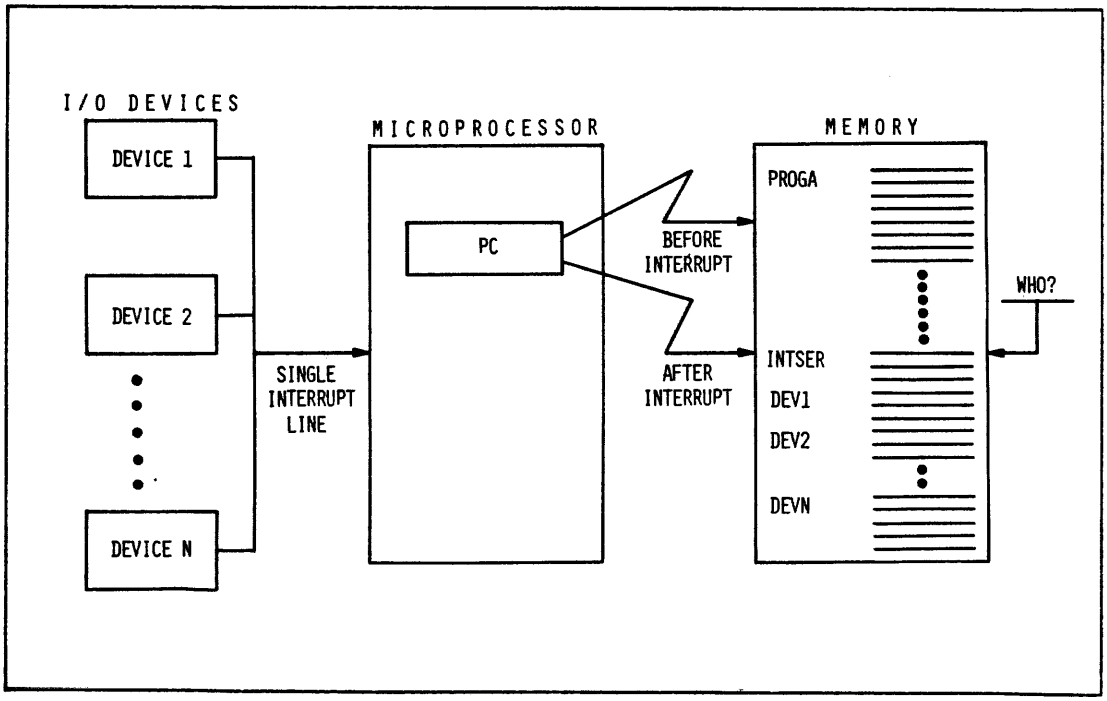


Figure 6-8. Single Interrupt Level

The TMS 9980A processor, specifically, saves this information automatically. The contents of the program counter and the contents of the status register are automatically saved. The contents of the working registers of the interrupted program are also automatically saved by switching to a different set of working registers to be used by the interrupt service routine.

Once the interrput service program is in control, the interaction with the I/O device proceeds much as it does in program-controlled I/O. The distinguishing point is that it was the I/O device, rather than the program, that initiated the interaction.

When the interaction has been completed, the interrupt request is removed, and control is returned to the interrupted program.

The question may arise: "Where is the interrupted program's data saved?" Some processors save it in a dedicated area of memory organized as a "last-in, first-out" (LIFO) stack. In such a case, the processor often maintains a stack pointer to keep track of the last entry on the stack as the stack swells and contracts with data entry or removal. A few processors have a limited stack residing in the processor rather than in memory. Other processors employ other methods for saving an interrupted program's data. The TMS 9980A employs a linked-list structure instead of a stack, and the interrupted program's workspace pointer, program counter, and status register are saved in the interrupt service routine's workspace.

In its simplest form (as shown in Figure 6-8), the interrupt structure of a processor causes all interrupts to be transferred to a common interrupt service program which must then decide which device caused the interrupt.

The TMS 9980A provides for prioritized, vectored interrupts. With a vectored interrupt structure, devices are assigned interrupt codes or "levels." Each interrupt level has a unique address in memory. This unique address associated with each interrupt level contains an address to which control is transfered when an interrupt at that level occurs. In other words, the address assigned to each interrupt level contains a pointer address (vector) to the interrupt service program. See Figure 6-9.

Two pieces of information are stored in an interrupt vector: the new contents for the workspace pointer and the program counter. Once the processor retrieves the vector contents, the old workspace pointer, program counter, and the state of the status register just before the interrupt are all saved automatically in the new workspace.

The advantage of enhancing interrupt capability by adding vectors is that it is no longer necessary for an interrupt service program to determine which device caused an interrupt, provided no more than one device is assigned to a level.

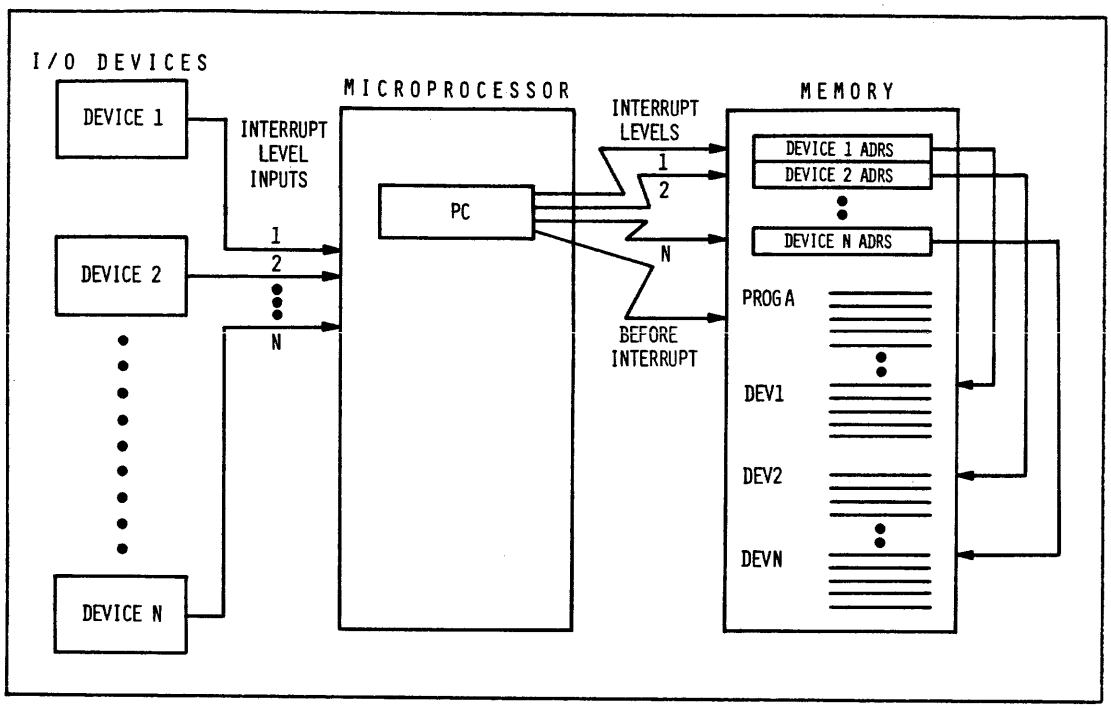


Figure 6-9. Vectored Interrupts

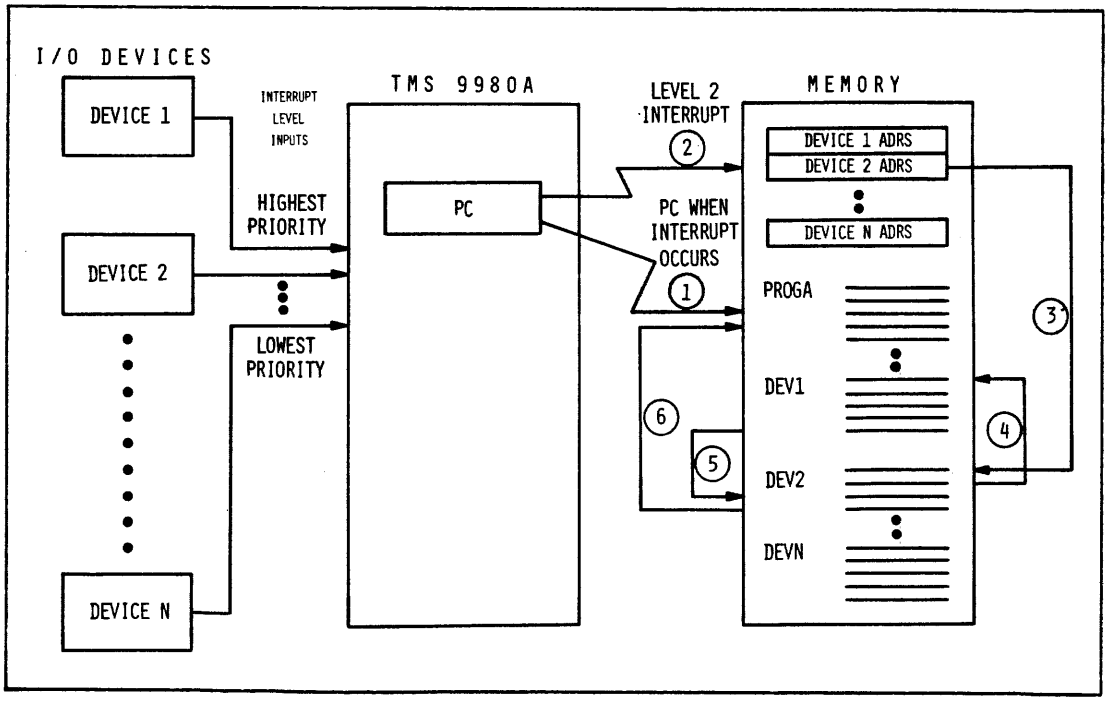


Figure 6-10. Priortized Vectored Interrupts

Additional sophistication has been added to vectored interrupts by the TMS 9980A by assigning a priority to each interrupt level to establish a relationship of importance among the levels. With this arrangement, a currently executing interrupt service program at one level might be interrupted by another interrupt at a higher level, but not by a level of lower priority. Refer to Figure 6-10.

In this figure, program A (indicated in the figure as PROGA) is in control and the program counter (PC) points to the current instruction within the program. The PC in pointing to the next instruction to be executed (circle 1) when device 2 initiates a level-2 interrupt request. The TMS 9980A identifies the interrupt request as level-2 and uses the vector address for a level-2 request to point the PC (change its contents) to the interrupt service routine (DEV2) that processes the level-2 interrupt (circle 3).

While the level-2 interrupt service routine is executing, a level-1 interrupt occurs which causes the TMS 9980A to take control away from the level-2 interrupt service routine and give control to the level-1 interrupt service routine (circle 4).

When the level-1 interrupt service routine (DEV1) completes its execution, control is returned to the level-2 interrupt service routine at its point of interruption (circle 5). The level-2 interrupt service routine completes its execution and control is returned to PROGA (circle 6) at the point where it was interrupted. In this case, a level-1 interrupt request was allowed to interrupt a level-2 interrupt service routine because a level-1 interrupt request is defined by the TMS 9980A to be of higher priority than a level-2 interrupt.

By employing interrupt-driven I/O, there is the advantage of allowing a device to initiate interaction with the processor, but a penalty is paid in the form of a requirement for a small degree of additional interface complexity. Also, if there are critical program-controlled timing loops in the application, the timing will be in error if an interrupt occurs during the execution of the loop. Either program-controlled timing loops must not be used or else interrupts must be prevented from occurring during the loop.

Direct Memory Access I/O

The third general category of I/O is direct memory access, (DMA). DMA is useful for very high-speed data transfers. As shown in Figure 6-11, the possibility of higher transfer speeds with DMA results from removing the processor as the middle element between data tranfers to or from I/O devices and memory.

In both program-controlled I/O and interrupt-driven I/O, data going to and from an I/O device and memory is passed back and forth by the processor under program control. With DMA, a dedicated controller is delegated the chore of controlling the system's buses

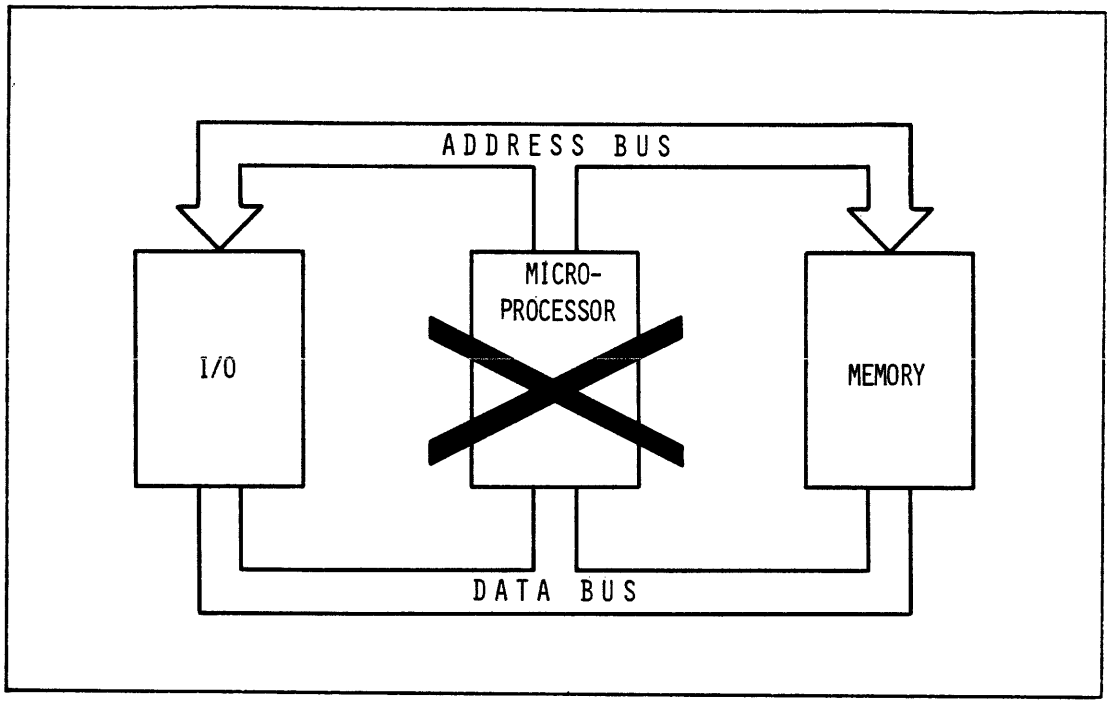


Figure 6-11. A DMA Controller Eliminates the Microprocessor for Data Transfers

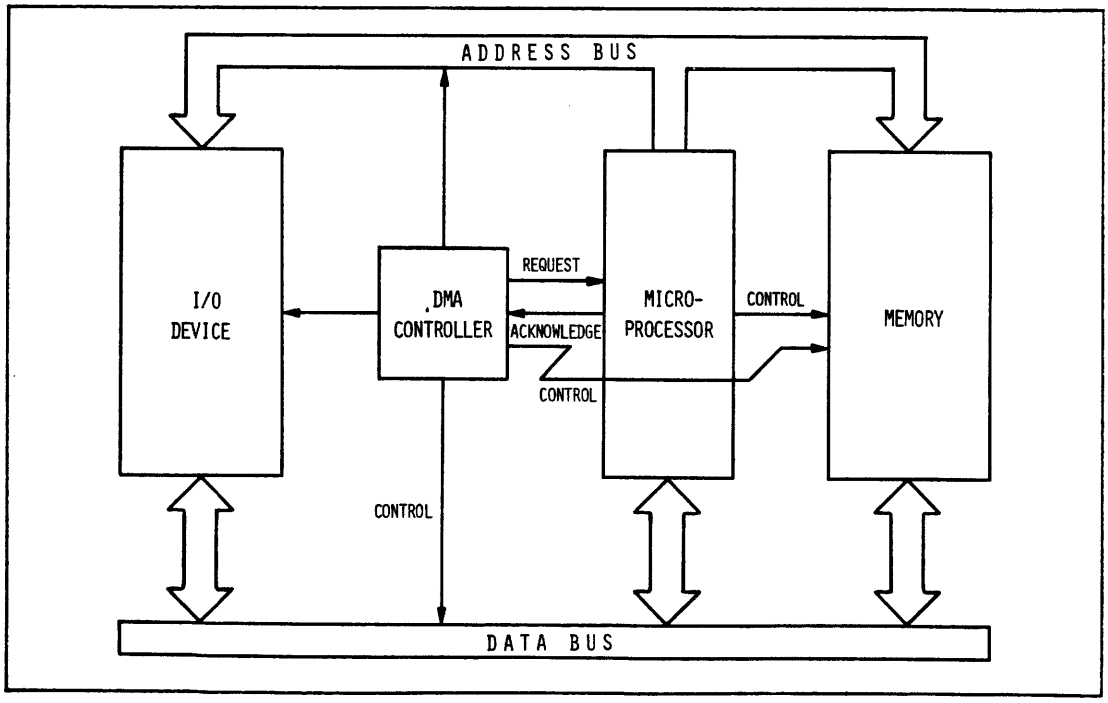


Figure 6-12. A DMA Controller in a Microprocessor System

to pass the data. Because the DMA controller is designed for the specific task of transferring data and not data processing and the other more general functions possessed by the microprocessor, the DMA controller can be very efficient at passing data, which makes very fast data transfers possible.

To facilitate a DMA transfer, the DMA controller must assume control of the data bus, address bus, and the memory control signals. Data will be passed on the data bus while the address bus is changed to reflect the address of the data coming from memory or the address to which it will go in memory. Normally, with DMA I/O, if more than one data word is passed, the words are in contiguous blocks; that is, they are located in sequential memory locations.

The processor also uses and controls the address and data buses. Of course, it is not possible for both the processor and the DMA controller to use the buses at the same time. The contention for use of the buses must be resolved so that only the DMA controller or the processor, but not both, is in control at any one time.

Various methods can be employed to accomplish this. The simplest method is to allow the DMA controller to alert the processor when it wishes to use the buses. The processor relinquishes control by sending an acknowledgement signal to the controller, and then suspending operation and placing its own interfaces to the buses into the high-impedance (or off-line) state. The DMA controller therefore has free access to the buses and can control the data transfer.

Once the piece of data or block of data has been passed, the DMA controller relinquishes control by removing its control request. Upon recognizing the request removal, the processor takes back control and proceeds with its operation. Figure 6-12 depicts the relationship of the DMA controller and the processor in the system.

A DMA I/O operation can be initiated by a program. Refer to Figure 6-13 where a system employs a high-speed disk as a peripheral device. At some point in the program, it becomes necessary to transfer a block of data from a sector on the disk to a memory buffer. The program causes the processor to pass the request for a data transfer along with an information packet to the DMA controller. The information informs the DMA controller where the information is on disk, where the information is to be put in memory, and the number of pieces of data to be transferred. It might be assumed that all disk data transfers would be in block sizes the size of a disk sector.

The request for a DMA transfer and the information packet could be passed to the DMA controller employing program-controlled I/O.

The disk controller has logic to recognize the disk address of data under the read head. As the selected disk address comes rotating under the head, the DMA controller requests and takes control of the address and data bus and passes the data directly

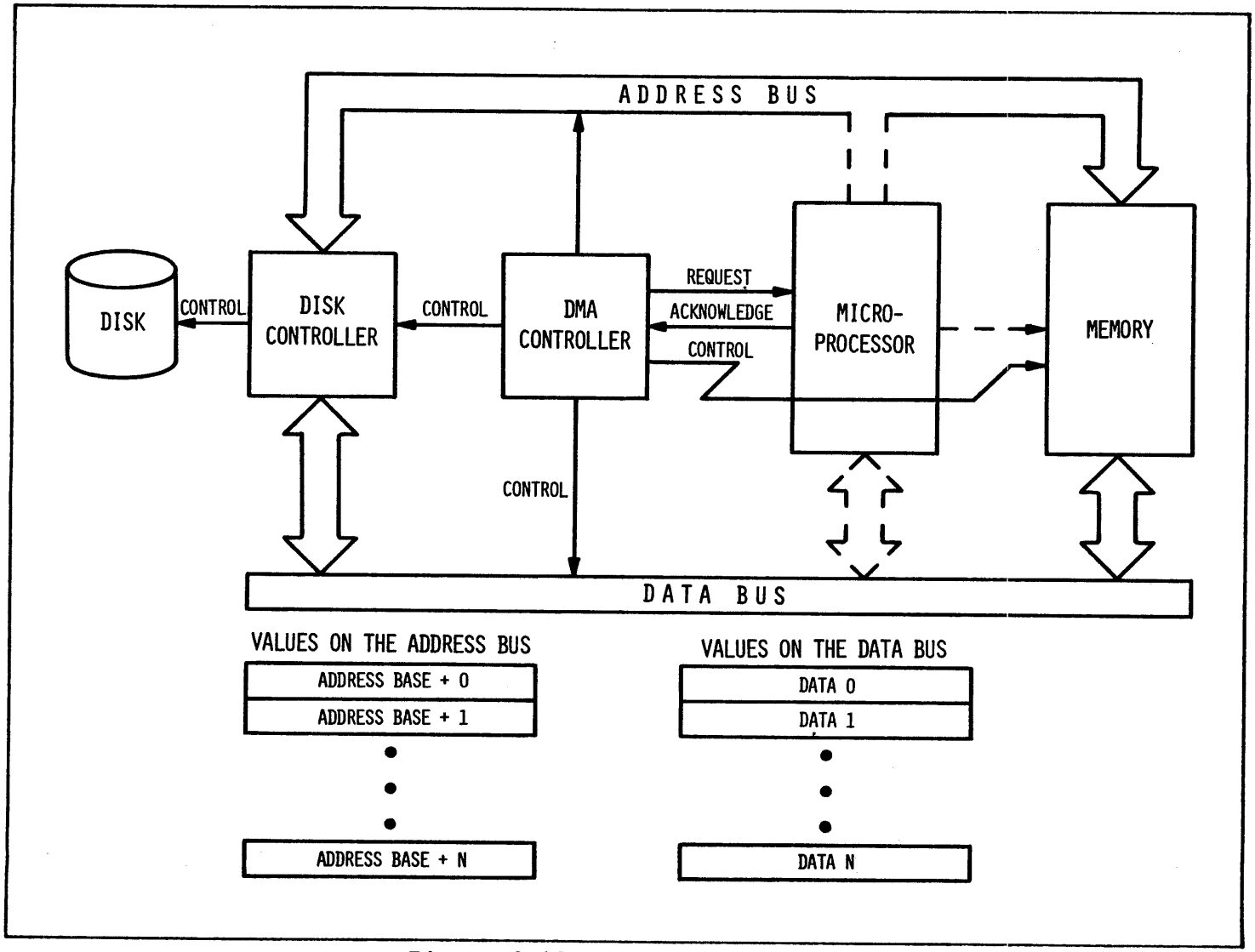


Figure 6-13. A DMA Controller with a High Speed Disk

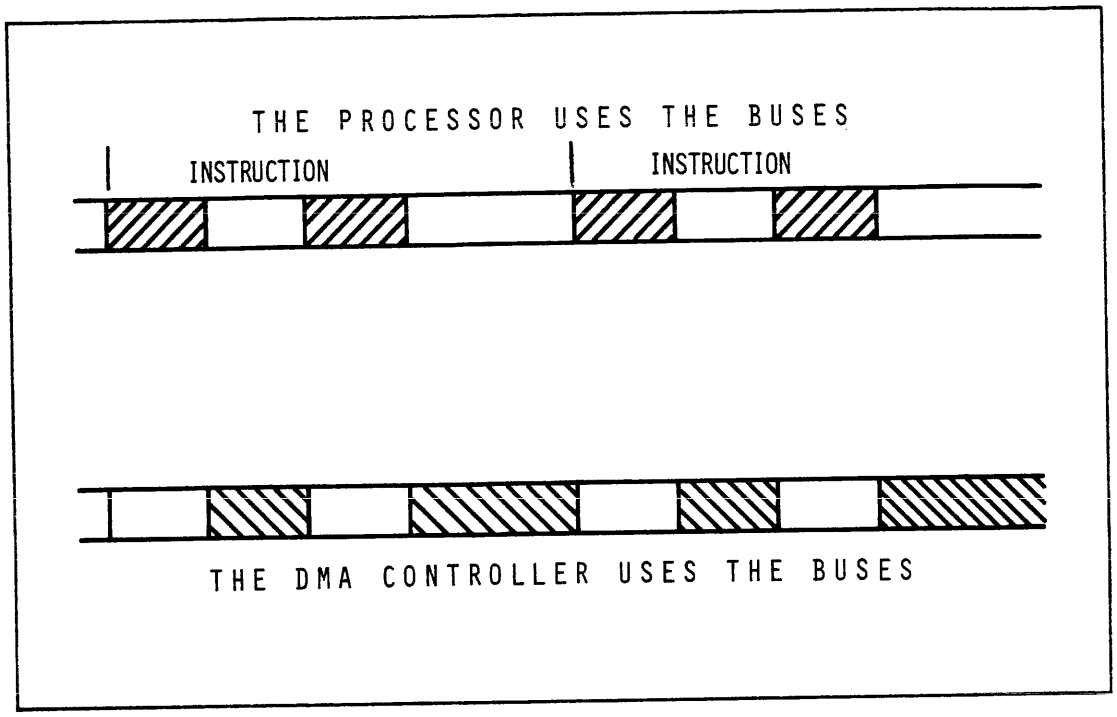


Figure 6-14. A DMA Controller Shares the Buses

into memory as it is plucked from the disk. To do this, the DMA controller sets the address bus to the memory location specified in the information packet and passes the first data word from the disk over the data bus to that memory location. It then increments the address on the address bus to the next sequential memory location and passes the second data word over the data bus to memory. Again, it increments the address bus for the next data transfer, and so on, until the specified number of data words is transferred. Often the data read from the disk is first buffered (that is, the data taken from the disk is placed in a memory buffer) before the DMA transfer.

Up to this point, a simple means of DMA has been discussed. While a DMA I/O operation is in progress, the processor is suspended, performing no processing. There are other methods of DMA I/O which allow a DMA transfer to proceed concurrently with processing. As can be seen from Figure 6-14, when a processor is executing, it requires the use of the address and data buses only a portion of its time. Other techniques take advantage of this "dead time" on the buses to permit the DMA controller to insert data at those moments.

Under these circumstances, a program can initiate a DMA request and then perform other processing while the DMA controller fulfills the request. The controller can be designed to alert the program with an interrupt to indicate that the transfer has been completed

and that the controller is available for additional transfers. With such an arrangement, the throughput of a system can be increased.

The advantage of DMA I/O is that it provides very high data transfer rates. Its disadvantages are that it requires extensive additional logic and is more complex to design.

Summary of I/O Categories

Three general categories of I/O have been discussed: program-controlled I/O, interrupt-driven I/O, and DMA I/O. Program-controlled I/O is the simplest method to implement but is limited to the lower speed ranges for data transfers. Interrupt-driven I/O has the advantage of allowing a device to initiate interaction with the processor and its program but requires more extensive hardware to implement and may affect program timing loops. The third category of I/O, DMA, allows very fast data transfers between memory and I/O devices but has the disadvantage of requiring much more complex logic in the I/O periphery.

Having discussed I/O in general, the attention now turns specifically to the I/O options available with the TMS 9980A microprocessor and the University Board.

6.4 OVERVIEW OF 9900 FAMILY I/O OPTIONS

The 9900 family of microprocessors, which includes the TMS 9980A on the University Board, allows use to be made of all three of the I/O categories discussed in the previous section. Program-controlled I/O is possible using either the data bus for data transfers or by using an alternative I/O port called the Communications Register Unit, or CRU. Interrupt-driven I/O is also possible. The architecture of the 9900 family provides for prioritized, vectored interrupts. The third I/O category discussed, DMA, is also available as part of the 9900 family architecture.

During the following discussion, reference is made to specific pins on the TMS 9980A. For a more complete description of these pins, refer to the TMS 9980A pin description in Appendix E.

Options Employing the Data Bus

There are three methods of I/O implementation available with the 9900 family:

- Memory-mapped
- DMA
- CRU.

The first two allow data transfers to be made to and from the I/O periphery on the data bus. (Recall that the data bus is also used for data transfers between the memory and the processor.)

Memory-Mapped. With memory-mapped I/O, an I/O device is treated as one group of memory locations. This allows any instruction with memory addressing capability to be used for addressing an I/O device. In turn, the I/O device must assume many of the characteristics of a memory device. The I/O device must recognize its address on the address bus when it occurs in conjunction with the "memory enable" (MEMEN-) control signal. If it is a device capable of both sending and receiving data, it must discriminate between the processor's read cycle and write cycle and must respond appropriately by placing data on the data bus or taking data off the bus. Figure 6-15 shows that the MOV instruction might be used for addressing both memory and I/O. Naturally the memory and the I/O device utilize different address values.

Generally speaking, most I/O devices have much slower response times than memory devices. Whereas memory devices normally provide for accesses in the nanosecond range, I/O devices usually operate in the microsecond or millisecond range.

This discrepancy in speed can be resolved by the TMS 9980A's READY memory control signal (see Figure 6-16) which was discussed in the previous chapter. If the I/O device has a response time slower than the access time for the processor, the interface logic of the I/O device must not allow the READY signal to become active until data is removed from or placed on the data bus. At that time, the device can raise the READY signal, just like slower memory.

DMA. The second I/O method employing the data bus is DMA. Recall that DMA is the fastest data transfer method because the processor is eliminated in the data transfer. Data is passed back and forth between an I/O device and memory on the data bus under the control of a DMA controller. Because the DMA controller and the processor both require control of the address and data buses, the contention for control at the same time must somehow be resolved.

With the TMS 9980A this contention is resolved very simply as shown in Figure 6-17. When the DMA controller wishes control of the buses, it alerts the processor by activating the hold request input line (HOLD-) to the processor. Upon completing its current instruction and upon detecting the HOLD- input, the processor suspends its operation, puts its interface to the address bus and data bus into the high-impedance or "off-line" state and sends the "hold acknowledgement" (HOLDA) signal to the DMA controller indicating the buses are now available to it. Upon detecting the HOLDA signal, the DMA controller takes control of the buses and commences the data transfer.

When the transfer is complete, the controller removes its request by deactivating the HOLD- signal. The processor sees the request drop, deactivates the HOLDA output to the controller, reassumes control of the buses, and proceeds with its operation.

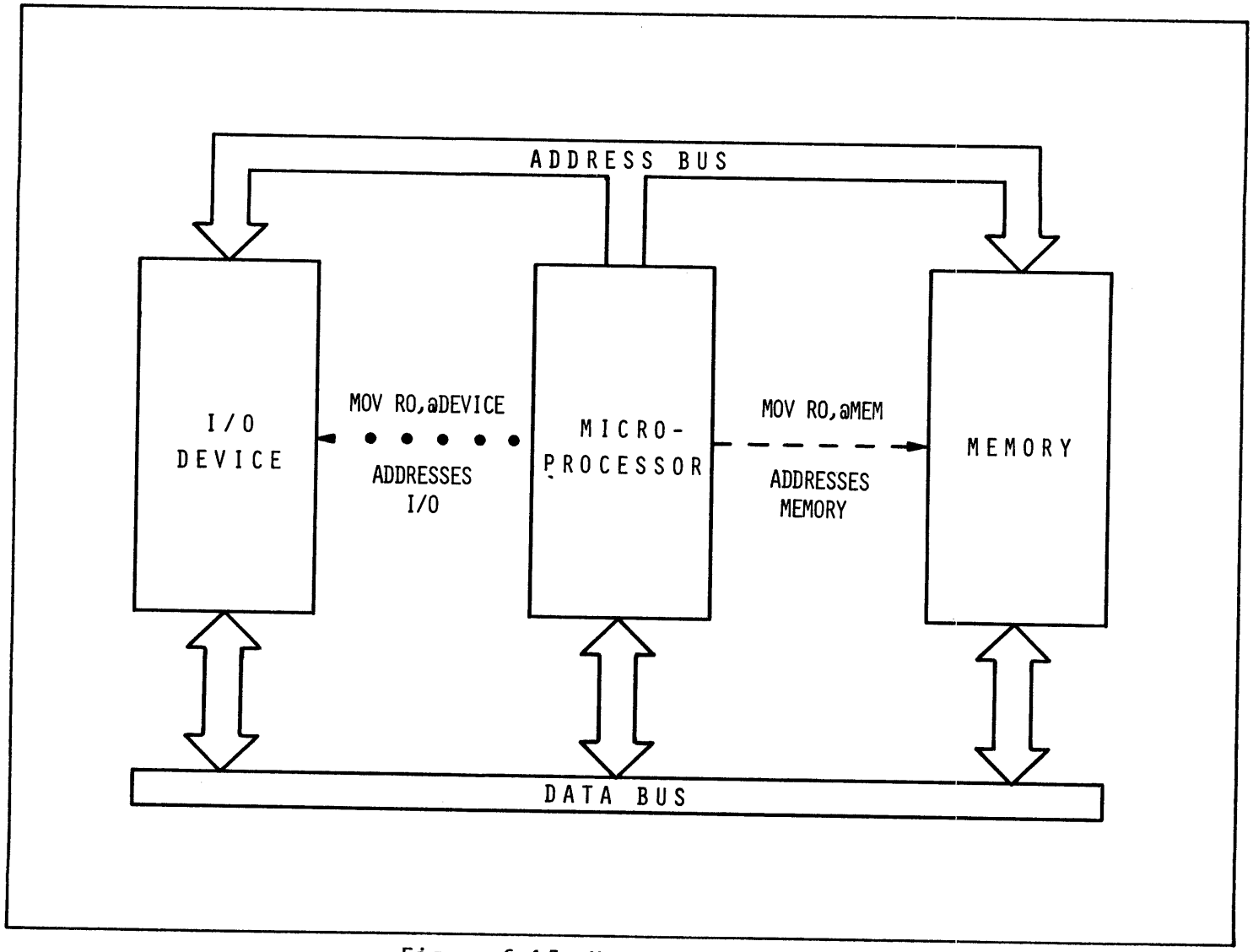


Figure 6-15. Memory Mapped I/O

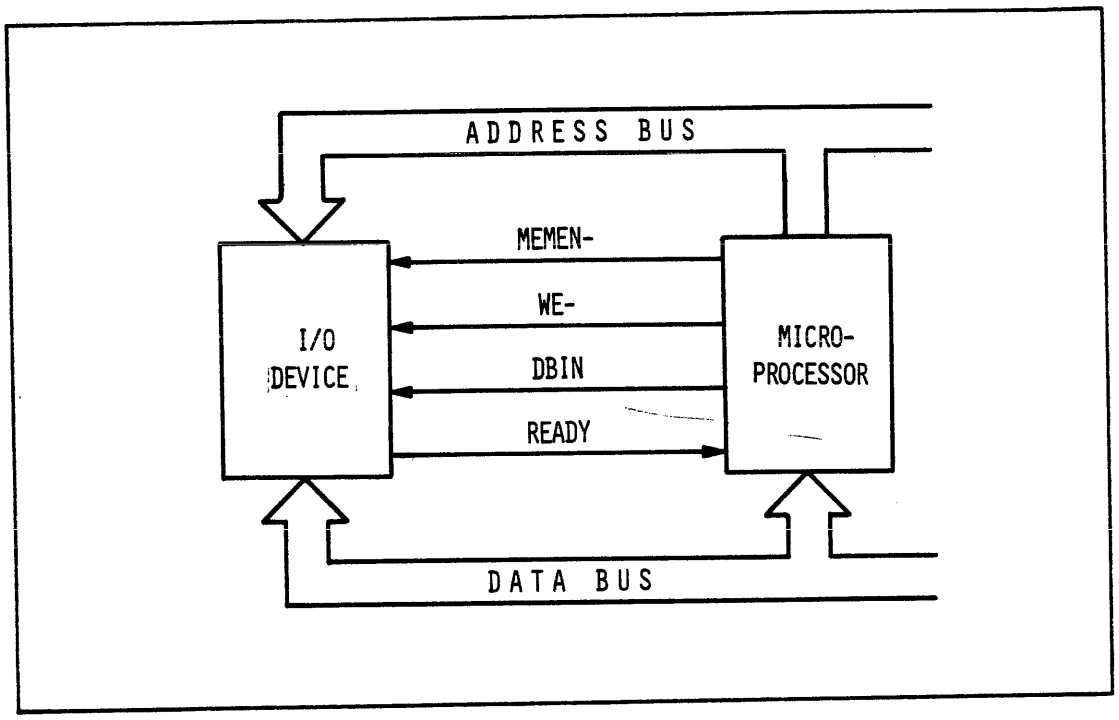


Figure 6-16. An I/O Device Simulating Memory

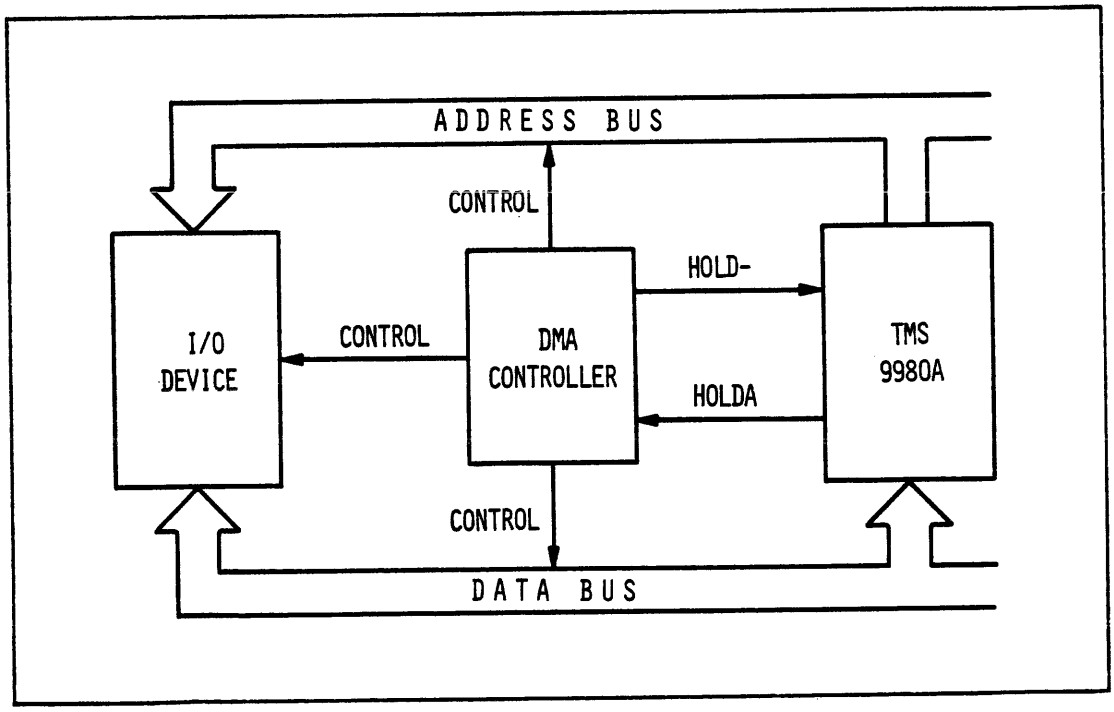


Figure 6-17. Resolving Bus Contention Between the TMS 9980A and a DMA Controller

Although the TMS 9980A microprocessor on the TM 990/189 has DMA capability, the additional circuitry necessary to implement DMA does not exist on the board. Additional circuitry could be added by the user to allow direct memory access to memory off-board the TM 990/189.

The two I/O methods discussed thus far, memory-mapped and DMA, have one thing in common. They both use the data bus for data tranfers. In both cases data is passed in parallel, that is, several bits at one time.

The ability to pass multiple bits of data at the same time on the data bus can be advantageous. There are also some disadvantages. With memory-mapped I/O, the I/O device must be made to simulate a memory device which often requires additional circuitry. The designer must demultiplex data on the data bus intended for memory from data intended for an I/O device. Also, addresses assigned to the I/O devices are no longer available for memory, so the memory-addressing range of the system is reduced.

Communications Register Unit (CRU)

Problems of memory-mapped I/O and DMA can be largely overcome by the use of a separate port, used only for data transfers to or from I/O devices. Such a dedicated port exists with the 9900 family. It is called the Communication Register Unit (CRU). The CRU is part of the processor's architecture. Associated with the CRU are three pins on the microprocessor. One pin is used for data input (CRUIN), another for data output (CRUOUT), and the third for data timing strobes (CRUCLK). The CRU connection between the TMS 9980A processor and the I/O is shown in Figure 6-18.

6.5 CRU CONCEPT, ADDRESS SPACE, AND OPERATION

CRU Concept

The CRU is a distinguishing feature of the 9900 family architecture and is a dedicated I/O port. It is composed of internal logic, including a shift register, and interfaces to the I/O via three pins on the TMS 9980A microprocessor. The pin CRUIN is used to serially transfer one or more bits into the processor. The CRUOUT pin is used to serially transfer one or more bits to an I/O device from the processor. This output data appears on the A13 address line of the TMS 9980A which is CRUOUT for the output CRU instructions. (Address line A13 is not used for addressing purposes during a CRU operation.) The CRUCLK pin is used as a timing strobe to coordinate data transfers.

As data is input or output, addresses appear on the address bus to select the particular I/O bit address for the data.

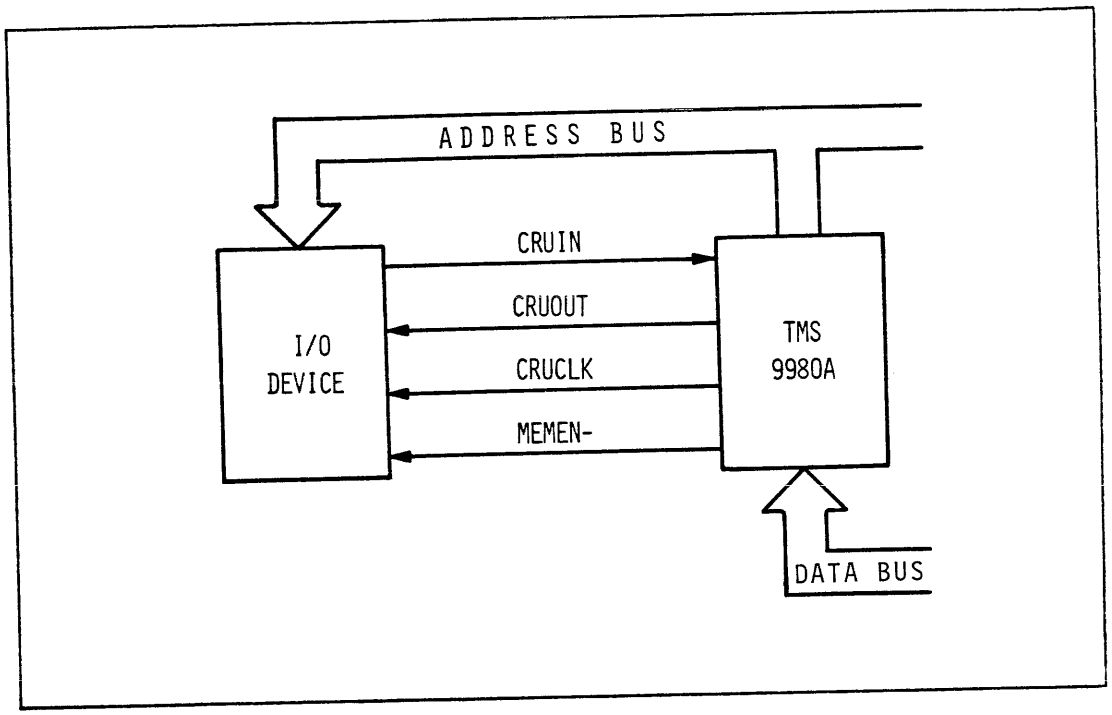


Figure 6-18. The CRU Connection

Data in and out via the CRU is in bit serial form. This means that in a multibit data transfer, the bits comprising the data are passed along one behind the other. Figure 6-19 shows the relationship between serial and parallel data.

The CRU offers several advantages. First, it is a separate data port from the data bus, which alleviates some of the design problems of having to demultiplex data intended for memory from data intended for I/O, as would be necessary with memory-mapped I/O. Secondly, the CRU permits the use of readily available, relatively inexpensive TTL components for interfacing. A special LSI peripheral component is not necessary; however, there are special CRU peripheral devices available the use of which may be advantageous in systems with a moderate to complex degree of I/O. The most important advantage of the CRU, though, is that it permits single bit addressing. This feature is especially important in a control environment where a single bit can be used to control a device. A single bit is all that is necessary to turn a device on or off or to test if a device is on or off.

Although this "bit diddling" capability is a prime advantage of the CRU, multiple-bit transfers (up to a maximum of 16 bits) can be accomplished with one instruction.

The CRU will be used in many of the examples that follow in the book, so a closer examination to explore its operation is necessary.

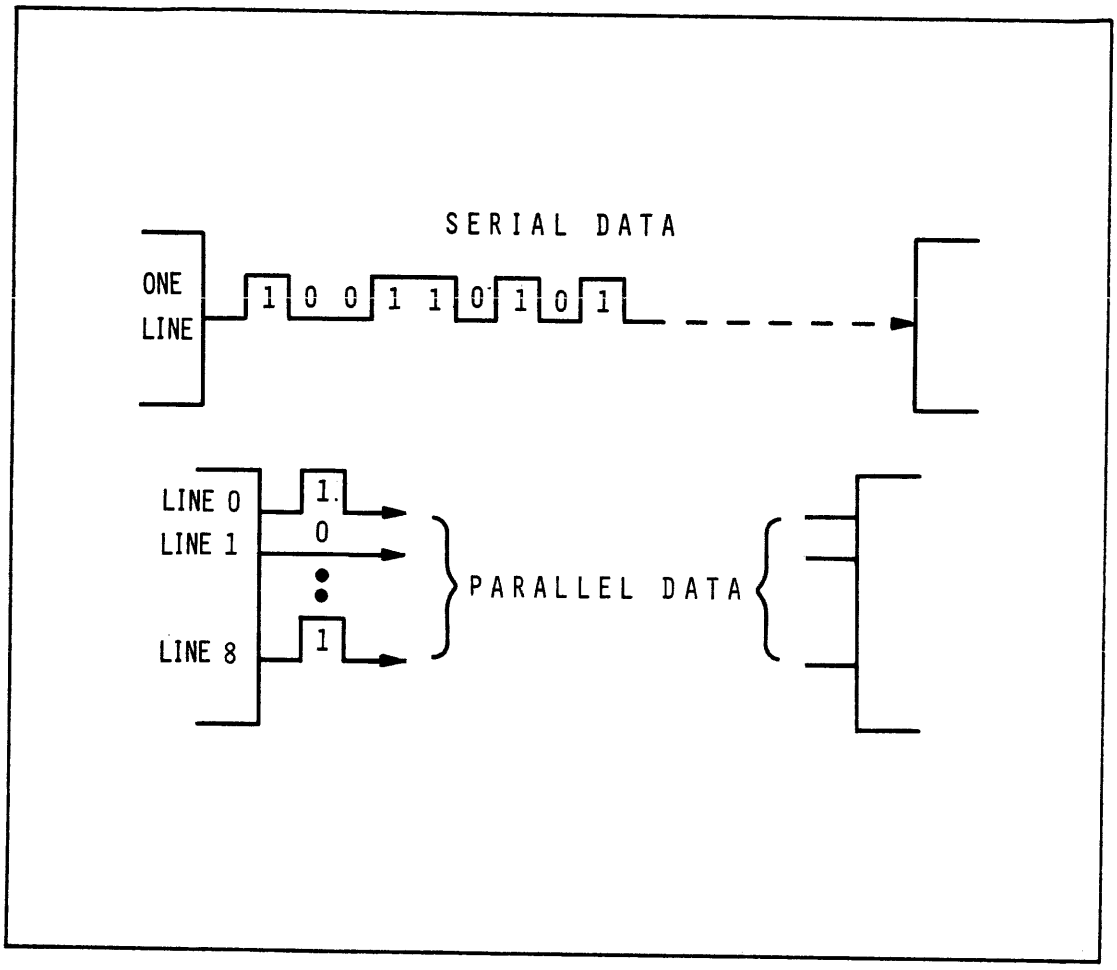


Figure 6-19. Serial Data versus Parallel Data

CRU Address Space

It was mentioned that an I/O device must recognize when it is being accessed by the processor (for either input or output). A common method to do this uses interface logic with the device so that it responds to a specific address on the address bus. The interface logic can be designed so that response is made only to addresses when they appear in conjunction with another control signal. This qualification allows one address to be used for both a memory location and a device location.

In all 9900 family microprocessor devices, the MEMEN- signal assumes this qualification role. If active low, the addresses are intended for a memory operation. If inactive high, any address appearing on the address bus may be regarded (though not necessarily) as a CRU address. In Figure 6-18, the MEMEN- signal is connected to I/O devices employing the CRU. Whenever a CRU I/O operation is being performed, this signal is inactive.

An output device (that is, a device to which data is sent) connected to a 9900 family processor via the CRU normally is made to respond only to its chosen address in conjunction with the CRUCLK strobe. Specifically, the TMS 9980A processor on the University Board possesses a range of 2K (2,048) bits of I/O addressing via the CRU. Notice that the CRUCLK strobe is not required for CRU input operation. During a noninput instruction if a device responds to the appearance of its address on the address bus and attempts to input data on CRUIN, the processor would simply ignore the data.

When a CRU instruction is executed (either input or output), the bit address of the selected device appears on the address bus on lines A2 through A12. (This is 11 bits of addressing, and since 2^{11} equals 2048; the addressing range of the CRU is 2K bits.)

The CRU bit address appearing on the address bus is developed from a base value in workspace register 12. In the case of a single-bit CRU operation, a signed displacement in the instruction is added to the base value to form the individual bit address.

With a multibit CRU operation, no displacement is specified in the instruction and the CRU bit address that appears initially on the address bus is the same as the base value in R12. Then to select each subsequent bit, the processor automatically increments the initial CRU address sequentially by one. These sequential bit addresses appear one after the other on the address bus (lines A2 through A12).

The CRU bit address consists of 11 bits. Workspace register 12 (which contains the CRU base address) is, of course, 16 bits wide. Which 11 bits of these 16 are used as the CRU base address? The CRU architecture is designed to use bits 4 through 14 of register 12. This means that when an absolute CRU address (or hardware address) is loaded into register 12, it must be offset (that is, moved) one bit position to the left. See Figure 6-20.

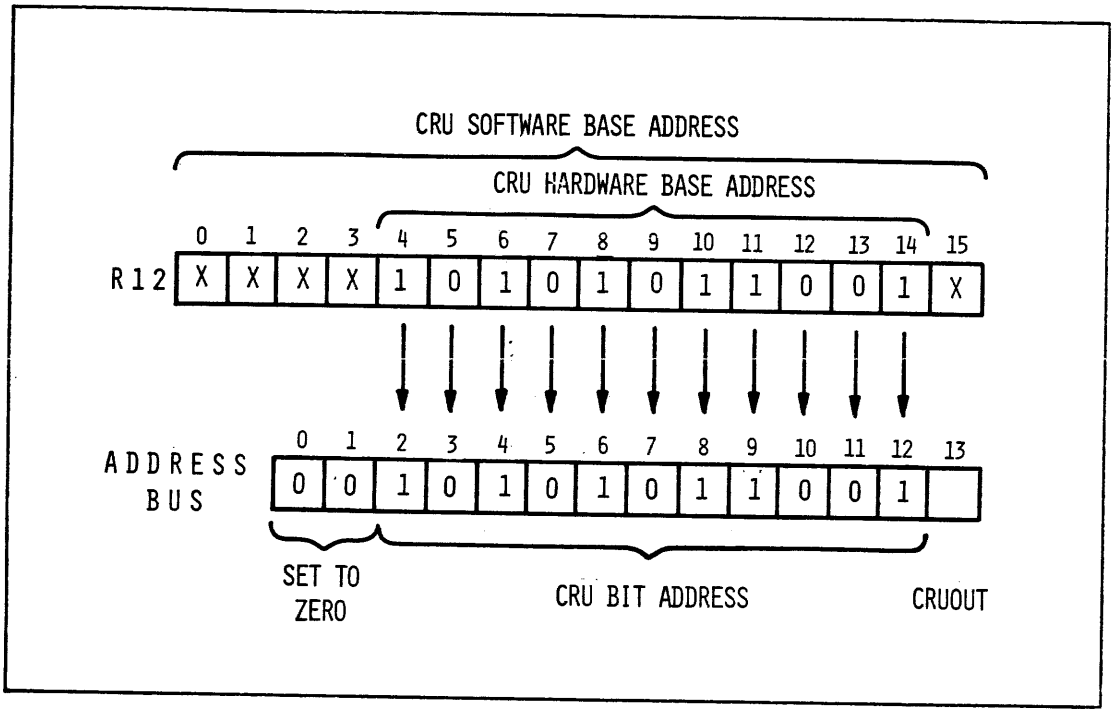


Figure 6-20. CRU Address Development for Multi-Bit CRU Instructions

The entire 16-bit contents of R12 containing the displaced (shifted) value is called the CRU software base address and the 11-bit address value that appears initially on the address bus to select a CRU bit in the periphery is called the CRU hardware base address.

As an example, suppose a programmer desires to establish a CRU software base address in register 12 for the CRU bit having the CRU hardware base address $014E_{16}$. The following two instructions accomplish this.

```
LI    R12,>14E
SLA   R12,1
```

A left shift of one bit position is equivalent to a multiplication by two, therefore the single instruction,

```
LI    R12,>29C
```

accomplishes the same thing since $29C_{16}$ is $14E_{16}$ times 2.

On the address bus, address lines A0 and A1 are forced to ZERO for CRU input and output operations. Address line A13 becomes CRUOUT and is used to output one or more bits serially during a CRU output operation.

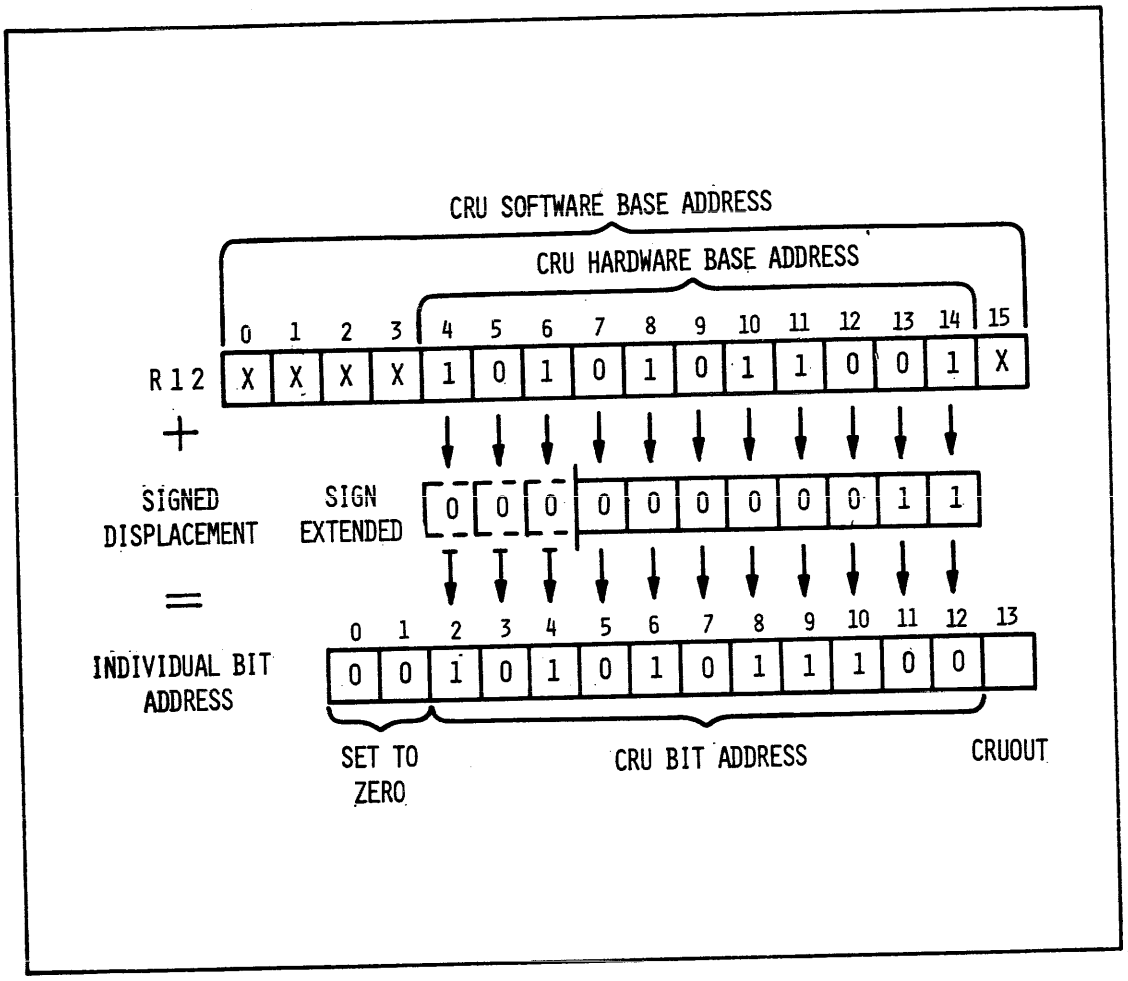


Figure 6-21. Address Development for Single Bit CRU Instructions

For the CRU instructions used in single-bit addressing, an 8-bit displacement is added to the CRU hardware base address. This 8-bit displacement value is taken from the low-order eight bits of the single-bit CRU instruction. The displacement is treated as a signed value with a range of -128 to +127. Refer to Figure 6-21.

CRU Operation

Having established a basic understanding of the concept of the CRU and its address space, a more detailed examination of the CRU's operation is now possible.

	SINGLE-BIT	MULTI-BIT
INPUT	TB	STCR
OUTPUT	SBO SBZ	LDCR

Figure 6-22. Summary of CRU Instructions

In the TMS 9980A instruction set, there are five instructions devoted to data transfers via the CRU. Three of these five are used for single-bit data transfers:

- TB test bit
- SBO set bit to one
- SBZ set bit to zero.

The other two are used for transfers of one to 16 bits:

- LDCR is used to transfer a group of bits to the I/O periphery.
- STCR is used to input a group of bits from the I/O.

Of the three single-bit CRU instructions, two are used for output. The SBO instruction outputs a logical ONE or sets an I/O bit. The SBZ instruction outputs a logical ZERO or resets an I/O bit. The TB instruction inputs a single bit into bit 2 of the status register (the EQUAL status bit). Following the TB instruction, the status of the EQUAL status bit can be tested by employing conditional jump instructions to decide if the input bit was on or off.

If the bit is a ONE, then a JEQ instruction will cause a jump; if the bit is a ZERO, a JNE instruction will cause a jump. Figure 6-22 summarizes the three single-bit CRU instructions.

Figure 6-23 shows the implementation of a 16-bit input and 16-bit output interface with the CRU.

Notice that in this implementation only four lines of the address bus, A9 through A12, are required to address the 16 bits of input as well as the 16 bits of output. Address line A9 is used to select the individual octal multiplexor (SN74LS251) or latch

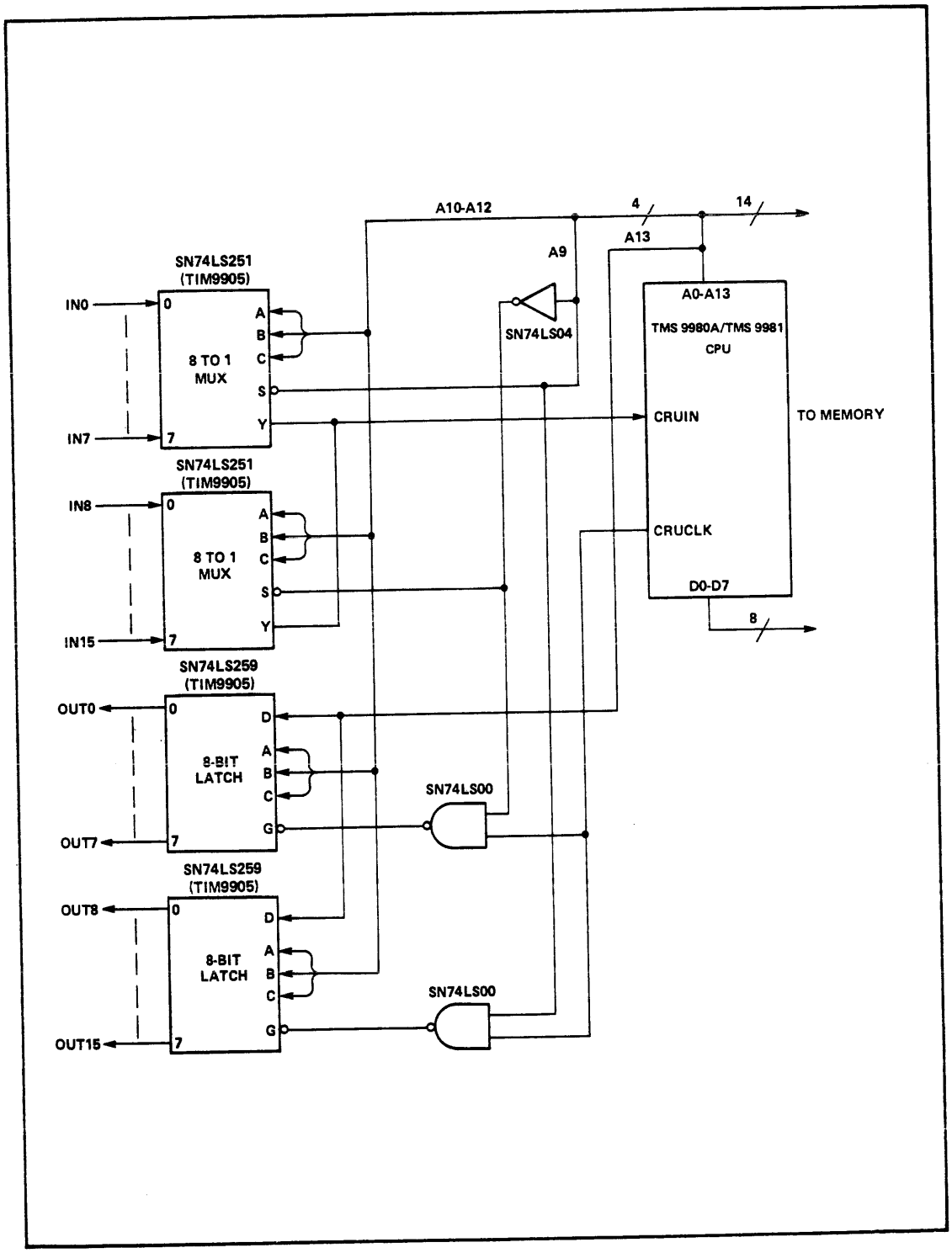


Figure 6-23. TMS 9980A/9981 16-Bit Input/Output Interface

(SN74LS259), and address lines A10 though A12 are used to select the individual input or output bit. Furthermore, notice that the CRUCLK strobe must be present before output data is allowed to latch either one of the octal latches. The requirement that CRUCLK be present before an output takes place allows the use of only four lines of address to address a total of 32 bits of I/O.

With a data-input CRU instruction, the address on the address bus, lines A9 thorugh A12, selects one of the 16 inputs to the pair of SN74LS251's as shown in Figure 6-23. The selected bit is placed on the CRUIN line to the processor. In the case of the single-bit TB input instruction, the state of the bit then goes to status bit 2 (the EQUAL status bit) of the status register where its state can be tested by a subsequent conditional jump instruction. In the case of the multibit CRU input instruction, STCR, the address on address lines A9 through A12 changes successively at fixed intervals so that sequential bit addresses are selected and shifted serially into the processor through CRUIN. Refer to Figure 6-24 for the timing relationship between the sequential addresses and the input data.

With a data output CRU instruction, the address on the address bus, lines A9 through A12, selects one of the 16 outputs from the pair of SN74LS259's as shown in Figure 6-23. The selected bit is placed on the CRUOUT (A13) line in conjunction with the CRUCLK strobe. In the case of the two single-bit CRU output instructions, the SBO instruction sets the selected bit to the ONE state and the SBZ instruction sets the selected bit to the ZERO state. With the multibit CRU output instructions, LDCR, the address on the address lines A9 through A12 changes at fixed intervals along with the CRUCLK strobe as the bits are shifted out serially from CRUOUT to the sequentially addressed latches. Again, refer to Figure 6-24 for the timing relationship between the sequential addresses and the output data.

The LSB of a memory word (bit 15) is transferred to or from the lowest CRU address in a multibit transfer. Figure 6-25 shows what happens with a 16-bit data transfer between a memory location and the CRU. Notice that upon input, the lowest CRU bit address is right-justified in the memory word so that it occupies the least-significant-bit location. Upon output, the LSB of the memory word goes to the lowest CRU bit address.

Sixteen bits are used in this example but a lesser number of bits could have been used. In a data transfer of more than eight bits, the data comes from or is stored right-justified in a 16-bit word. In a data transfer of a byte or less, the data is right-justified in the addressed byte. In storing the input bits using the STCR instruction, unfilled bits in the byte or word are set to ZERO's.

Figure 6-26 shows a 6-bit input from the CRU to memory location $3AC2_{16}$.

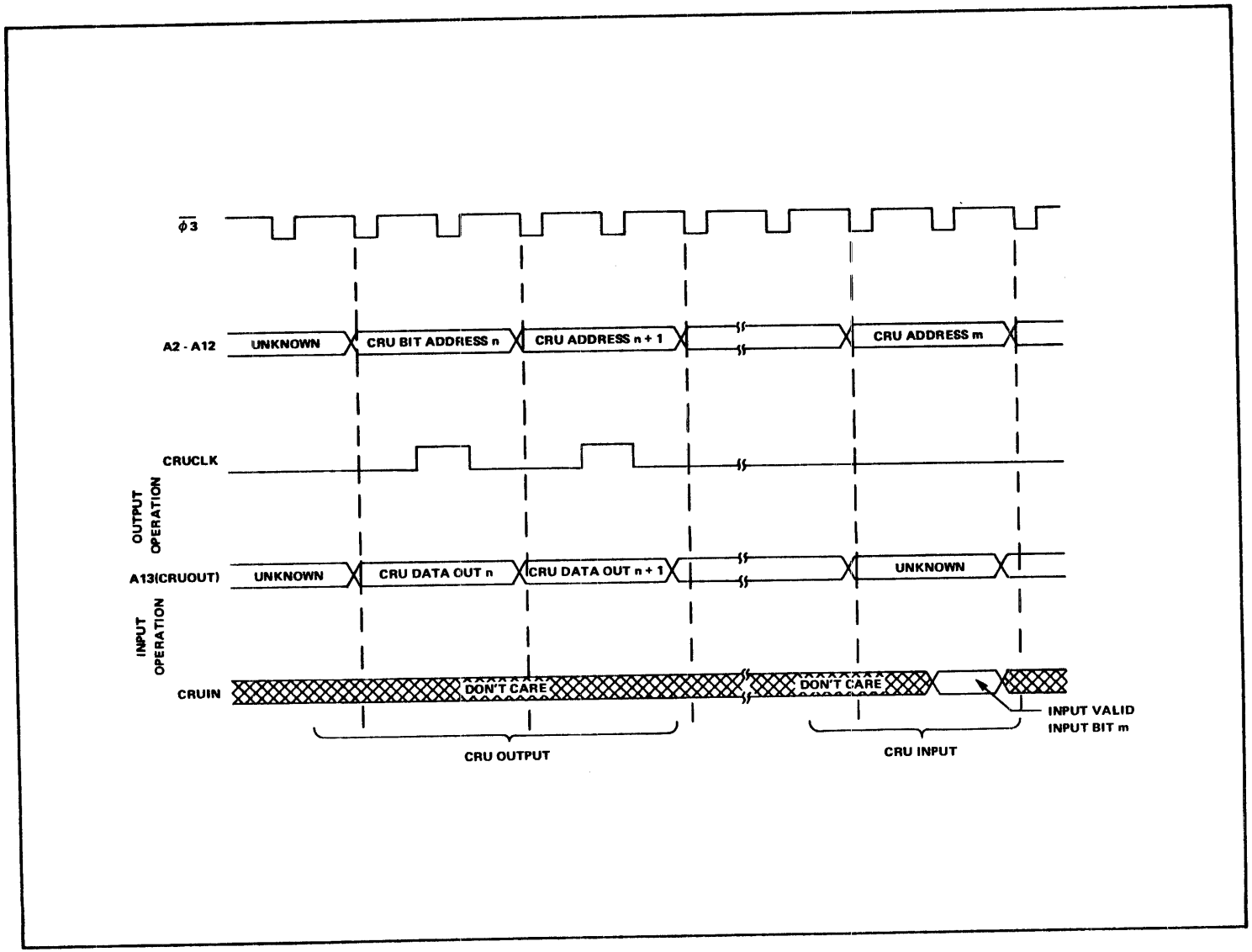


Figure 6-24. TMS 9980A/9981 CRU Interface Timing

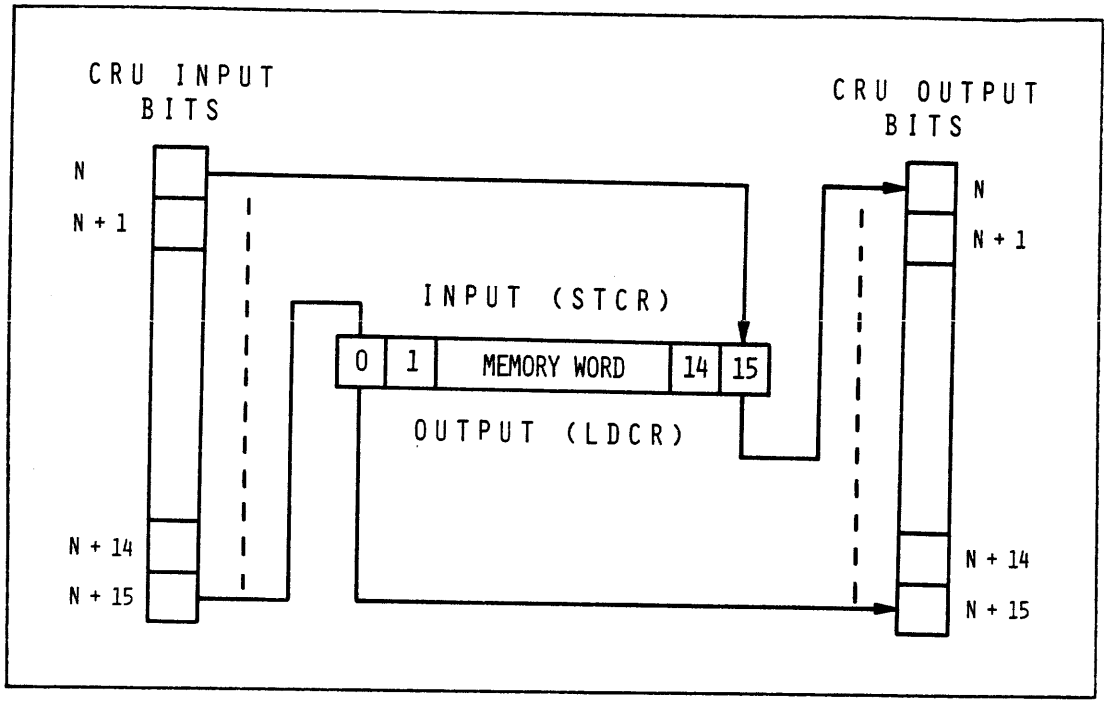


Figure 6-25. TMS 9980A/TMS 9981 LDCR/STCR Data Transfers

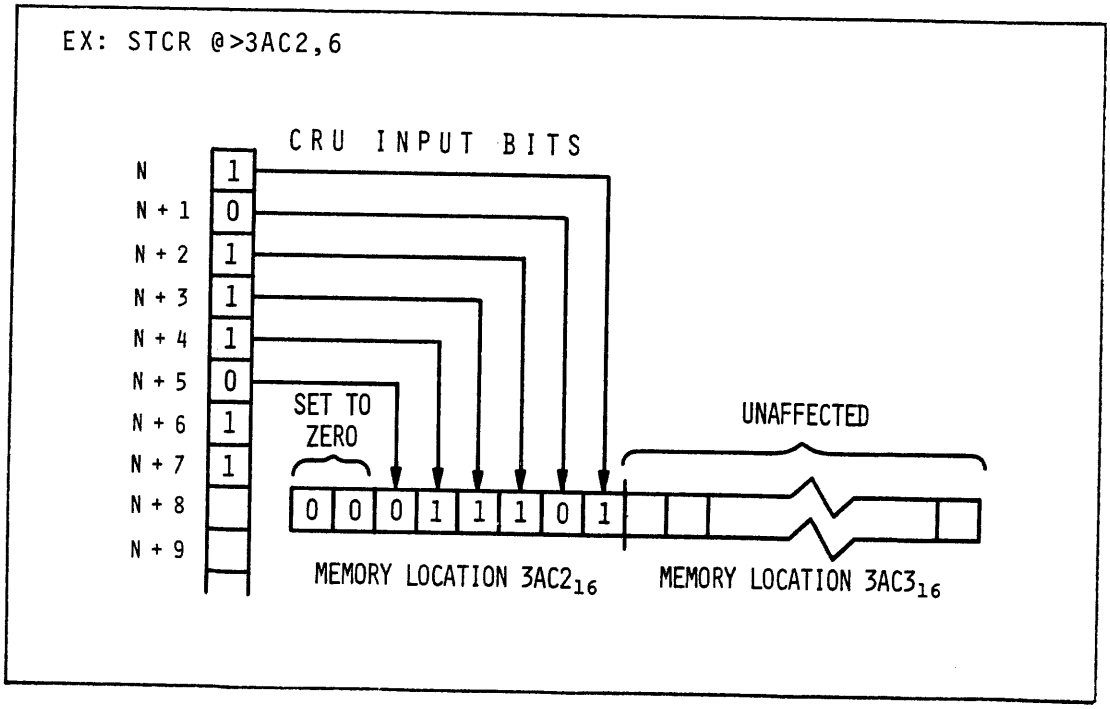


Figure 6-26. A Six-Bit CRU Data Input

Figure 6-27 shows a 12-bit output from workspace register nine.

External Instructions

In additon to the five CRU instructions used for data transfers, there are five other "external" instructions that do not result in data transfers but do cause unique values to be placed on the address bus in conjunction with a CRUCLK strobe. These five instructions have the mnemonics IDLE, RSET, LREX, CKON, and CKOF. The mnemonics were derived from the instructions' defined functions in the 990 minicomputer (from which the 9900 family of microprocessors was derived). With the TMS 9980A microprocessor, these instructions are user-defined; that is, they can be used for specific functions defined by the system designer.

Execution of each of the instructions causes a unique 3-bit value to appear on address lines A0, A1, and A13 in conjunction with the CRUCLK strobe. Figure 6-28 shows the unique value for each instruction. The IDLE instruction is different from the other external instructions in that its 3-bit code and CRUCLK occur repeatedly; that is, the instruction is executed repeatedly, until the idle state is terminated. The IDLE instruction is commonly used to suspend a program until an interrupt occurs.

Transfer Speed

As was mentioned earlier the architecture of a system's I/O can significantly affect the throughput of a processor system; that is, how fast data can be input into the system, be processed, and output.

It was concluded that of the I/O techniques available with the TMS 9980A, DMA is the fastest. It might also be interesting to compare the speed of CRU I/O with the speed of memory-mapped I/O.

An 8-bit transfer from a memory location to the I/O using the data bus and memory-mapped I/O requires about 21 microseconds with a TMS 9980A running at 2.0 MHz. At this same clock speed, an 8-bit CRU data transfer takes about 26 microseconds. In this specific case, CRU I/O takes about 25 percent longer. The faster speed of memory-mapped I/O is due primarily to the ability to pass all eight bits in parallel on the data bus. With the CRU, bits must line up one behind the other, serially, as they are shifted in on the CRUIN line or out on the CRUOUT line.

This slower speed of the CRU is often not a handicap, however, because I/O devices themselves are often much slower than the processor or memory. The relatively slower speed of the CRU when compared to memory-mapped I/O may be less important than the advantage the use of the CRU offers by avoiding the problem of demultiplexing the data bus between memory and I/O.

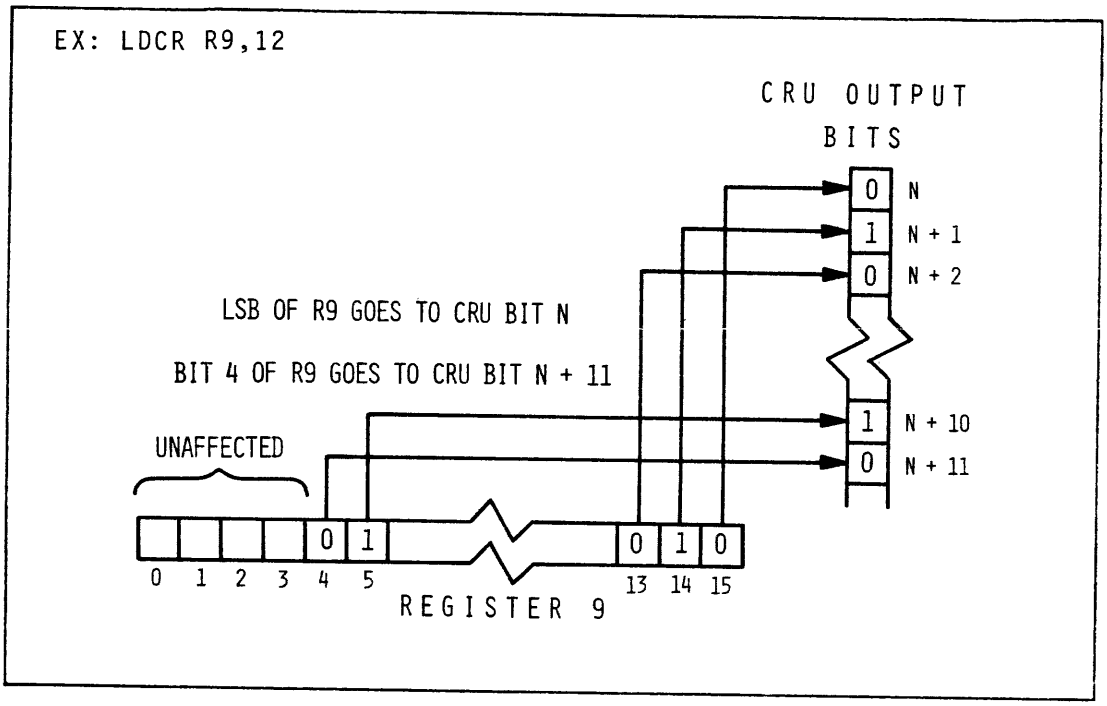


Figure 6-27. A Twelve-Bit CRU Data Output

INSTRUCTION	VALUE PLACED ON ADDRESS BUS		
	A0	A1	A13
IDLE	H	L	L
RSET	H	H	L
LREX	H	H	H
CKON	L	H	H
CKOF	H	L	H
CRU INSTRUCTIONS	L	L	H/L

Figure 6-28. The Three-Bit Codes for External Instructions

In fact, an operation to test or set a bit among a group of I/O bits can be accomplished faster with the CRU than with memory-mapped I/O. The individual bit-addressing ability of the CRU avoids the additional masking operation that memory-mapped I/O requires to isolate a particular bit in a group. For example, suppose a programmer desires to test an I/O bit to see if a switch is off or on. With memory-mapped addressing, the sequence of instructions might look like this:

```
MOV     @SW,R0      GET A GROUP OF I/O BITS
ANDI    R0,>0001    ISOLATE THE SWITCH BIT
JEQ     ON          TEST IF ON OR OFF.
```

With the CRU, the sequence of instructions might look like this (assuming R12 contains a suitable base address):

```
TB      15          INPUT SWITCH STATE
JEQ     ON          TEST IF ON OR OFF.
```

The CRU is a versatile I/O port of the TMS 9980A. Its particular advantages are that it is a separate port for I/O and not intended for memory data transfers, and it facilitates single-bit I/O addressing. There is a group of instructions devoted to the CRU which are discussed in detail shortly.

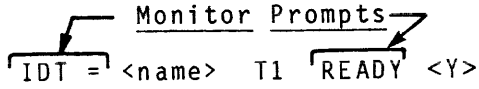
6.6 UNIBUG MONITOR ADDITIONAL COMMANDS (D AND L)

Most of the monitor commands are discussed in previous chapters. At this point, two additional monitor commands are introduced. The "D" and "L" commands allow a program in memory to be dumped onto audio cassette and a program on a cassette to be loaded into memory.

The "DUMP" or D command allows a program to be saved on an audio cassette.

The syntax for the D command is

D<start address> T2 <stop address> T2 <entry address> T2



The contents of memory from the location specified as "start address" to the location specified as the "stop address" will be formatted and recorded on audio tape. The "entry address" entered is the location in memory to begin program execution so that the program counter can be set accordingly. A space or comma is entered after the "entry address," following which the monitor displays IDT=. In reply to this prompting message, the operator can enter up to eight characters that identify the program being saved. After the entry of this identifying name, the operator enters a space. At this point, the monitor displays the message READY and waits

for a Y character to be entered indicating the audio cassette deck has been set up and is ready to record the data. When the Y character is entered, the cassette-deck control bit is activated, and after a short period of time to allow the deck to reach operating speed, the data is recorded on the tape.

As an example, suppose the operator desires to save a program that resides in memory locations 0220_{16} through $031B_{16}$. The first instruction to be executed in the program resides at address 0220_{16}. The program has the name "SINE."

The operator enters D 220 31C 220. The monitor displays IDT= and the operator enters SINE, followed by a space. The monitor then responds with READY Y/N. After ensuring that a cassette is properly loaded, all connections to the cassette deck are correct, and all switch settings are correct, the operator enters a Y.

After waiting a suitable time for the cassette to reach operating speed, the monitor dumps the program from the starting address to the ending address in 990 standard object code format. After completion of the dump, the monitor deactivates the cassette-deck control bit.

Loading a program from a cassette into memory is accomplished as follows. First, place the cassette in the cassette player with the tape read head positioned in front of the program to be entered. Ensure all connections are correct, then enter the "L" (Load) command on the keyboard. The monitor will activate the cassette-deck control bit and begin loading data. The data will be loaded into the addresses specified when the tape was created. After the data is loaded, the monitor deactivates the cassette-deck control bit and displays the name of the program read from the object program.

The syntax for the L command is

```
L
XXXXXX                Monitor displays IDT name.
```

6.7 INSTRUCTION SUBSET 4

The operation of the CRU has been discussed in detail. It was mentioned that there are five instructions in the TMS 9980A's instruction set that are used with the CRU. Three of these instructions (TB, SBO, and SBZ) are used for single-bit CRU addressing. TB allows an input bit to be tested. SBO sets an output bit to a logic ONE, and SBZ sets an output bit to a logic ZERO. The other two instructions (STCR and LDCR) can be used for multibit CRU data transfers. STCR allows up to 16 bits to be input on CRUIN. LDCR allows up to 16 bits to be output serially on CRUOUT.

Each of these five CRU I/O instructions is described in instruction subset four. Additionally, the five external instructions which can be used for user-defined functions are described in this subset.

TEST BIT

TB

CODE: TB Displacement $1F00_{16}$

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

Length: 1 word

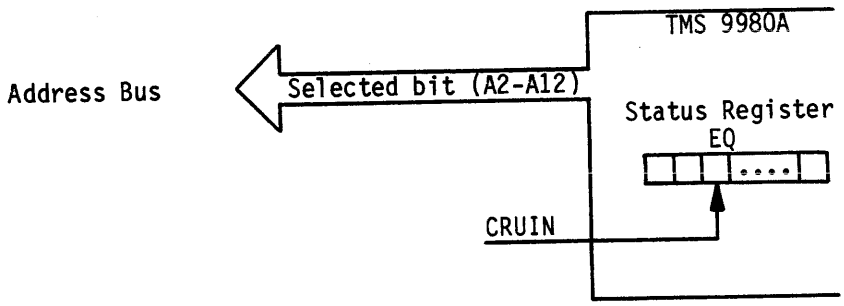
RESULT: CRU Bit Read → EQ Bit of Status Register

STATUS REGISTER:

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

OPERATION:

Read a selected CRU input bit. The address of the selected bit is determined by adding the CRU hardware base address (the contents of R12 bits 4 through 14) and the 8-bit signed displacement in the low order byte of the instruction. (The 8-bit displacement is sign-extended to 11 bits before being added to the CRU hardware base address in R12.) The address of the selected bit appears on address lines A2 through A12. Address lines A0 and A1 are ZERO. The selected bit is input through CRUIN and the equal status bit is set to the value read.



NOTES:

Used to read the state of a CRU input bit and to make subsequent decisions based upon the value read.

Example:

Determine if the switch at CRU hardware address >1A0 is on (ONE) or off (ZERO).

```
LI   R12,>350    >350 IS >1A8 DISPLACED LEFT ONE PLACE
TB   -8          (>1A8) + (-8) = CRU HARDWARE ADDRESS >1A0
JEQ  ON          JUMP TO "ON" IF SWITCH IS A ONE
```

SET BIT TO LOGIC ONE — SBO

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: SBO Displacement $1D00_{16}$	0	0	0	1	1	1	0	1	Displacement							

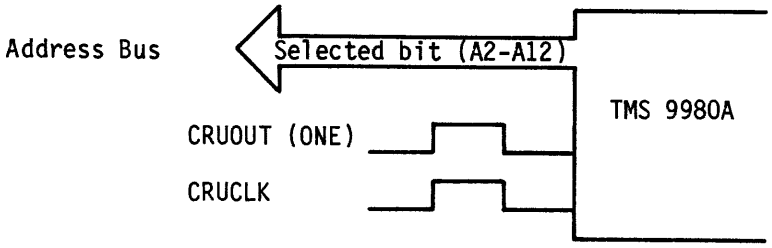
Length: 1 word

RESULT: Logic ONE ⟶ Selected CRU bit

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

Set a selected CRU output bit to a logic ONE. The address of the selected bit is determined by adding the CRU hardware base address (the contents of R12 bits 4 through 14) and the 8-bit displacement in the low order byte of the instruction. (The 8-bit displacement is sign-extended to 11 bits before being added to the CRU hardware base address in R12.) The resulting address appears on address lines A2 through A12 which is used to select the individual bit. Address lines A0 and A1 are ZERO. The logic ONE bit goes out on CRUOUT in conjunction with a CRUCLK strobe.



NOTES:

Used to set a CRU output bit to a logic ONE.

Example:

Turn on a motor with a switch at CRU hardware address >F.

```
LI   R12,0      BASE ADDRESS IN R12 IS ZERO
SBO  15         TURN ON (SET TO ONE) THE MOTOR
```

SET BIT TO LOGIC ZERO

SBZ

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: SBZ Displacement $1E00_{16}$	0	0	0	1	1	1	1	0	Displacement							

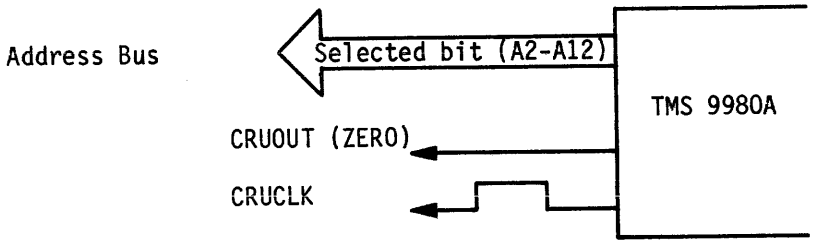
Length: 1 word

RESULT: Logic ZERO ⟶ Selected CRU Bit

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

Force a selected CRU output bit to a logic ZERO. The address of the selected bit is determined by adding the CRU hardware base address (the contents of R12 bits 4 through 14) and the 8-bit displacement in the low order byte of the instruction. (The 8-bit displacement is sign-extended to 11 bits before being added to the CRU hardware base address in R12.) The resulting address appears on address lines A2 through A12 which is used to select the individual bit. Address lines A0 and A1 are ZERO. The logic ZERO bit goes out on CRUOUT in conjunction with a CRUCLK strobe.



NOTES:

Used to force a CRU output bit to a logic ZERO.

Example:

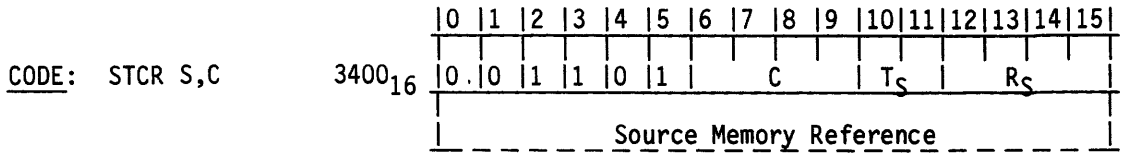
Turn off an indicator light at CRU hardware address >20.

```
LI   R12,32      PUT >20 IN R12
SLA  R12,1       DISPLACE CRU ADDRESS ONE PLACE LEFT
SBZ  0           TURN OFF (FORCE TO ZERO) THE LIGHT
```

or, optionally, to eliminate the SLA instruction,

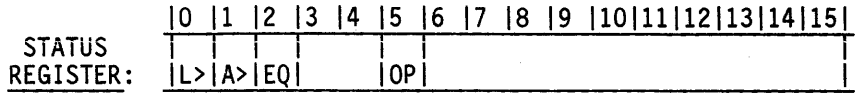
```
LI   R12,64      PUT CRU BASE ADDRESS (>40) in R12
SBZ  0           TURN OFF (FORCE TO ZERO) THE LIGHT
```

STORE COMMUNICATIONS REGISTER UNIT STCR



Length: 1 or 2 words

RESULT: CRU lines ⟶ (S) for C bits



OPERATION:

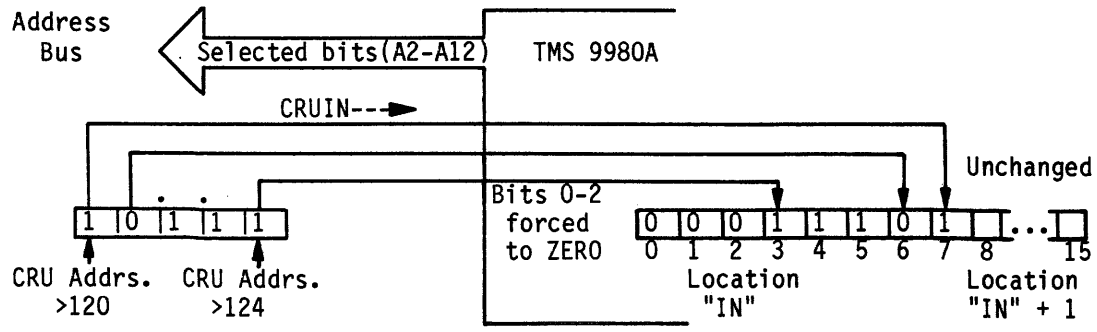
Input the number of bits specified by C from consecutive CRU hardware addresses to the source operand. If C is zero, 16 bits are transferred. Each bit is transferred in sequentially on CRUIN. The CRU hardware base address in R12 (bits 4 through 14) indicates the beginning CRU hardware address which appears on address lines A2 through A12 to select the first input bit and the address changes sequentially on the address lines at timed intervals to select subsequent bits. If the number of bits specified is greater than eight, the source operand is a word address; otherwise, the source operand is a byte address. The first bit transferred in goes to the least-significant bit of the source operand and subsequent bits transferred in fill the source operand toward the most-significant bit position. Any unfilled bits in the source operand are forced to ZERO. After the transfer, the entire source operand (word or byte), not just the bits transferred, is compared to zero and status bits L>, A>, and EQ are set accordingly. With byte addressing, the OP status bit is also affected. The contents of R12 remain unchanged.

NOTES:

Example:

Input five bits from CRU hardware address >120
to memory location IN

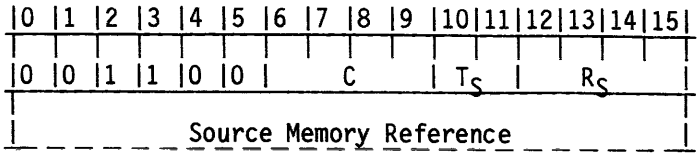
```
LI    R12,>240   >240 IS >120 MOVED ONE PLACE LEFT
STCR  @IN,5
```



Instruction Summary 6-4

LOAD COMMUNICATION REGISTER UNIT — LDCR

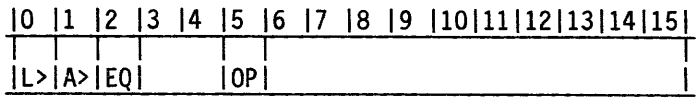
CODE: LDCR S,C — 3000_{16}



Length: 1 or 2 words

RESULT: (S) ⟶ CRU for C bits

STATUS REGISTER:



OPERATION:

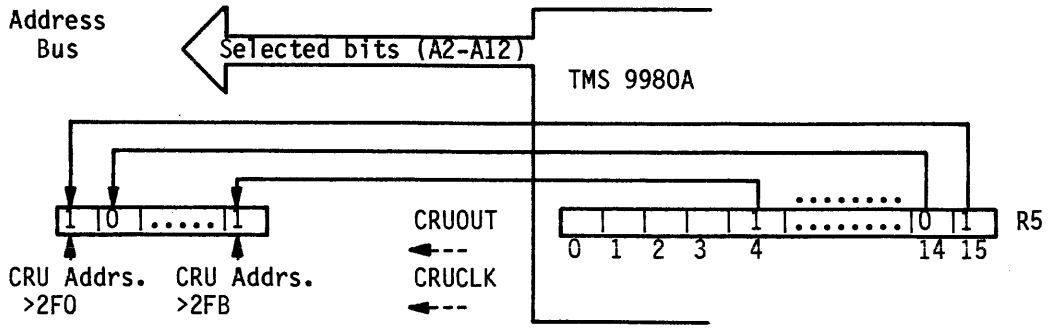
Output the number of bits specified by C from the source operand to consecutive CRU hardware addresses. If C is zero, 16 bits are transferred. Each bit is transferred out sequentially on CRUOUT in conjunction with a CRUCLK strobe. The CRU hardware base address in R12 (bits 4 through 14) indicates the beginning CRU hardware address which appears on address lines A2 through A12 to select the destination of the first output bit and the address changes sequentially on the address lines at timed intervals to select subsequent bits. If the number of bits specified is greater than eight, the source operand is a word address; otherwise, the source operand is a byte address. The first bit transferred out is the rightmost bit in the word or byte source operand. The entire source operand (word or byte), not just the transferred bits, is compared to zero and status bits L>, A>, and EQ are set accordingly. With byte addressing, the OP status bit is set if the bits in the source operand establish odd parity. The contents of R12 and the contents of the source operand remain unchanged.

NOTES:

Example:

Output a 12-bit value in R5 through the CRU, beginning at CRU hardware address >2F0.

```
LI    R12,>5E0   >5E0 IS >2F0 MOVED ONE PLACE LEFT
LDCR  R5,12      OUTPUT THE BIT TRAIN
```



Instruction Summary 6-5

CODE: IDLE 0340_{16}

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

Length: 1 word

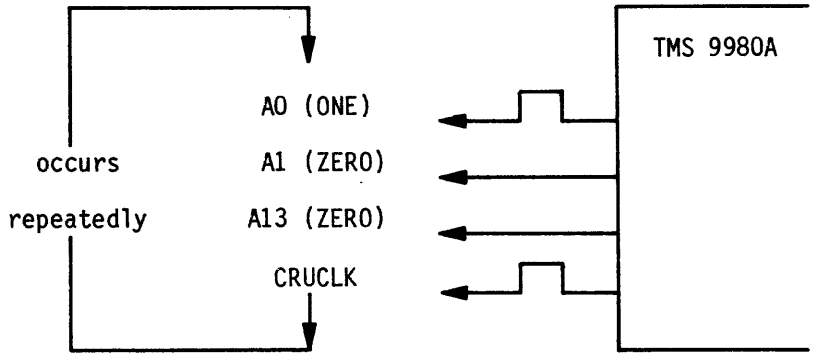
RESULT: Idle the computer

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

Idle the computer until an interrupt occurs. The program remains at the IDLE instruction, executing it repeatedly. The values 1, 0, and 0 appear on address lines A0, A1 and A13, respectively, in conjunction with a CRUCLK strobe with each execution of the instruction. When an interrupt occurs, causing a context switch, the saved return address is the instruction following the IDLE instruction.



NOTES:

Used to suspend the processor while awaiting an interrupt. The unique code on A0, A1 and A13 with CRUCLK may be used to implement specific hardware functions.

RESET

RSET

CODE: RSET 0360_{16}

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

Length: 1 word

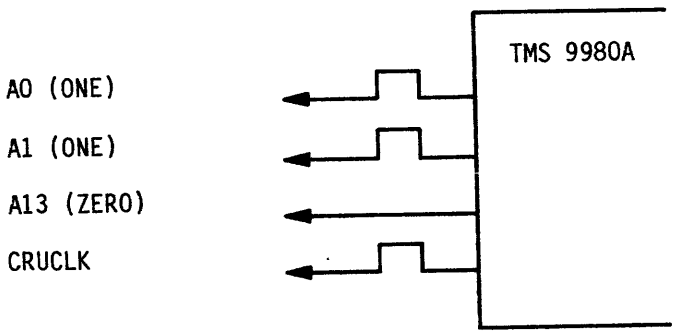
RESULT: User-defined function

STATUS REGISTER:

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

OPERATION:

The values 1, 1 and 0 appear on address lines A0, A1 and A13, respectively, in conjunction with a CRUCLK strobe. The interrupt mask (status bits 12 through 15) is cleared (forced to zero).



NOTES:

The unique code on A0, A1 and A13 with CRUCLK may be used to implement specific hardware functions.

LOAD OR RESTART EXECUTION LREX

CODE: LREX $03E0_{16}$

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

Length: 1 word

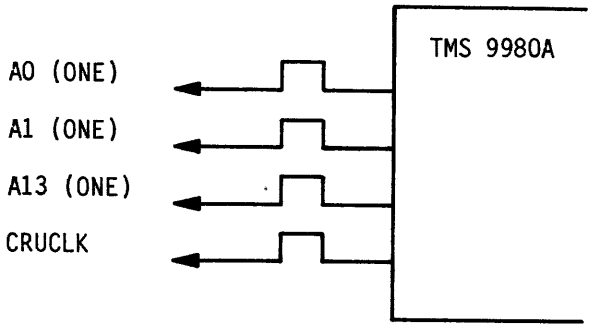
RESULT: User-defined function

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

The values 1, 1, and 1 appear on address lines A0, A1, and A13 in conjunction with a CRUCLK strobe.



NOTES:

The unique code on A0, A1, and A13 with CRUCLK may be used to implement specific hardware functions. On the TM 990/189, this instruction causes an initialization of the system.

CLOCK ON

CKON

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: CKON	$03A0_{16}$	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0

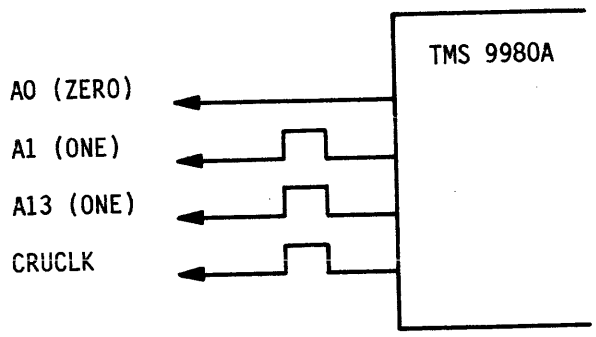
Length: 1 word

RESULT: User-defined function

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

The values 0, 1, and 1 appear on address lines A0, A1, and A13, respectively, in conjunction with a CRUCLK strobe.



NOTES:

The unique code on A0, A1, and A13 with CRUCLK may be used to implement specific hardware functions.

CLOCK OFF

CKOF

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: CKOF	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0

$03C0_{16}$

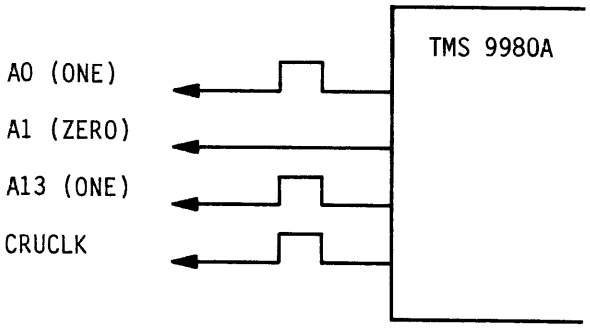
Length: 1 word

RESULT: User-defined function

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

The values 1, 0, and 1 appear on address lines A0, A1, and A13, respectively, in conjunction with a CRUCLK strobe.



NOTES:

The unique code on A0, A1, and A13 with CRUCLK may be used to implement specific hardware functions.

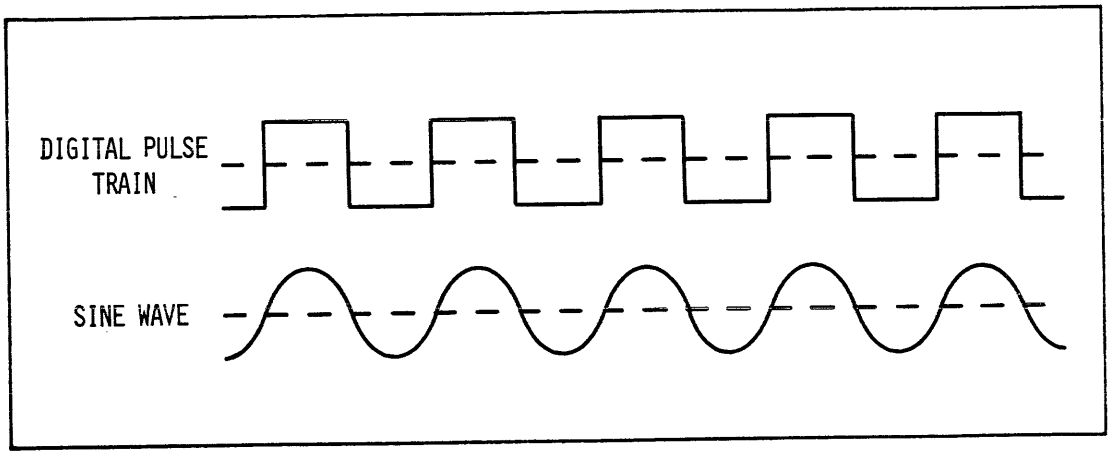


Figure 6-29. A Digital Pulse Train Resembles a Sine Wave

6.8 PROGRAM EXAMPLE: CYCLE GENERATOR

Goal of the Program Example

The following program illustrates several important concepts. It demonstrates the use of the CRU, especially its use on the University Board, and it demonstrates the concept of program-controlled I/O as well as program-controlled timing loops.

What the Program Does

The program produces a tone on the University Board's piezoelectric speaker. The speaker is driven from the TMS 9901 on the University Board. (The TMS 9901 is a special LSI peripheral component for the 9900 family and is discussed in detail in the following chapter.) For right now, it is enough to know how to address the speaker. It has a CRU hardware address of $21E_{16}$. The program is designed to have enough flexibility to be adapted to a number of applications. Specifically, the program will be used to drive the speaker to produce a tone, but the same principles could be applied to allow duty-cycle control of a device such as a motor or valve.

The program employs single-bit CRU instructions to produce a digital pulse train which approximates an equivalent sine wave illustrated in Figure 6-29.

Program Design

The program accepts three inputs: the on time expressed as the number of 50-microsecond time intervals, the off time expressed as 50-microsecond time units and, thirdly, the number of cycles.

Different values for the on time and off time are permitted to allow duty cycling. To simulate a sine wave, the on time and off time will be the same. The number input represents the time required for half a cycle. Figure 6-30 is the flowchart for the program. Figure 6-31 is a listing of the program. Refer to Figure 6-31 for the following discussion.

The program begins by setting the workspace pointer and then sets the CRU software base address in R12 (the LI instruction at location 224_{16}). The speaker has a CRU hardware base address of $21E_{16}$, but this hardware address must be offset one bit position to the left when it appears in R12, so R12 is loaded with the value $43C_{16}$, which is $21E_{16}$ shifted left one position (or multiplied by 2).

Following this "housekeeping," the value in R0 (the number of time units for the on time) is copied into R3 so that it can be decremented but not lost. The speaker is turned on with the SBO instruction at location $22A_{16}$. A displacement of zero is used, because the specific bit address is established in R12. The instructions included between locations $22A_{16}$ to 236_{16} constitute the "on-time loop" during which the speaker is on. This loop is carefully structured to force the execution time of the loop to be 50 microseconds, or as close as possible. A timing analysis of the loop will be made shortly, but for now the remaining logic will be examined. At location $22C_{16}$ the off time is copied into R4 so that it can be decremented later. At first glance, it may appear unusual to see the off-time count in the on-time loop, but it is done here as a timing consideration. Notice that the MOV instruction at location $22C_{16}$ and the SBO instruction at location $22A_{16}$ are executed each time through the loop, though it is not necessary to satisfy the strict logic of the program. Again, the placement of the instructions as well as their redundant execution is a part of the timing considerations.

At location $22E_{16}$ the on-time count is decremented. If the result is nonzero, the SRC instruction at location 232_{16} is executed. The SRC instruction provides a very specific timing delay which causes the balance of the time loop to be 50 microseconds. The SRC instruction specifically was picked to do this "fine tuning" of the timing because it is an instruction the execution time of which can be spread over a wide range depending on the count supplied in the shift operation. Any of the other shift instructions could have been used. Following the delay, the absolute jump at location 234_{16} ties the loop back to location A1.

When the on-time count in R3 decrements to zero, the JEQ instruction at location 230_{16} transfers control to location D0 where the SRC instruction at location $23E_{16}$ fills out the last on-time period and allows the unconditional jump instruction at location 240_{16} enter the off-time loop at location 236_{16} (labeled D1).

The instructions from location 236_{16} to location 242_{16} constitute the off-time loop which is structured similarly to the on-time

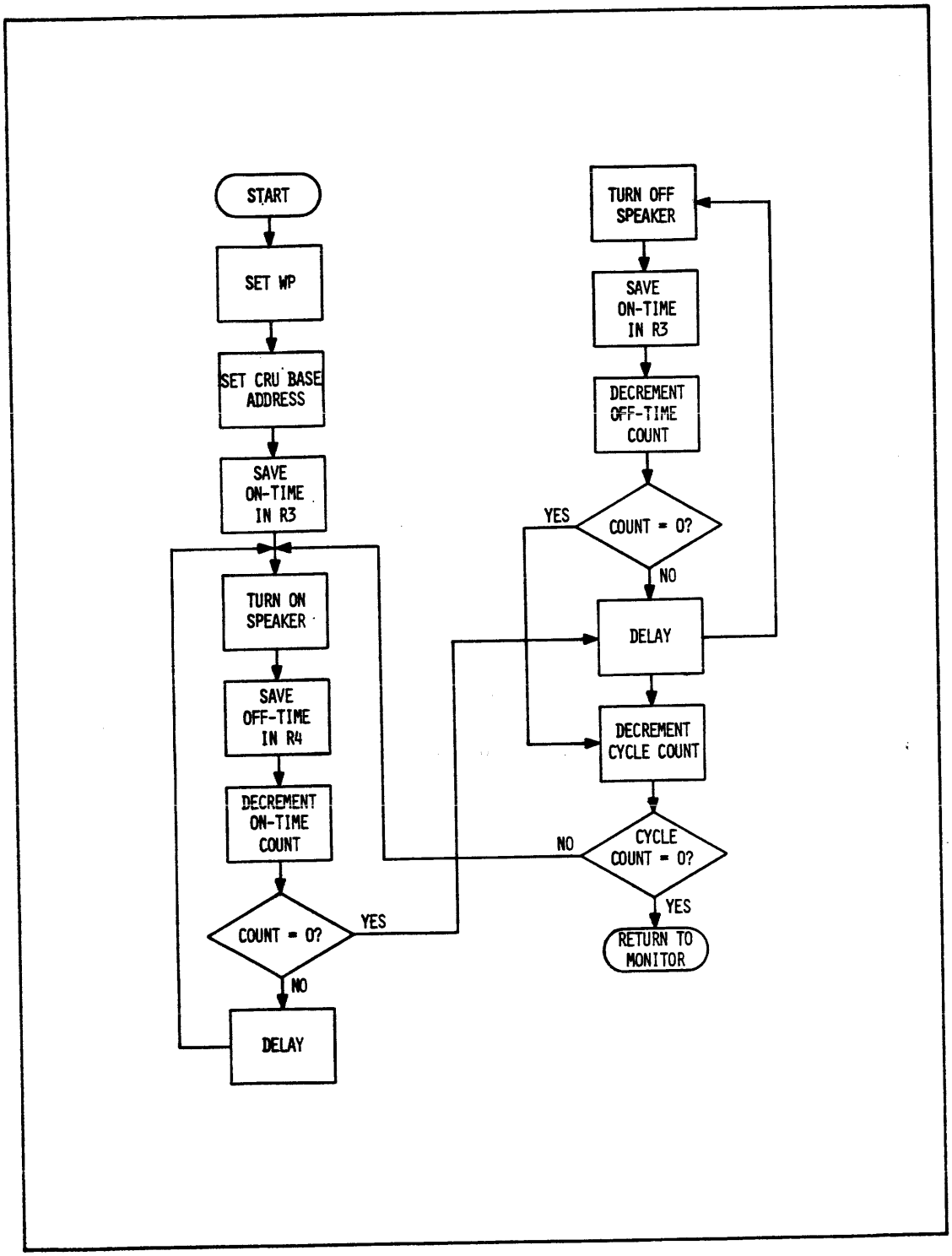


Figure 6-30. Wave Generator Program Flowchart

```
                SDSMAC      3.1 *    10:40:05 FRIDAY, DEC 22, 1978
                                                                    PAGE 0002
 0001                    *  INPUTS:
 0002                    *    R0 - ON TIME IN UNITS OF 50 MICROSEC.
 0003                    *    R1 - OFF TIME IN UNITS OF 50 MICROSEC.
 0004                    *    R2 - NUMBER OF CYCLES
 0005                    *  OTHER REGS. USED:
 0006                    *    R3 - TEMP. STORAGE
 0007                    *    R4 - TEMP. STOARAGE
 0008                    *    R5 - DELAY GENERATOR
 0009                    *    R12 - CRU BASE ADRS.
 0010                    *
 0011 0200               ADRG >200
 0012 0200          WS   BSS  32          WORKSPACE AREA
 0013 0220 02E0     ST   LWPI WS          SET WP
      0222 0200
 0014 0224 020C          LI   R12,>43C SET CRU BASE ADRS
      0226 043C
 0015 0228 C0C0     WC   MOV  R0,R3      SAVE "ON" TIME
 0016 022A 1D00     A1   SBO  0           TURN ON SPEAKER
 0017 022C C101          MOV  R1,R4       SAVE "OFF" TIME
 0018 022E 0603          DEC  R3          DECREMENT "ON" TIME
 0019 0230 1306          JEQ  D0          "ON" TIME EXPIRED?
 0020 0232 0B35          SRC  R5,3        NO-DELAY
 0021 0234 10FA          JMP  A1
 0022 0236 1E00     D1   SBZ  0           TURN OFF SPEAKER
 0023 0238 C0C0          MOV  R0,R3       SAVE "ON" TIME
 0024 023A 0604          DEC  R4          DECREMENT "OFF" TIME
 0025 023C 1302          JEQ  D2          "OFF" TIME EXPIRED?
 0026 023E 0B35     D0   SRC  R5,3        DELAY
 0027 0240 10FA          JMP  D1
 0028 0242 0602     D2   DEC  R2          DECREMENT CYCLE COUNT
 0029 0244 16F2          JNE  A1          ALL CYCLES COMPLETE?
 0030 0246 0460          B    @>3000      YES-RETURN TO MONITOR
      0248 3000
 0031      0220          END  ST
NO ERRORS
```

Figure 6-31. Wave Generator Program Listing

loop. At location $23C_{16}$ the JEQ instruction transfers control out of the loop when the off time expires to location 242_{16} (labeled D2). There, the DEC instruction decrements the cycle count in R2. If the cycle count is nonzero, control is transferred to A1 and another cycle is begun. When the cycle count does go to zero, the B instruction at location 246_{16} turns control to the monitor.

The on-time loop and the off-time loop are designed to each require 50 microseconds of execution time.

Some insight into how this exact delay was accomplished can be gained by examining the on-time loop. Each of the six instructions in this loop requires a finite amount of time to execute. The amount of time for each instruction to execute is determined by the number of clock cycles required by each instruction, the addressing mode(s) used, the number of memory accesses with wait states (if any), and the system clock speed.

The formula for calculating the execution time is given in the TMS 9980A/TMS 9981 Microprocessor Data Manual. The formula is reproduced here for the sake of convenience in Figure 6-32.

The timing analysis will start by examining the first instruction in the on-time loop, the SBO instruction at location $22A_{16}$. Table 4 from the TMS 9980A/TMS 9981 Microprocessor Data Manual (reproduced in Figure 6-33) shows that the SBO instruction requires 16 clock cycles (if there are no wait states). (The design of the University Board requires no wait states for the processor to access memory.) The University Board system clock runs at 2.0 MHz, which yields a clock-cycle time of 0.5 microsecond. Applying the formula to the SBO instruction, the amount of time required to execute the instruction is

$$T = 0.5[16 + (0 \times 8)] = 8.0 \text{ microseconds.}$$

A similar analysis reveals that the MOV instruction at location $22C_{16}$ requires 11.0 microseconds.

Similarly, it can be shown that the DEC instruction at location $22E_{16}$ requires 8.0 microseconds. The JEQ instruction at location 230_{16} requires 5.0 microseconds if the transfer of control is not made, or 6.0 microseconds if the transfer of control is made. In most instances through the loop, the jump will not be made. The transfer of control is made only when the count in R3 goes to zero.

The SRC instruction at location 232_{16} requires anywhere from 10 microseconds to 30 microseconds depending upon the shift count. A shift count of 3 produces an execution time of 12 microseconds.

TMS 9980A/TMS 9981 INSTRUCTION EXECUTION TIMES

Instruction execution times for the TMS 9980A/TMS 9981 are a function of:

1) Clock cycle time, $t_{c(\phi)}$

2) Addressing mode used where operands have multiple addressing mode capability

3) Number of wait states required per memory access.

Table 4 lists the number of clock cycles and memory accesses required to execute each TMS 9980A/TMS 9981 instruction. For instructions with multiple addressing modes for either or both operands, Table 4 lists the number of clock cycles and memory accesses with all operands addressed in the workspace-register mode. To determine the additional number of clock cycles and memory accesses required for modified addressing, add the appropriate values from the referenced tables. The total instruction-execution time for an instruction is:

$$T = t_{c(\phi)} (C+W \cdot M)$$

where:

T = total instruction time;

$t_{c(\phi)}$ = clock cycle time;

C = number of clock cycles for instruction execution plus address modification;

W = number of required wait states per memory access for instruction execution plus address modification;

M = number of memory accesses.

As an example, the instruction MOVB is used in a system with $t_{c(\phi)}$= 0.400μs and no wait states are required to access memory. Both operands are addressed in the workspace register mode:

$$T = t_{c(\phi)} (C+W \cdot M) = 0.400\ (22+0 \cdot 8) = 8.8\ \mu s.$$

If two wait states per memory access were required, the execution time is:

$$T = 0.400\ (22 + 2 \cdot 8)\ \mu s = 15.2\ \mu s.$$

If the source operand was addressed in the symbolic mode and two wait states were required:

$$T = t_{c(\phi)} (C+W \cdot M)$$

$$C = 22 + 10 = 32$$

$$M = 8 + 2 = 10$$

$$T = 0.400\ (32 + 2 \cdot 10) = 20.8\ \mu s.$$

Figure 6-32. Formula For Determining TMS 9980A Instruction Execution Time

INSTRUCTION EXECUTION TIMES

INSTRUCTION	CLOCK CYCLES C	MEMORY ACCESS M	ADDRESS MODIFICATION*** SOURCE	ADDRESS MODIFICATION*** DESTINATION
A	22	8	A	A
AB	22	8	B	B
ABS (MSB = 0)	16	4	A	–
(MSB = 1)	20	6	A	–
AI	22	8	–	–
ANDI	22	8	–	–
B	12	4	A	–
BL	18	6	A	–
BLWP	38	12	A	–
C	20	6	A	A
CB	20	6	B	B
CI	20	6	–	–
CKOF	14	2	–	–
CKON	14	2	–	–
CLR	16	6	A	–
COC	20	6	A	–
CZC	20	6	A	–
DEC	16	6	A	–
DECT	16	6	A	–
DIV (ST4 is set)	22	6	A	–
DIV (ST4 is reset)*	104-136	12	A	–
IDLE	14	2	–	–
INC	16	6	A	–
INCT	16	6	A	–
INV	16	6	A	–
Jump (PC is changed)	12	2	–	–
(PC is not changed)	10	2	–	–
LDCR (C = 0)	58	6	A	–
(1<C<8)	26+2C	6	B	–
(8<C<15)	26+2C	6	A	–
LI	18	6	–	–
LIMI	22	6	–	–
LREX	14	2	–	–
LWPI	14	4	–	–
MOV	22	8	A	A
MOVB	22	8	B	B
MPY	62	10	A	–
NEG	18	6	A	–
ORI	22	8	–	–
RSET	14	2	–	–
RTWP	22	8	–	–
S	22	8	A	A
SB	22	8	B	B
SBO	16	4	–	–
SBZ	16	4	–	–
SETO	16	6	A	–
Shift (C ≠ 0)	18+2C	6	–	–
(C ≠0, Bits 12–15 of WRO = 0)	60	8	–	–
(C = 0, Bits 12–15 of WRP = N ≠ 0)	28+2N	8	–	–
SOC	22	8	A	A
SOCB	22	8	B	B
STCR (C = 0)	68	8	A	–
(1≤C≤7)	50	8	B	–
(C = 8)	52	8	B	–
(9≤C≤15)	86	8	A	–
STST	12	4	–	–
STWP	12	4	–	–
SWPB	16	6	A	–
SZC	22	8	A	A
SZCB	22	8	B	B
TB	16	4	–	–
X**	12	4	A	–
XOP	52	16	A	–
XOR	22	8	A	–
RESET function	36	10	–	–
LOAD function	32	10	–	–
Interrupt context switch	32	10	–	–
Undefined op codes: 0000-01FF, 0320 033F, 0C00-0FFF, 0780-07FF	8	2	–	–

**Execution time is added to the execution time of the instruction located at the source address.

***The letters A and B refer to the respective tables that follow.

Figure 6-33. Instruction Clock Cycle Requirements

The unconditional jump instruction at ocation 234_{16} requires 6.0 microseconds. Adding the execution times:

SBO	8.0 microseconds
MOV	11.0 microseconds
DEC	8.0 microseconds
JEQ	5.0 microseconds (normally)
SRC	12.0 microseconds
JMP	6.0 microseconds

produces a total of 50 microseconds through the loop. A similar analysis of the off-time loop (locations 236_{16} through 241_{16}) would show the same results.

Looking again at the on-time loop, the last execution of the loop requires 51 microseconds because of the additional time required by the JEQ instruction at location 230_{16}. Follow the sequence of instructions the last time through the loop to verify this.

The off-time loop normally produces an identical 50 microsecond delay; however, the last time through the off-time loop produces a delay of only 47 microseconds.

The slight timing skew in the last time through the on-loop and off-loop is of little consequence in the normal use of the program.

As an initial experiment, run the program with inputs in R0 to produce a 1-kHz tone of one-second duration.

To produce a 1-kHz tone, a cycle must be completed every millisecond, which means the on time is 500 microseconds and the off time is also 500 microseconds. A cycle of this duration can be specified by putting a cycle count of 10 (A_{16}) in R0 and 10 (A_{16}) in R1.

To produce a tone of one-second duration, 1000 cycles must be produced. This can be specified to the program by placing $3E8_{16}$ (1000_{10}) in R2.

With these inputs in R0, R1, and R2, the program will produce a 1-second, 1-kHz tone.

Program Operation

Use the UNIBUG monitor commands to load the program into memory at the addresses indicated by the listing. Set the workspace pointer to 200_{16} and then initialize R0, R1, and R2 to their proper value. Use the P command to set the program counter to the address of

the first executable instruction (220_{16}). Finally, enter the E command to cause the program to execute.

The program should produce a 1-second, 1-kHz tone and then return to the UNIBUG monitor. To run the program again, initialize R2 again (or change R0, R1, and R2 if a different tone or duration is desired), initialize the program counter again, and enter the E command.

6.9 SUMMARY

In this chapter the third part of a microprocessor system, the input/output section, is introduced.

The advantage of the three general types of I/O is discussed. Program-controlled I/O in which the program initiates interaction and specifically controls the input/output operation is the most simple. Interrupt-driven I/O allows an I/O device to initiate interaction with the processor. By taking control of the system buses directly, DMA allows the fastest method of I/O data transfer. All three of these general types of I/O are available with the TMS 9980A, although DMA I/O is not implemented on the TM 990/189 board.

The 9900 family of microprocessors allows three methods of input/output data transfers. Memory-mapped I/O treats a peripheral device as a memory location and allows any memory reference instruction to be used for addressing I/O. DMA is also available to permit high-speed data transfers. The communication register unit is a separate, dedicated I/O port which permits direct single-bit addressing.

Specific CRU instructions are introduced and utilized in a program example.

In the next chapter, the discussion of I/O is continued with the introduction of specific I/O peripheral devices available on the University Board.

6.10 EXERCISES

1. Why is the I/O section often the most complex part of a processor system?

2. What advantages are there to using a programmable LSI component instead of discrete SSI or MSI components as the interface between a microprocessor and I/O devices?

3. Of the three general categories of I/O, which one would most likely be used for the following devices in a microprocessor system?

(a) A sensor that is activated when the presence of a deadly gas in a factory is detected

(b) A high-speed disk

(c) A device used to periodically measure the temperature in an office.

4. The TMS 9980A has vectored and prioritized interrupt capability. What advantages does this offer over a microprocessor having interrupt capability, but without vectored or prioritized capability?

5. Assume the following logic state at the CRU hardware addresses indicated.

CRU Hardware Address	Logic State
10_{16}	1
11_{16}	0
12_{16}	1
13_{16}	1
14_{16}	0
15_{16}	0
16_{16}	1
17_{16}	0
18_{16}	1
19_{16}	0
$1A_{16}$	1
$1B_{16}$	1
$1C_{16}$	1
$1D_{16}$	0
$1E_{16}$	0
$1F_{16}$	1
20_{16}	0
21_{16}	1
22_{16}	1
23_{16}	0
24_{16}	1
25_{16}	1
26_{16}	1
27_{16}	1
28_{16}	1

Also assume that each of the following set of instructions begins at location 280_{16}.

What is the content of the PC after each sequence of instructions? Memory location AB is $2AC_{16}$.

(a)
```
LI      R12,>20
TB      0
JEQ     AB
```

```
(b)  LI      R12,>10
     SLA     R12,1
     TB      0
     JEQ     AB

(c)  LI      R12,>50
     TB      0
     JEQ     AB

(d)  LI      R12,>42
     TB      0
     JNE     AB

(e)  LI      R12,>42
     TB      3
     JNE     AB

(f)  LI      R12,>42
     TB      -3
     JEQ     AB

(g)  LI      R12,>20
     TB      >16
     JNE     AB

(h)  LI      R12,>48
     TB      -19
     JEQ     AB
```

6. Assume that register 12 contains 100_{16}. From the following data, write the instructions necessary to set the logic state indicated in the left column at the CRU hardware address indicated in the right column.

	Logic State	CRU Hardware Address
(a)	ONE	80_{16}
(b)	ZERO	88_{16}
(c)	ONE	$9A_{16}$
(d)	ZERO	$7F_{16}$
(e)	ONE	70_{16}
(f)	ZERO	63_{16}
(g)	ONE	CB_{16}
(h)	ZERO	47_{16}

7. Assuming the same CRU hardware addresses and logic states shown in Exercise 5, what is the content of register 6 after each of the following sequence of instructions? Assume R6 contains the value $A1F7_{16}$ prior to each pair of instructions.

```
(a)  LI      R12,>20
     STCR    R6,9

(b)  LI      R12,>10
     SLA     R12,1
     STCR    R6,9

(c)  LI      R12,>20
     STCR    R6,0

(d)  LI      R12,>28
     STCR    R6,13

(e)  LI      R12,>28
     STCR    R6,5

(f)  LI      R12,>36
     STCR    R6,10
```

8. A TMS 9980A-based system employs the CRU to address an I/O device with a CRU hardware base address of 70_{16}. The CRU addresses are structured as follows.

> >70 is equal to ZERO when the device has no character to send.
> >70 is equal to ONE when the device has a character to send.
> >71 is the LSB of an 8-bit character.
> >78 is the MSB of the 8-bit character.

Write a program to check to see if a character is ready. Keep checking until one is ready, then read the 8-bit character into memory byte location KA. Check the parity of the character input. If the parity is odd, transfer control to location ER.

9. Write a program segment to transfer the 40-bit value beginning at memory location FD to an output device with a CRU hardware base address of 140_{16}.

10. What is the advantage of serial data transfer as compared to parallel data transfer? Give an example of where serial data transfer would be advantageous.

11. What is the advantage of parallel data transfer as compared to serial data transfer? Give an example of where parallel data transfer would be advantageous.

12. With a TMS 9980A running at 2.0 MHz, what is the output data rate via the CRU for an 8-bit data transfer? If this data rate is too fast for a device, what alternatives might be taken to accommodate the device?

6.11 LAB EXPERIMENTS

1. Use the D command to save the cycle generator program on a cassette tape. Destroy the program in memory (perhaps by removing power) and then use the L command to load the program back into memory.

2. Calculate, enter the values, and run the cycle generator program to produce the number of cycles, the cycle length, and the percentage of on-time shown below.

Number of Cycles	Cycle Length	Percentage of On-Time
1000	1 millisecond	20%
65,536	0.5 millisecond	50%
15	1 second	70%

How long will each program run?

3. The CRU software base address for the rightmost LED in the group of four LED's on the TM 990/189 is 20_{16}. (Hexadecimal 20 is the value to be placed in R12.) Modify the cycle generator program to vary the length of time that the LED is illuminated and its intensity using the values in Lab Experiment 2.

4. Modify the cycle generator program to pulse the LED repeatedly on and off using the values below.

On	Off	Total Time
0.5 sec.	0.5 sec.	20 sec.
0.5 sec.	1.0 sec.	30 sec.
1.0 sec.	0.5 sec.	30 sec.
1.0 sec.	1.0 sec.	60 sec.

5. Given the frequency of the following musical tones, modify the cycle generator program to produce a 1-second duration tone for each.

CHAPTER 7

INPUT/OUTPUT DESIGN

7.1 INTRODUCTION

In the preceding chapter the fundamental concepts of I/O are introduced, with particular emphasis on Texas Instruments' CRU concept. In this chapter examples of using the various techniques introduced in the last chapter are presented. Two LSI devices included on the University Board are discussed in detail. These provide the interface between the TMS 9980A's CRU port and peripheral devices.

7.2 I/O INTERFACING CONSIDERATIONS

Memory-Mapped I/O

The first type of I/O to be discussed is memory-mapped I/O. As with most microprocessor systems, this method of I/O is possible with the 9900 family, particularly the TMS 9980A-based University Board. With some microprocessors, I/O devices use the same signals as memory devices. In these cases I/O uses up a portion of addressable memory. For example, a microprocessor capable of addressing 65,536 bytes of memory might have only 63,488 bytes of memory installed, with the remaining 2048 bytes assigned to I/O devices. Consider the memory allocation and address decoding for such a case.

Suppose that, due to other system constraints, the highest 14K of address space is to be allocated to nonvolatile memory and the lowest 48K to read/write memory. Figure 7-1 shows the memory map for this allocation, and Figure 7-2 illustrates one of several possible approaches to the problem of address decoding. Decoding of the higher order address bits is necessary, because individual memory circuits only accept 8 to 14 bits of address, depending on the type of memory devices used. With the address-decoding logic of Figure 7-2a, it is seen that the two most-significant bits set the RAM select line. Only when both bits are logical ONE are ROM or I/O space selected. If RAM is not selected, then the next three bits select between ROM and I/O. The truth table of Figure 7-2b reflects the logic of Figure 7-2a. The three output signals are used to enable the major memory circuits, which must further decode the necessary address lines to select bytes for communication with the CPU.

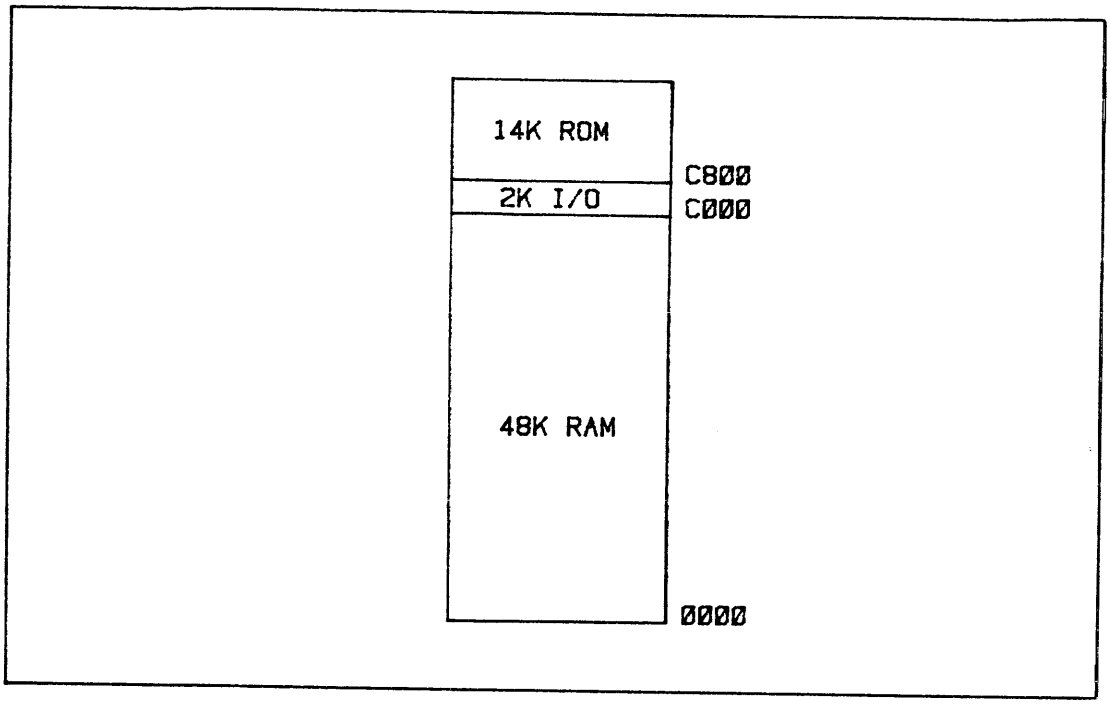


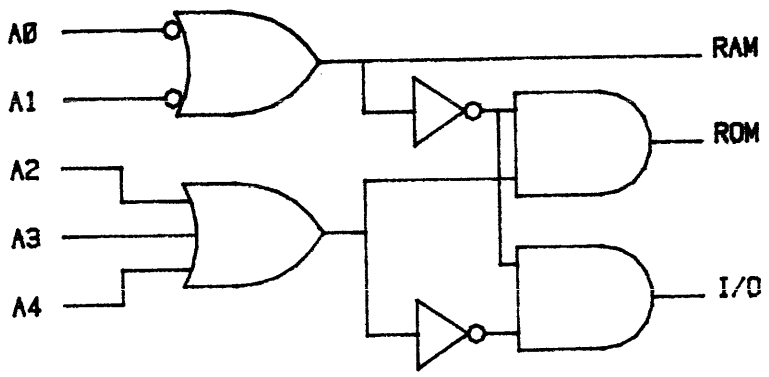
Figure 7-1. Example of Memory Allocation

With some microprocessors, separate read and write strobes are activated on memory instructions and I/O instructions. These simplify the design considerably, because it is then not necessary to partition address space. However, where separate I/O strobes are used, often only the low-order eight bits of address space are applied to I/O, which limits the total amount of I/O to 256 bytes. For many microprocessor applications this is adequate.

The concept of handshaking was mentioned previously in Chapter 1. A common example of handshaking will illustrate more clearly what the term means. The example is the transmission and reception of data using a standardized interface, called RS-232, which is the title number of the standard, as defined by the Electronics Industry Association (EIA). A device commonly employed to effect the interface is called a Universal Asynchronous Receiver/Transmitter (UART).

In simple terms, the transmission of one byte of data in accordance with the RS-232 procedure requires that the following steps be taken.

- Turn the device on.
- Set RTS (request-to-send), signaling the device to prepare to receive data.
- Wait for CTS (clear-to-send), a signal set by the device to indicate readiness.
- Send the data.
- Turn the device off.



A. DECODE LOGIC

A0	A1	A2	A3	A4	RAM	I/O	ROM
0	X	X	X	X	1	0	0
X	0	X	X	X	1	0	0
1	1	0	0	0	0	1	0
1	1	1	X	X	0	0	1
1	1	X	1	X	0	0	1
1	1	X	X	1	0	0	1

B. DECODE TRUTH TABLE

Figure 7-2. Address Decoding Example

Items 2, 3, and 4 may be repeated as often as desired. Items 1 and 5 are generally done only once. Figure 7-3 illustrates a routine that will transmit a block of memory, one byte at a time, using this procedure. This routine assumes a byte of memory-mapped I/O control lines at location C1 and a data byte at location I1. Within the byte at C1, the most-significant bit will be assigned to CTS, so that it can be tested with a JLT instruction. The least-significant bit will be assigned to RTS.

Direct Memory Access (DMA)

DMA is useful because it can relieve the operating program of much of the burden of controlling the communication of data with a device. This is especially important where the device is capable of sending or receiving data at a high rate of speed such as a disk or magnetic-tape transport. There are two primary classes of DMA: "suspended animation" and transparent.

In DMA the I/O device controller is initiated by the processor using, perhaps, memory-mapped I/O. As a minimum, the controller requires the following information:

- Address of beginning of data buffer
- Length of buffer (or address of end of buffer)
- I/O Mode (read, write, rewind, etc.).

Depending on the device, additional information may be required. A disk drive, for example, needs the address of the data on the disk.

Following the initialization of the controller, the action that results depends on whether the controller is designed to operate transparently or by suspending the processor. In transparent operation, the controller uses the buses only when the processor is not using them. This requires careful synchronization. In this mode, once the processor has initialized the controller, the processor can proceed with other work, returning when interrupted by the controller at the completion of the transfer. This is particularly valuable when used in a multi-tasking environment. For example, tasks A and B are competing for processor time. Task A is in control and requests data from a magnetic tape. This request causes the processor to initialize the controller and pass control to task B. Later, when the DMA I/O is complete, task A can regain control and use the data, while, perhaps, task B is waiting for I/O to complete.

In suspended animation, two signals are used to allow the controller to supervise the activity on the buses, somewhat similar to the handshaking signals in the RS 232 example given above. These are HOLD and HOLDA. When the controller desires control of the buses, it lowers the HOLD line to the processor. When the processor reaches a state where it can wait, it stops execution, places all its bus-control lines in a high-impedance state, and raises the

```
      LI      R0,B1          INITIALIZE BUFFER POINTER
      LI      R1,B2          INITIALIZE BUFFER LENGTH
      LI      R2,I1          SET UP I/O POINTERS
      LI      R12,C1
      LI      R11,>100       TURN ON RTS

XT    MOVB    *R12,R3        GET CONTROL BYTE
      JGT     XT             NOT READY YET
      MOVB    *R0+,*R2       OUTPUT DATA BYTE
      DEC     R1             DECREMENT COUNTER; THRU?
      JNE     XT             NO, CONTINUE
      CLR     *R12           YES, TURN OFF RTS
```

Figure 7-3. Memory-Mapped I/O Example

HOLDA line. At this time, the controller assumes control of the buses and completes its transfer. When the transfer is complete, it releases HOLD. The processor senses this cancellation, deactivates HOLDA, and resumes operation. Figure 7-4 shows a flowchart diagram for this type of DMA operation.

At any time that multiple devices contend for control of a bus, DMA in particular, great care must be given to the ability of the devices to control the bus without interfering with one another. Basically, all potential bus controllers must remain in the high-impedance state until control is granted. Recall that in three-state logic, a driver output may be active high, active low, or high impedance. In the high-impedance state, some other driver determines the signal on the line. In the absence of three-state logic, an open-collector arrangement must be used, with all inactive devices in the high state. This configuration is inferior to three-state logic due to the slower switching speeds available, and termination problems.

In some of its larger systems, Texas Instruments has added a new dimension to the DMA concept, called the TILINE®. With this arrangement, there is a prioritized competition for the bus at all times, and a potential controller has control of the bus only when it needs this control. At other times, the bus is available to other users. For this reason, TILINE-based systems are asynchronous, in that a memory-to-processor transfer, for example, occurs at a rate determined solely by the response time of the memory.

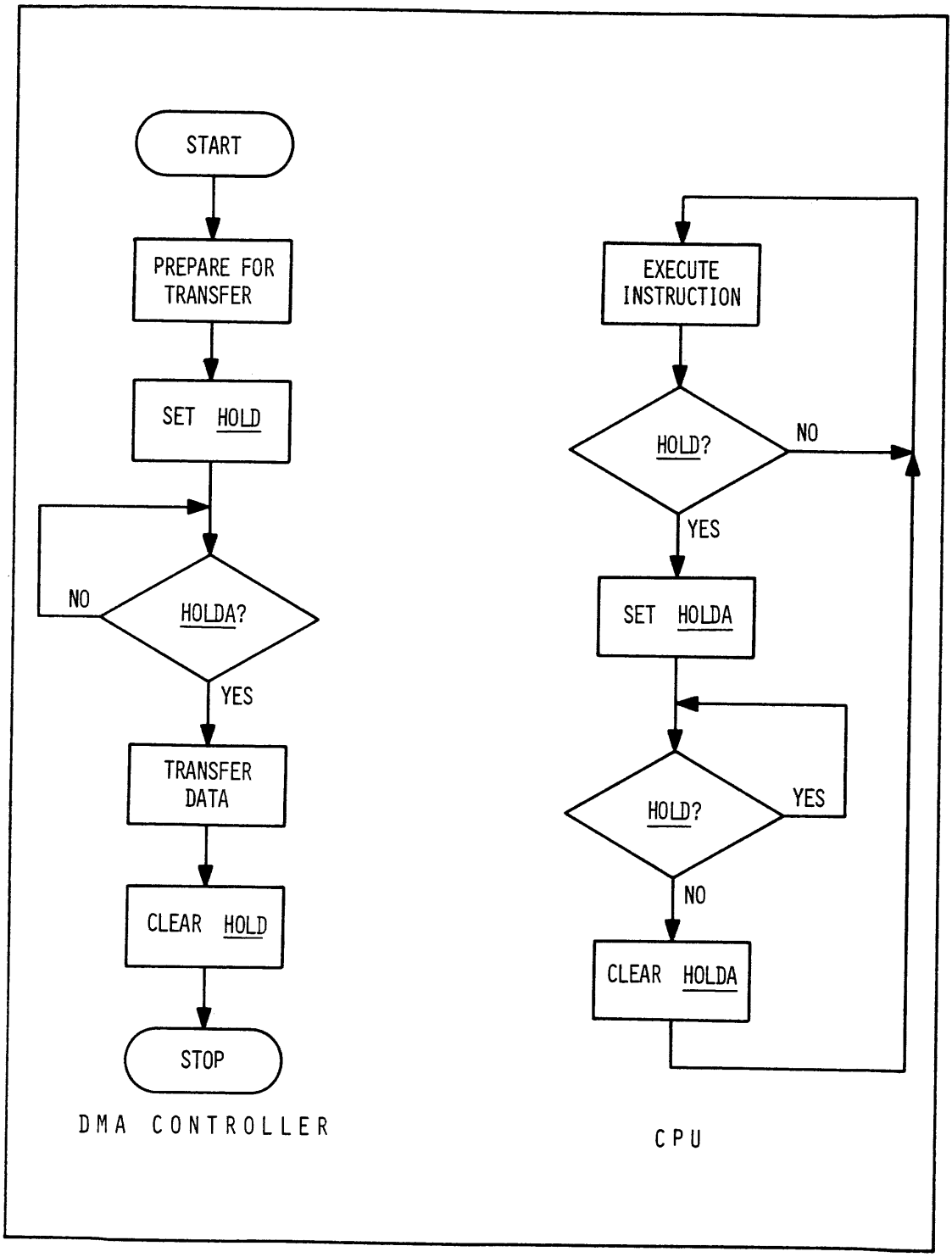


Figure 7-4. DMA Control Flowcharts

Communications Register Unit (CRU)

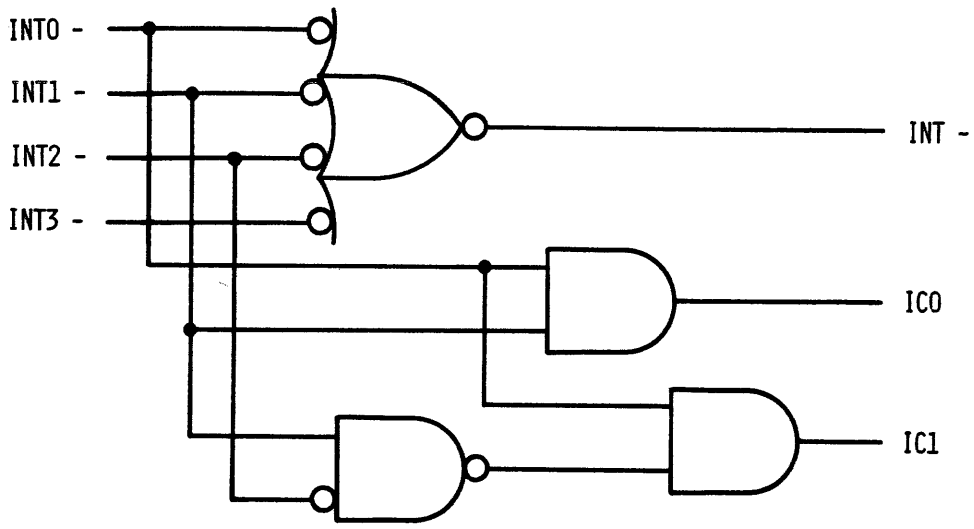
The CRU is a unique I/O port that was developed by Texas Instruments for its process-control minicomputers and has been incorporated into the 9900 family. This subsystem of the microprocessor is a highly flexible and easy-to-use part of the system. It is introduced in the previous chapter. In this chapter the CRU concept is applied to some real problems of interfacing. The discussion examines two particularly useful devices on the University Board.

Recall that the CRU allows single-bit addressing to turn a bit either on or off, or to test a bit. It also allows reading or writing up to 16 bits with one instruction, in the multiple-bit mode. When more than one bit is addressed in a single instruction, the processor sequentially addresses the bits indicated. The I/O interface device must be designed to demultiplex the CRUOUT line on output, and to multiplex the CRUIN line on input.

Even though this discussion centers on the CRU, some time will be taken to describe the interrupt logic implemented on the 9900 series processors, since this is an important form of I/O. The word "interrupt" has been selected, because, no matter what else the processor is doing, it is sometimes desirable to interrupt it to take care of some urgent matter. On the 9900 series microprocessors, a combination of several input lines is used to indicate when an interrupt request is active and to indicate the priority level of the device requesting the interrupt. The number of input lines devoted to interrupts varies with different members of the family. With the TMS 9980A, specifically, there are three input lines. These are used to present to the microprocessor the priority of a device.

Priority is a means of identifying the device requesting an interrupt. During system design, all the devices that might request interrupts are grouped to represent a priority structure. A master reset switch, for example, would be assigned the highest priority in most cases. Lower priority levels would depend upon the application. When a device requests an interrupt, some logic external to the processor must indicate the presence of the interrupt and its priority. Figure 7-5 illustrates such a priority encoder for a four-level system. In Figure 7-5, INT indicates the presence of an interrupt request, while IC0 and IC1 indicate its priority level. The TMS 9980A uses a six-level system with the decoding built into the chip. There might be several devices at the same priority level. In such a case, additional data would have to be available to identify each individual device.

The response of the processor to an interrupt request is to execute what is called a context switch. This is described in greater detail in the next chapter, but its operation is designed basically to save the contents of the current workspace pointer, program counter, and status register, and transfer program control by replacing the workspace pointer and program counter with new contents from two memory words located at a place called a transfer vector. The processor uses the interrupt priority code to point



INT0	INT1	INT2	INT3	INT	IC0	IC1
0	x	x	x	0	0	0
1	0	x	x	0	0	1
1	1	0	x	0	1	0
1	1	1	0	0	1	1
1	1	1	1	1	1	1

Figure 7-5. Four-level Priority Encoder

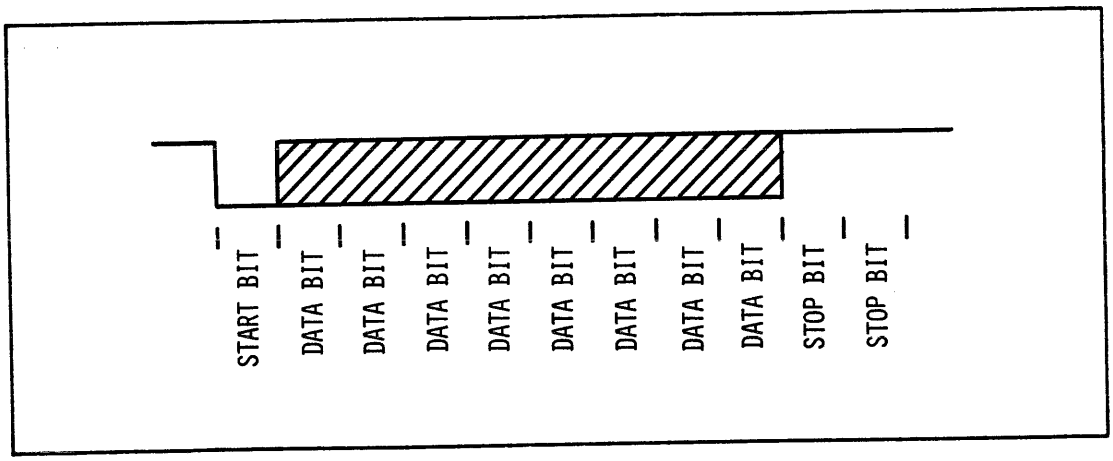


Figure 7-6. Serial Data Format

to the specific transfer vector associated with that priority level. When the context switch is completed, an interrupt processing routine is in control, and manages all details required to respond to the given interrupt. When the routine is completed, the old program counter, workspace pointer, and status registers are restored and processing resumes where it left off. As a part of the interrupt routine, the CRU might be used to indicate to the interrupting device that its request has been processed.

Because of the close relationship between interrupts and I/O, many general-purpose I/O devices have some degree of interrupt-request logic built in. For example, priority encoding is included as part of the logic of the TMS 9901, which is a 9900 family peripheral component. It is described in detail later in this chapter.

There are many standard I/O problems which are amenable to CRU-based solutions. As a means of illustration several are considered in the following paragraphs.

UART--Universal Asynchronous Receiver-Transmitter. This is a general device that interfaces the processor with a relatively slow asynchronous serial data channel. The channel is asynchronous in that there is no synchronizing clock. There is prior knowledge about the data rate, and there are framing bits that allow the receiver to decipher the channel state to determine the character being sent. Figure 7-6 shows the format for transmitting an 8-bit data word. This can be done by the processor with software, but in many applications it is better to use a hardware device to perform the framing and serialization. The TMS 9902 Asynchronous Communications Controller (ACC), another 9900 family peripheral component, is such a device. The CRU interface of the ACC allows single bits to be accessed for control, and multiple bits to be accessed for data input and output. The data channels, transmit and receive, carry the framed and serialized data, such as indicated in Figure 7-6. Details of the operation of the ACC are described later.

A-to-D and D-to-A Converters. The digital computer contains quantities expressed as a collection of ONE's and ZERO's. External devices and peripherals often supply quantities that are continuous in nature, such as the voltage produced by a motor's tachometer, or the resistance of a temperature-monitoring device. Sometimes, external devices require a continuous input control quantity, such as a voltage-controlled frequency generator. For the computer to supply the latter, a digital-to-analog converter (DAC) is required. Conversely, an analog-to-digital converter (ADC) is frequently used to supply data to the computer. Figure 7-7 illustrates a simple DAC, capable of converting 4 bits of data into 16 discrete voltages. Most DAC's and ADC's use 8, 10, or 12 bits of data for much better approximation to true continuous data. ADC's are more complex. Many ADC's use DAC's as part of their structure.

General-Purpose Interface. Each microprocessor system has special requirements; however, rather than design a special CRU interface for each one, it is usually easier to include a general-purpose interface and assign the individual bits to serve the special requirements. The TMS 9901 Programmable System Interface (PSI) is such a general-purpose interface. For a small number of special-purpose control lines, the PSI adequately provides the necessary logic to interface between the CRU and the outside world. Further details on both the TMS 9901 and the TMS 9902 are provided in the next section.

7.3 I/O PERIPHERAL COMPONENTS

The TMS 9901 PSI is included on the University Board while the TMS 9902 ACC can be added as an option. Each device can be programmed to assume a multitude of functions. To use these devices it is necessary to understand their operation and how to program them.

TMS 9901 Programmable Systems Interface

The TMS 9901 Programmable Systems Interface (PSI) is designed to provide a general-purpose I/O, interrupt priority, and interval timer capability to CRU-based microprocessor systems.

Figure 7-8 illustrates the functional arrangement of the PSI and its interface with the CRU and interrupt logic. First consider the interrupt logic of the device. Refer to Figure 7-9. There are 15 pins that can be used for interrupt-request inputs. Associated with each pin is a mask bit that is used either to enable or disable the pin under program control. The prioritizer and encoder selects from among the active interrupt and provides a 4-bit code (IC0-IC3) indicating which one is active, along with the interrupt-request (INTREQ) line. Although the TMS 9901 produces a 4-bit interrupt code, the TMS 9980A microprocessor accepts only three. It is important

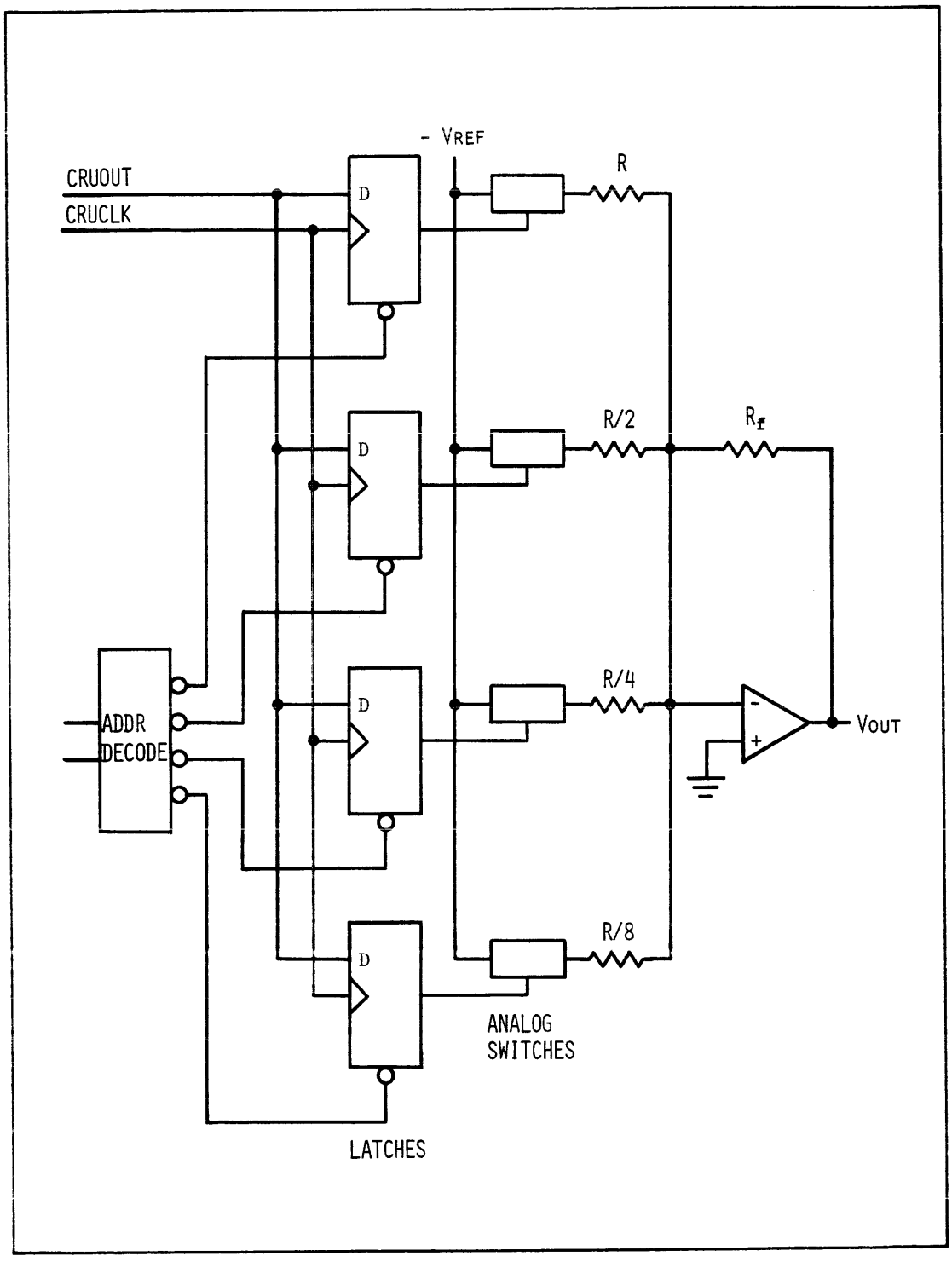


Figure 7-7. Simplified D-A Converter

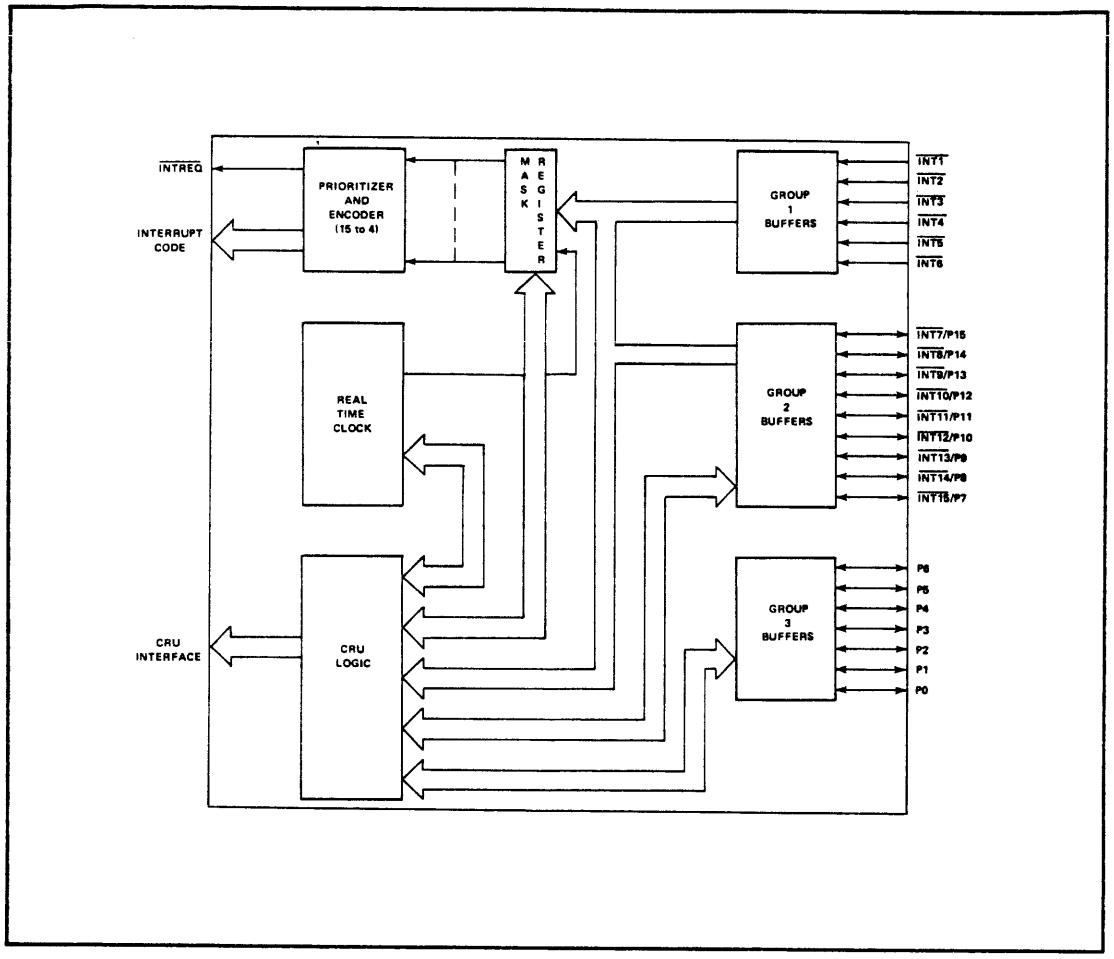


Figure 7-8. Functional Block Diagram, TMS 9901

to understand the relationship between the external interrupt-request lines and the interrupt levels on the TMS 9980A. Table 7-1 indicates the interrupts recognized by the TMS 9980A and the inputs required to activate those interrupts.

Table 7-1. Interrupt Vectors for TMS 9980A

Action	Interrupt Code at TMS 9980A (IC0-IC2)	Transfer-Vector Address	Input at TMS 9901
Reset	000	0000	
Reset	001	0000	INT1-
Load	010	3FFC	INT2-
Level 1	011	0004	INT3-
Level 2	100	0008	INT4-
Level 3	101	000C	INT5-
Level 4	110	0010	INT6-
--	111		

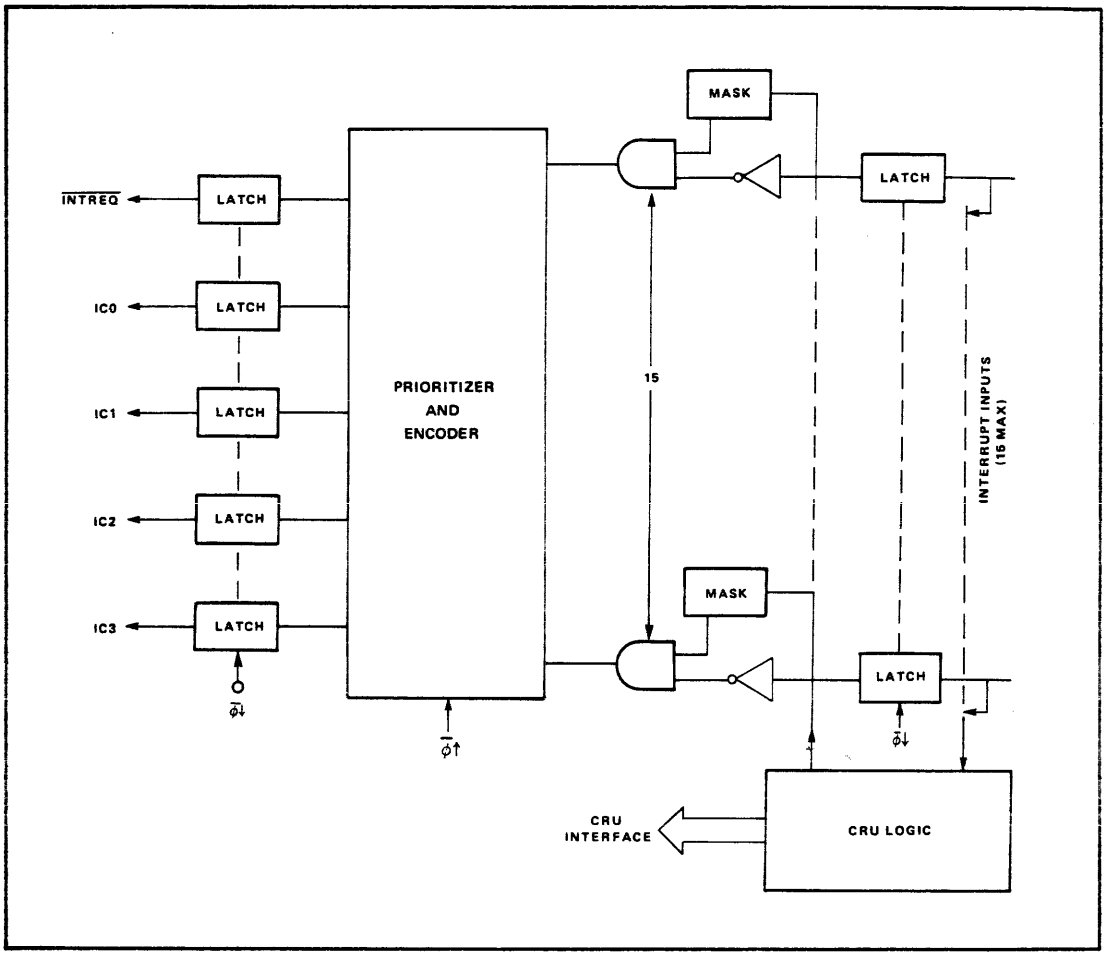


Figure 7-9. Interrupt Control Logic, TMS 9901

Interrupt pins that are disabled by the masks can be used as standard input pins.

To enable an interrupt line, first clear CRU bit ZERO to place CRU bits 1 to 15 in interrupt mode, then write a ONE to the bit associated with the desired interrupt. To disable an interrupt input, write a ZERO to that bit. As an example, the following code enables interrupt levels 1 and 3.

```
LI   R12,CR        SET UP CRU BASE
LI   R1,10         R1 = 0000000000001010
LDCR R1,0          OUTPUT CONTROL BIT AND MASKS
```

After executing the above instructions, bits 2, 4, 5, and 6 are available for input, and bits 16 to 31 are available for input or output.

Table 7-2. TMS 9901 Pin Assignments

CRU Bit	Pin	Usage
7	34	INT7
8	33	INT8
9	32	INT9
10	31	INT10
11	30	INT11
12	29	INT12
13	28	INT13
14	27	INT14
15	23	INT15
23	23	P7
24	27	P8
25	28	P9
26	29	P10
27	30	P11
28	31	P12
29	32	P13
30	33	P14
31	34	P15

There are 16 pins that can be used as standard I/O ports. Table 7-2 identifies these, identifying them by CRU bits and device pins. As inputs they are read via CRU bits 16 to 31. As outputs they are written to via CRU bits 16-31. Each of the pins is defined to be an input until it is written to. At that time it becomes an output port and remains so until a master reset returns all ports to input status, or until an RST2 function of the 9901 occurs. To accomplish this, write a ONE to CRU bit ZERO, then a ZERO to bit 15. All I/O lines then revert to input mode. Do not forget to write a ZERO to the control bit again, CRU bit ZERO, to return the 9901 back to normal operating mode.

TMS 9902 Asynchronous Communications Controller (ACC)

The ACC is another CRU-based interface device that performs serial data transmission and reception for the 9900 family of microprocessors. Although the interface is simple, the device is quite powerful, allowing the user to select the data length, data rate, format, parity, etc. The options available are summarized in Appendix D, to which the reader is referred.

7.4 TIMER OPERATION

Both I/O devices, the TMS 9901 and TMS 9902, have programmable timers. This section descibes their use.

Access to the timer on the TMS 9901 is available through bit ZERO of the TMS 9901. When this bit is set to ONE, writing and

reading bits 1 to 14 refer to the timer value. The timer can be considered to be a 14-bit counter driven by a clock signal with a frequency 1/64 times the Ø clock signal to the chip. Thus, for a system with a 2-MHz clock, the timer is driven by a 31.250-kHz clock. Since 2^{14} = 16,384, the timer then has a maximum interval of about 524 msec and a resolution of about 32.0 µsec.

To initialize the timer, a ONE is written to bit ZERO followed by a 14-bit start count to CRU bits 1 to 14. The programmer must be careful not to write to bit 15, because this is the RST2 function bit. If this is done, a master reset occurs, which may not be desired. The timer automatically starts decrementing from the value entered. When its decrementer register equals ZERO, it issues an interrupt via INT3, which may or may not be enabled. This interrupt is cleared by writing to the mask bit with either a ZERO or ONE. If the timer is to be polled, the control bit, bit ZERO, is set to ONE, and then bits 1 to 14 are read by a STCR instruction. The control bit must then be set back to ZERO to allow normal operation of the device. The clock read register will not be updated by the timer unless the control bit is ZERO. Whenever the timer times out, the INT3 mask bit should be written to. If the software operates in the interrupt mode, the interrupt service routine must set the mask bit.

The timer in the TMS 9902 operates in a slightly different manner. The timer clock is the TMS 9902's internal clock, derived from the external clock. The external clock is divided by either three or four, depending on CLK4M, (bit 3 of the TMS 9902's control register).

If CLK4M is ZERO, the internal clock is the external clock divided by 3. If CLK4M is ONE, the internal clock is the external clock divided by 4. The internal clock is then divided by 64 and applied to an 8-bit counter, which is initially loaded via CRU bits 0 to 7. The internal clock is used by both the interval timer and the serial data interface in the TMS 9902. This means that the software may not indiscriminately alter the status of CLK4M for timer applications, since that would also affect the serial data rate of the device.

Using this timer, with a system clock of 2 MHz, the maximum interval available is

$$Int_{max} = \frac{4}{2 \times 10^6} \times 64 \times (2^8 - 1) = 32.6 \text{ msec.}$$

Similarly, the minimum interval is

$$Int_{min} = \frac{1}{2 \times 10^6} \times 3 \times 64 = 96 \text{ µsec.}$$

Note that these occur for separate values of CLK4M. When the timer times out, TMELP (bit 25) is set, and if TIMENB (bit 20) has been set, then INT (bit 31) and TIMINT (bit 19) are set. Also INT- output is active, and this is passed through the interrupt request logic. Refer to the TMS 9902 ACC Data Manual for details.

7.5 INSTRUCTION SUBSET 5

Conditional Jumps

Many of the instructions presented in earlier chapters resulted in certain bits in the status register being either set or cleared. The six jump instructions covered in this subset provide the programmer with ways to test the status results of those instructions. It is important to remember two points on the use of conditional jumps:

- They test the results of the most recently executed instruction that affected the status register bit being tested.
- They do not themselves affect the status register.

Because of the first point, it is usually wise to put the jump instruction immediately following the instruction the results of which are being tested. Because of the second point, several jumps may be placed in sequence, to effect additional possibilities.

In each of the six conditional jump instructions covered, the displacement field is interpreted exactly as in Chapter 3. Details are provided in Instruction Summaries 7-1 through 7-6.

Byte Instruction

When a byte instruction is executed with a register as an argument, the left byte of the register is used. Obviously, there may be times when the right byte is needed; consequently, the Swap Bytes instruction can be used to satisfy this requirement. See Instruction Summary 7-7.

Logical Instructions

It is sometimes necessary to take the logical or arithmetic negative of a number. The logical negative is usually called the "ONE's complement," and is achieved by simply inverting each bit. This can be accomplished with the Invert Instruction (see Instruction Summary 7-8). Arithmetic negation is "TWO's complement," and the instruction is called Negate (see Instruction Summary 7-9).

The final two instructions are used to initialize words to commonly used values, 0 and -1. Since the hexadecimal representation of -1 is FFFF, the instruction that performs this is called Set to Ones (see Instruction Summary 7-10). Similarly, the instruction that initializes to zero is called Clear (see Instruction Summary 7-11).

JUMP IF LOGICAL LOW

JL

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: JL Location $1A00_{16}$	0	0	0	1	1	0	1	0	Displacement (words)							

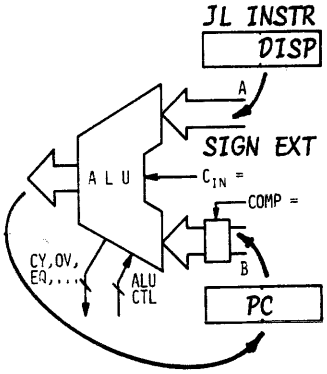
Length: 1 word

RESULT: If L>=0 and EQ=0, (PC) + Displacement in bytes ⟶ (PC)
If L>=1 or EQ=1, (PC) unchanged

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

If the logical greater than status bit (L>) and equal status bit (EQ) are both ZERO, then add the signed displacement in bytes to the program counter. Otherwise, the next instruction in sequence is executed.



NOTES:

Tests status register bits to control program flow. Usually follows immediately an instruction that affects the status register, and JL tests the results of that instruction.

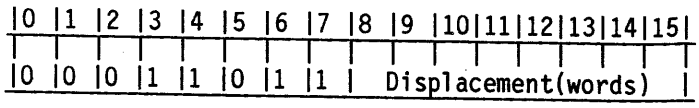
Example:

```
CI R3,>2795
JL LS          JUMP TO LS IF LOW
```

JUMP IF LOGICAL HIGH

JH

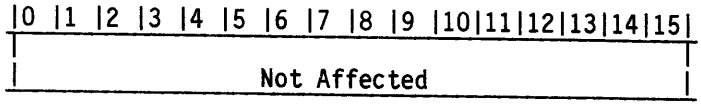
CODE: JH Location



Length: 1 word

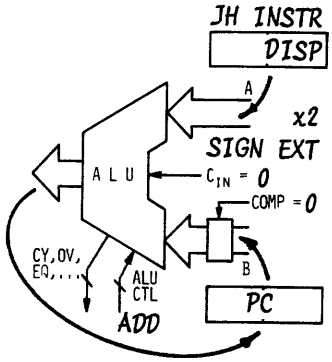
RESULT: If L>=1, (PC) + Displacement in bytes ⟶ (PC)
If L>=0, (PC) unchanged

STATUS REGISTER:



OPERATION:

If the logical greater than status bit (L>) is ONE, then the signed displacement in bytes is added to the program counter. Otherwise, the next instruction in sequence is executed.



NOTES:

Tests status register bits to control program flow. Usually follows immediately an instruction that affects the status register, and JH tests the results of that instruction.

Example:

```
C  R3,R7

JH NE          TRANSFER TO NE IF HIGH
```

JUMP IF LOGICAL LOW OR EQUAL — JLE

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: JLE Location 1200_{16}	0	0	0	1	0	0	1	0	Displacement (words)							

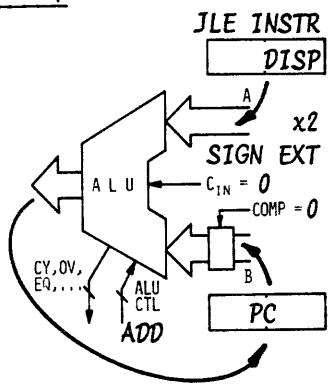
Length: 1 word

RESULT: If L>=0 or EQ=1, (PC) + Displacement in bytes ⟶ (PC); otherwise, (PC) unchanged

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

If the logical greater than status bit (L>) is ZERO, or if the equal status bit (EQ) is ONE, the signed displacement (in bytes) is added to the program counter. Otherwise, the next instruction in sequence is executed.



NOTES:

Tests status register bits to control program flow. Usually follows immediately an instruction that affects the status register, and JLE tests the results of that instruction.

Example:

```
C    R7,R8

JLE  LP      JUMP TO LP IF LOW OR EQUAL
```

JUMP IF LOGICAL HIGH OR EQUAL — JHE

CODE: JHE Location 1400_{16}

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	0	1	0	0	Displacement (words)							

Length: 1 word

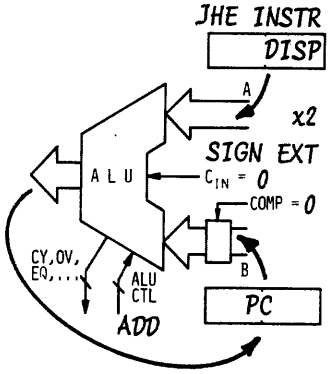
RESULT: If L>=1 or EQ=1, (PC) + Displacement in bytes ⟶ (PC)
If L>=0 and EQ=0, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

If logical greater than status bit (L>) or equal status bit (EQ) is ONE, then add the signed displacement in bytes to the program counter. Otherwise, the next instruction in sequence is executed.



NOTES:

Tests status register bits to control program flow. Usually follows immediately an instruction that affects the status register, and JHE tests the results of that instruction.

Example:

```
CI  R6,>6F6F

JHE LP          TRANSFER TO LP IF HIGH OR EQUAL
```

JUMP IF NO OVERFLOW

JNO

CODE: JNO Location 1900_{16}

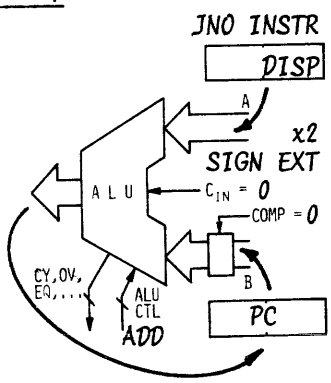
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	0	0	1	Displacement (words)							

Length: 1 word

RESULT: If OV=0, (PC) + Displacement in bytes ⟶ (PC)
If OV=1, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															



OPERATION:

If the overflow status bit is not set, then add the signed displacement in bytes to the program counter. Otherwise, the next instruction in sequence is executed.

NOTES:

Used to test for overflow in previous result. Refer to discussion on binary arithmetic to determine when overflow occurs.

Example:

```
A     R6,R9
JNO   OK          TRANSFER TO OK IF NO OVERFLOW
```

JUMP IF ODD PARITY

JOP

CODE: JOP Location $1C00_{16}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	1	1	1	0	0	Displacement (words)							

Length: 1 word

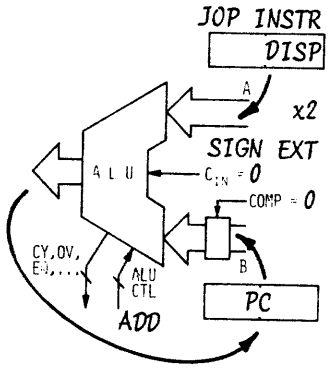
RESULT: If OP=1, (PC) + Displacement in bytes → (PC)
If OP=0, (PC) unchanged

STATUS REGISTER:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Not Affected															

OPERATION:

If odd parity status bit is ONE, then add the signed displacement in bytes to the program counter. Otherwise, the next instruction in sequence is executed. Odd parity results when a byte binary value contains an odd number of ones. This bit is set only by byte operations.



NOTES:

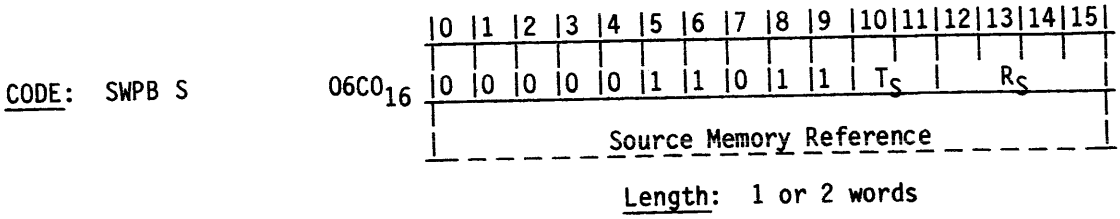
Tests results of last operation affecting OP bit in status register. Used in error-detection applications.

Example:

```
LDCR  R3,8     INPUT 8 BITS FROM CRU
JOP   ER       TRANSFER TO ER IF ODD PARITY
```

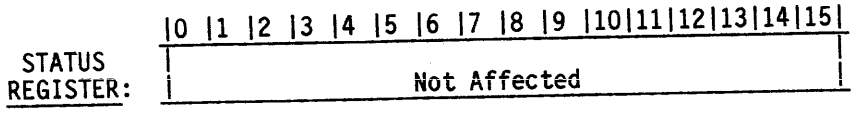
SWAP BYTES

SWPB



Length: 1 or 2 words

RESULT: (S) BYTE 1 → (S) BYTE 2
(S) BYTE 2 → (S) BYTE 1



OPERATION:

Exchange bytes in a word, addressed by any of the five basic addressing modes.

NOTES:

Used to swap bytes within a word. Typically done within a workspace register, since byte instructions on registers always refer to the left byte, whereas memory address bytes are directly addressable.

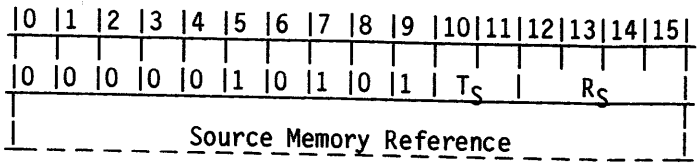
Example:

```
SWPB R3        SWAPS BYTES IN REGISTER 3
```

INVERT

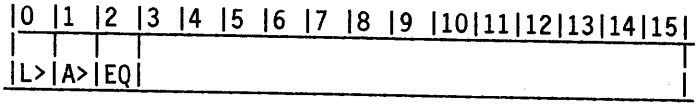
INV

CODE: INV S



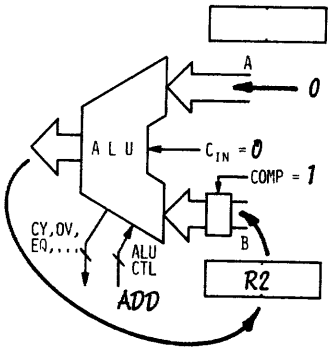
Length: 1 or 2 words

RESULT: $(\overline{S}) \longrightarrow (S)$



OPERATION:

Logically invert each bit in a word, using any of the five basic addressing modes. The result is compared to zero, and the status bits are set accordingly. This is a ONE's complement negation.



NOTES:

Use INV to change all bits in a word. Sometimes, only certain bits are of interest and the others are "don't-cares."

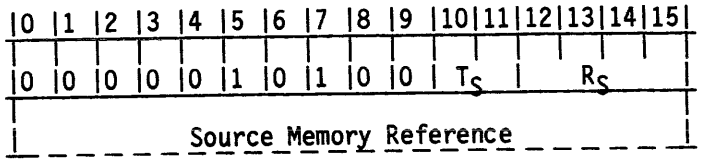
Example:

```
INV  R2      CHANGE EACH BIT IN WORKSPACE REGISTER 2
```

NEGATE

NEG

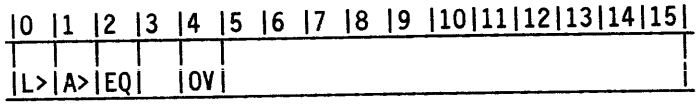
CODE: NEG S



Length: 1 or 2 words

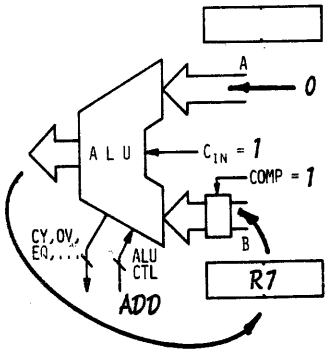
RESULT: -(S) ⟶ (S)

STATUS REGISTER:



OPERATION:

Replace a word with its TWO's complement: invert each bit and add 1 to the result. The result is compared to zero, and the status bits are set accordingly.



NOTES:

Use NEG to obtain the TWO's complement of a number.

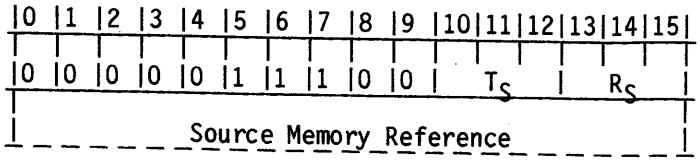
Examples:

NEG R7	WORKSPACE REGISTER 7 NEGATED
NEG @NUMBER	NUMBER NEGATED

SET TO ONES

SETO

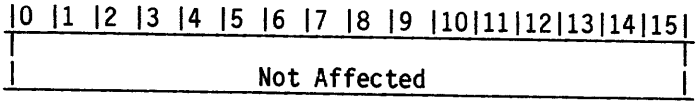
CODE: SETO S



Length: 1 or 2 words

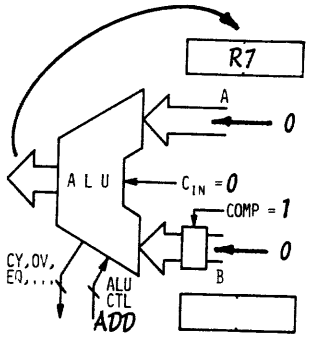
RESULT: FFFF ⟶ (S)

STATUS REGISTER:



OPERATION:

Set each bit in a 16-bit word to ONE. This is the same as setting a signed binary number to -1.



NOTES:

Use SETO to initialize a location with -1. In some cases, this may simplify loop termination logic.

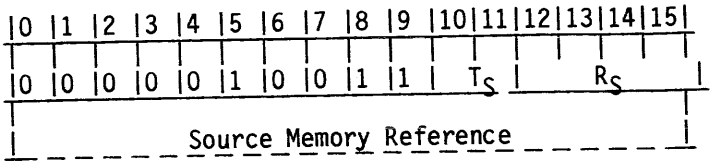
Examples:

```
SETO  7          WORKSPACE REGISTER 7 = -1
SETO  @DATA      DATA = -1
```

CLEAR

CLR

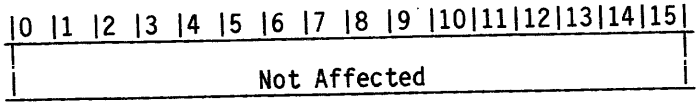
CODE: CLR S $04C0_{16}$



Length: 1 or 2 words

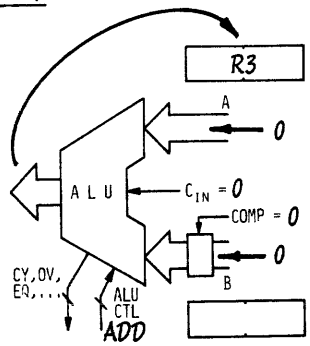
RESULT: 0 ⟶ (S)

STATUS REGISTER:



OPERATION:

Set to ZERO each bit in a 16-bit word.



NOTES:

Use CLR to set to zero workspace registers and general memory.

Example:

CLR 3	WORKSPACE REGISTER 3 = 0000
CLR @CO	CO = 0000

7.6 PROGRAM EXAMPLE: TRAFFIC-LIGHT CONTROLLER

This program demonstrates the use of CRU I/O and the programmable timer on the TMS 9901 using a simplified version of a common application: the control of a traffic light.

Specifications

In real-world traffic control, the traffic-light system usually is quite complex. It must usually control more than one lane in any direction, including separate controls for right-turn-only and left-turn-only lanes. Often it will automatically adjust itself for changing traffic patterns. It should operate in a fail-safe mode (i.e., not all lights green at once). And, in really dense environments such as business areas, many signals should be carefully coordinated.

Nevertheless, the purpose of this example is to acquaint the student with the procedures for using interval timers and I/O in controller applications, rather than to design a complex traffic controller. Therefore, this example of a traffic controller will be simple. Each direction of traffic is controlled by a four-light system: stop, go, caution (CAU), and protected left turn (PLT). A typical four-way intersection is depicted in Figure 7-10 while the sequence of lights for the traffic controller is indicated in Table 7-3. Notice the four-way-stop states which occur immediately following the protected-left-turn states. The four-way-stop states are allocated less time than the caution states, since it is assumed that traffic entering the intersection to turn left will be moving slower than traffic moving straight ahead. Also, notice that state 0 follows state 7.

There are two basic questions to answer before writing the controller program: How should the timing be controlled and how should the output be handled?

Table 7-3. State Sequence, Traffic Light Controller

State	Light (N-S)	Light (E-W)	Time (Seconds)
0	Stop	Stop, PLT	10
1	Stop	Stop	3
2	Stop	Go	20
3	Stop	CAU	5
4	Stop, PLT	Stop	10
5	Stop	Stop	3
6	Go	Stop	20
7	CAU	Stop	5

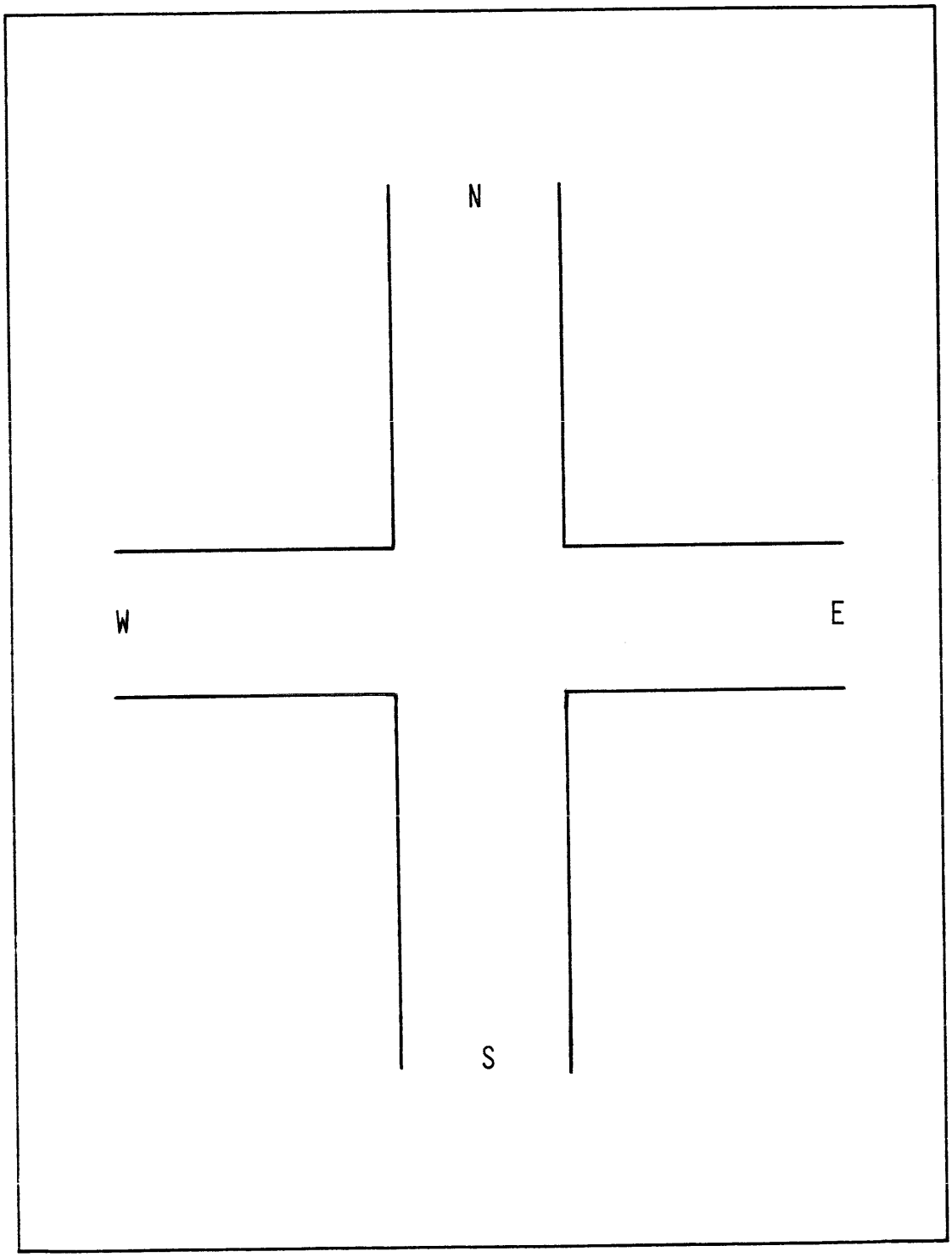


Figure 7-10. Four-way Intersection

In a dedicated system with nothing else to do, the timing could be handled fairly easily in software. For example, as shown in the following program, it takes 200 milliseconds to get from P1 to P2, assuming a 2MHz clock.

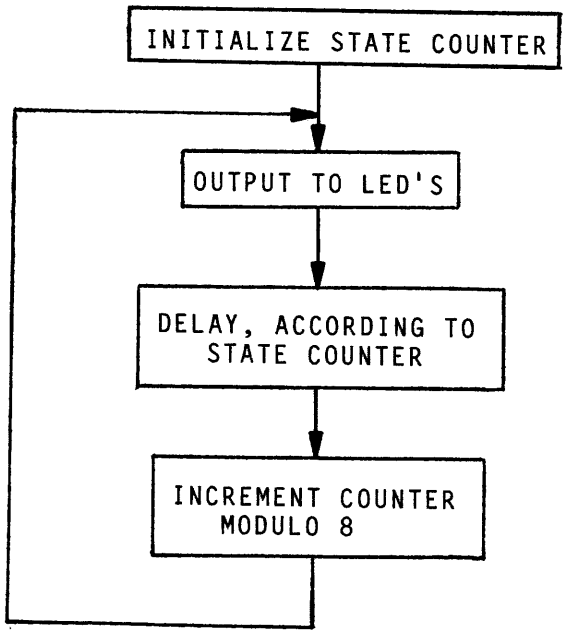
```
P1 LI  R1,14286
LP DEC R1
   JNE LP
P2 EQU $
```

A delay loop such as this might be inserted every time a new state is entered. However, it might be desirable to allow one computer to control several lights, each of which may be running according to different time tables. In this case, a programmable timer is necessary. This example incorporates a programmable timer, although not in a manner that would make most efficient use of it. For such efficiency a discussion of interrupts, subroutines, and context switches would be necessary, but is beyond the scope of this chapter. (These topics are discussed in the following chapter and the reader is encouraged to relate the advantages of such concepts to this example.)

The second problem is the handling of I/O. Normally, the controller drives all light pairs directly, but since only four LED's are included on the University Board, the program outputs only the four signals corresponding to the north-south lanes.

Flowchart

Since the program is not subject to external control, it is a very simple loop:



Program Listing

Writing the program requires the following steps. First, determine the value with which to start the timer. Assume a time unit of 0.20 second. Since the system clock is 2 MHz, and the timer decrements every 64 system-clock cycles, the timer register must be loaded with a value of $0.20\ (2 \times 10^6)/64 = 6250_{10}$. This translates to $186A_{16}$. To write this number into the clock registers, the simplest approach is shift it left one bit, and add a 1 to the least-significant bit. When the result is written to the CRU, the least-significant bit is transferred first. In this case, the LSB is a ONE, placing the TMS 9901 in the clock mode. The remaining bits are loaded into the clock register. See Figure 7-11.

The CRU base address for the user TMS 9901 is 0000. Since the timer uses bits 0 to 15, this base address can be used to access the timer. To write all four LED control bits at once, an LDCR instruction is used. The use of this instruction requires that the CRU base address be the address of the lowest-order bit to be written. The four LED's are connected to CRU bits 16 to 19 (10_{16} to 13_{16}). Therefore, the CRU software base address to be loaded into R12 is $2 \times 10_{16} = 20_{16}$.

The two equivalences given below simplify the notation in the code.

```
DL   EQU  >30D5    ENABLE AND START TIMER
U9   EQU   0       CRU BASE ADDRESS FOR TIMER
```

To simplify the program, two tables of data are used, one to store delay counts, which are equal to five times the number of seconds to be delayed (since the timer interval is 0.200 second), and one to store LED assignments. The LED assignments are are given in Table 7-4.

Table 7-4. LED Assignments and Pattern

Assignment	LED Pattern
P0 (Bit 10_{16}) = Stop	0001
P1 (Bit 11_{16}) = Caution	0010
P2 (Bit 12_{16}) = Go	0100
P3 (Bit 13_{16}) = Protected Left Turn	1000

To initialize the tables, the DATA assembler directive is used to provide 16-bit word quantities. Since the program calls for data by bytes, some preparation is required to initialize the data. The first table contains the number of 200 msec intervals required for the delay. The first delay is for ten seconds, for a count of 50. Since $50_{10} = 32_{16}$ the first byte is >32. Similarly, the second is >0F. Therefore, the first word in the delay table is

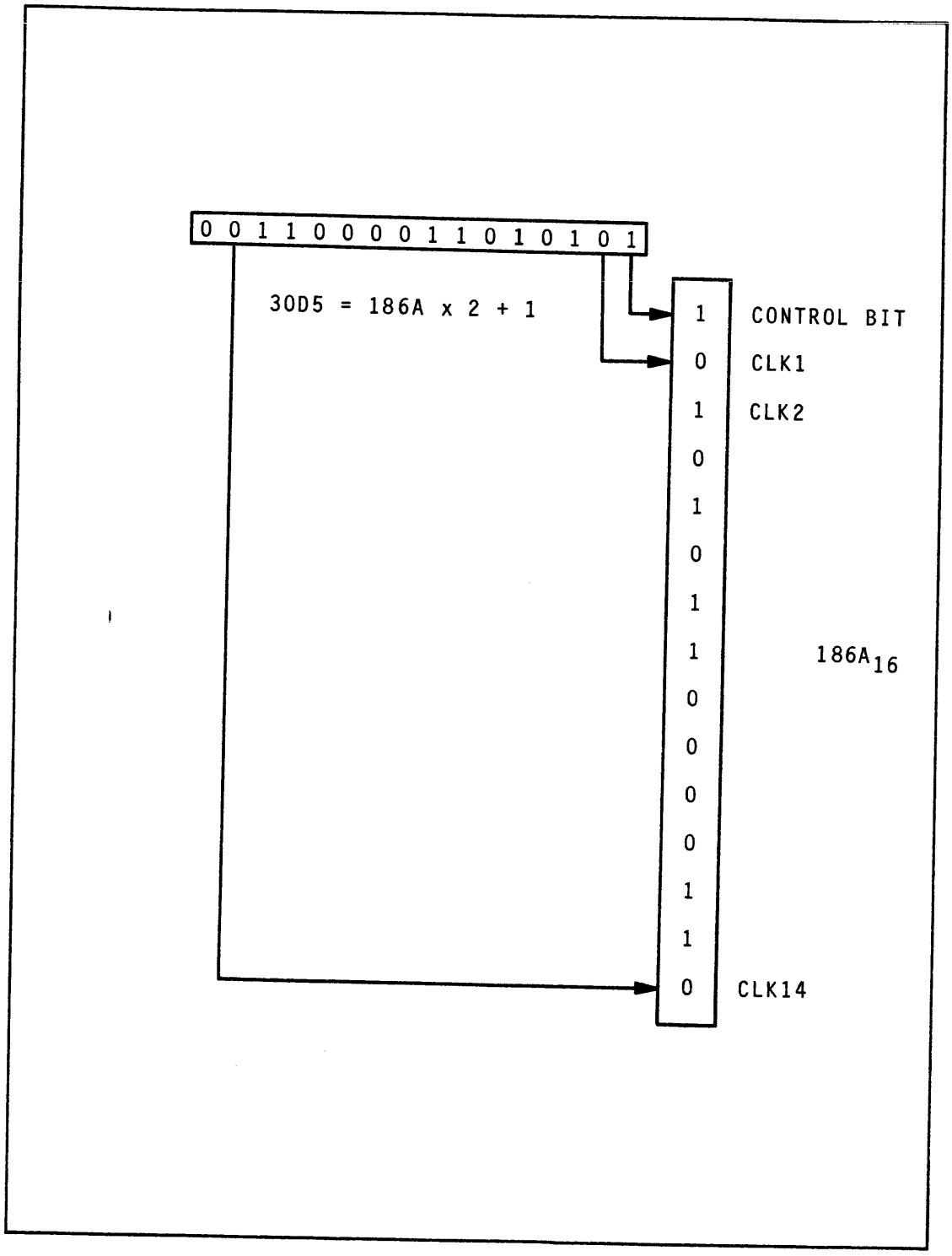


Figure 7-11. CRU Bit Assignments

```
0001 0200                      AORG >200
0002       0000  R0            EQU  0              REGISTER EQUATES
0003       0001  R1            EQU  1
0004       0002  R2            EQU  2
0005       000C  R12           EQU  12
0006       30D5  DL            EQU  >30D5          (.2*2E6/64)*2+1
0007       0000  U9            EQU  0              USER 9901 CRU ADDRESS
0008 0200          WP          BSS  32
0009 0220 320F     TB          DATA >320F,>6419
     0222 6419
0010 0224 320F                 DATA >320F,>6419
     0226 6419
0011 0228 0101     LD          DATA >0101,>0101
     022A 0101
0012 022C 0901                 DATA >0901,>0402
     022E 0402
0013 0230 02E0     ST          LWPI WP             SET WORKSPACE POINTER
     0232 0200
0014 0234 0300                 LIMI 0              DISABLE INTERRUPTS
     0236 0000
0015                           *
0016                           *   SET 9901 TO INTERRUPT MODE AND ENABLE
0017                           *   CLOCK INTERRUPT, LEVEL 3
0018                           *
0019 0238 020C                 LI   R12,U9
     023A 0000
0020 023C 0A1C                 SLA  R12,1
0021 023E 1E00                 SBZ  0
0022 0240 1D03                 SBO  3
0023                           *
0024                           *   INITIALIZE DELAY AND STATE REGISTERS
0025                           *
0026 0242 0202                 LI   R2,DL
     0244 30D5
0027 0246 04C1                 CLR  R1
0028                           *
0029                           *   MAIN LOOP: OUTPUT LED PATTERN CORRESPONDING
0030                           *    TO STATE REGISTER
0031                           *
0032 0248 020C     S2          LI   R12,U9+16
     024A 0010
0033 024C 0A1C                 SLA  R12,1
0034 024E 3121                 LDCR @LD(R1),4      OUTPUT PATTERN
     0250 0228
0035                           *
0036                           *   SET UP FOR OUTER TIMING LOOP
0037                           *
0038 0252 D021                 MOVB @TB(R1),R0     GET DELAY COUNT
     0254 0220
0039 0256 0980                 SRL  R0,8                SHIFT TO L.S.BYTE
0040                           *
0041                           *   SET UP FOR INNER TIMING LOOP
0042                           *
0043 0258 020C                 LI   R12,U9
     025A 0000
0044 025C 0A1C                 SLA  R12,1
0045 025E 33C2     TM          LDCR R2,15               SET & START TIMER
0046                           *
0047                           *   INNER TIMING LOOP
0048                           *
```

Figure 7-12. Program Listing, Traffic Light Controller

```
                SDSMAC      3.1 *    09:37:35 FRIDAY, DEC 22, 1978
                                                              PAGE 0003
 0049 0260 1F0F  TS      TB   15                 TIMED OUT?
 0050 0262 16FE          JNE  TS                  NO, WAIT
 0051              *
 0052              *     INNER LOOP COMPLETE
 0053              *
 0054 0264 1E00          SBZ  0                  BACK TO INTERRUPT MODE
 0055 0266 1D03          SBO  3                  CLEAR TIMER OUTPUT
 0056 0268 0600          DEC  R0                 DELAY COMPLETE?
 0057 026A 16F9          JNE  TM                  NO, REPEAT INNER LOOP
 0058 026C 0581          INC  R1                 INCREMENT STATE REGISTER,
 0059 026E 0241          ANDI R1,7                 MODULO 8
      0270 0007
 0060 0272 10EA          JMP  S2                 REPEAT FOR NEXT STATE
 0061      0230          END  ST
NO ERRORS
```

Figure 7-12. Program Listing, Traffic Light Controller (Continued)

>320F. The following two lines of assembly-language directives set up the delay table.

```
TB DATA >320F,>6419
   DATA >320F,>6419.
```

The table of delay counts must contain the sequence of decimal numbers 50, 15, 100, 25, 50, 15, 100, 25. From Table 7-3, the first time delay in the sequence is 10 seconds. To effect this delay, a counter will be set to count 50 iterations of a 200-millisecond delay, since 50 x 0.200 = 10. The entire sequence of state delays from Table 7-3 is realized by a sequence of decimal numbers 50, 15, 100, 25, 50, 15, 100, 25. With the LED output port assignments indicated above, observe that outputting the four bits representing a BCD value of 1 turns the stop light on and the others off. Therefore, the first byte of the LED table is >01. Reference to Table 7-3 shows that the first four bytes are all >01. The fifth byte should be a binary 1001, with STOP and PLT bit. This process is continued on through the table. Thus, two assembly-language statements can be used to set up the LED table:

```
LD DATA >0101,>0101
   DATA >0901,>0402.
```

The remainder of the program is given in Figure 7-12, with appropriate comments. The program can be assembled and then executed with the E command (the program counter will be loaded with the program entry vector via the END directive). The user can load the object (column 3 of the listing) at the memory address (column 2) and execute with the entry address (0230) in the program counter.

7.7 SUMMARY

In this chapter, a detailed description of input/output techniques is presented, with emphasis on two of the standard peripheral interface devices in the 9900 series of microprocessors. Descriptions of memory-mapped I/O, direct memory access, and the communications register unit techniques are given, with some examples. Use of the TMS 9901 is described in detail, with a program example given to illustrate its operation.

Instruction Subset 5 is covered. The remainder of the conditional jump instructions are described as well as some useful logical instructions. The Swap Bytes instruction, useful in conjunction with the other byte instructions, is also discussed.

With the new information provided in this chapter, the programmer can proceed to adapt the TM 990/189 microcomputer to an endless variety of tasks requiring connection to the outside world. The real power of microprocessor can thus be realized.

7.8 EXERCISES

1. Assume a device is connected at memory address >F004 and is capable of supplying 16 bits of input and 16 bits of output. Write the instructions necessary to

   (a) Read the value at bit #6
   (b) Write a 7(0111) to bits 3-6
   (c) Pulse (set and reset) bit 15.

2. Assume an 8-bit memory-mapped I/O device is connected at address >F009. Write the code required to output bits 3-10 of R7.

3. How long would it take a DMA device to load memory from >0100 to >477F at a rate of 50 kilobytes/sec?

4. A tape transport is connected to a computer memory via DMA. It takes 0.045 second to ramp up to speed at 37.5 inches per second. If the data density is 1600 bytes/inch, how long would it take (seconds) to dump memory from locations 0 to >7FFF?

5. A TMS 9901 device is to be used as the time base for a tone generator. The timer must deliver an interrupt every half-cycle of the output tone. Assuming a clock rate of 2 MHz, what should the hex value of the timer register be for tones of

   (a) 1 kHz
   (b) 100 Hz
   (c) 10 kHz.

How accurate would these be as frequency standards?

6. A TMS 9901 device is to be set up for the following I/O and interrupt requirements. Write the instructions to do this, assuming an absolute CRU address of >20:

Pin #	Use
17	$\overline{\text{INT 1}}$
18	$\overline{\text{INT 2}}$
9	$\overline{\text{INT 3}}$
8	IN
7	IN
6	IN
38	IN
37	IN
26	OUT
22	OUT
21	OUT
20	OUT
19	IN
23	IN
27	OUT
28	OUT
29	IN
30	IN
31	IN
32	IN
33	IN
34	IN

7. A TMS 9902 ACC device is to be used to transmit Baudot teletypewriter code. This code uses one start bit, one stop bit, five data bits, and no parity. Assume a system clock of 2 MHz and write the code to set up the ACC for this communications mode for a baud rate of 75 baud.

8. Write the instructions to program the TMS 9902 (in place of the 9901) to generate the tones required in exercise 5.

7.9 LAB EXPERIMENTS

1. Using the on-board LED's, design an automobile turn indicator that turns the LED's on and off according to the following tables.

Phase	Left Turn	Right Turn
0	0000	0000
1	000●	●000
2	00●●	●●00
3	0●●●	●●●0
4	●●●●	●●●●

It should cycle the left-turn pattern four times then return to the monitor. The user can then set a register to cycle through the right-turn pattern four times. Select an optimal cycle period by trial and error.

2. Design a missile launch-control countdown timer. Display the BCD value of each second from 10 down to 1. When zero is reached, flash all four lights together on and off at a rate of 10 flashes per second.

3. Modify the example program to simulate rush-hour traffic timing requirements. Devise a suitable means of indicating rush hour, and change the timing so that state 2 is held for 10 seconds and state 6 for 30 seconds.

4. Develop a four-phase stepper motor driven simulator which can control a stepper motor with the phase sequence shown below. Use the on-board LED's to indicate the current phase pattern.

	CW	CCW
STEP	PHASES	PHASES
1	1 0 1 0	0 1 1 0
2	1 0 0 1	0 1 0 1
3	0 1 0 1	1 0 0 1
4	0 1 1 0	1 0 1 0

Output the patterns to the LED's (CRU address 10_{16} - 13_{16}) and hold for five seconds to allow checkout. For an audible indication that the pattern is changing, execute BLWP @>300C immediately prior to the LDCR instructions.

CHAPTER 8

MODULAR PROGRAMMING

8.1 INTRODUCTION

This chapter is a study of a basic programming technique, the construction of subroutines. The judicious use and structuring of subroutines is a fundamental principle of effective programming.

In relation to subroutines, the concept of a program context is explored as is the importance of context switching. The special features of the 9900 family that implement this concept are described in detail.

This chapter introduces the last subset of the TMS 9980A's instructions. Included in this subset are instructions that are especially useful in writing subroutines and performing a context switch. Also in this subset is a pair of instructions that have the power of subroutines, the 9900 family's multiply and divide instructions.

This chapter also introduces the subroutines residing in the TM 990/189 UNIBUG monitor that may be employed by a user's application program to perform useful functions such as inputting or outputting characters.

As in previous chapters, a specific program is developed. This program illustrates the use of subroutines and the concept of context switching.

8.2 PROGRAM MODULARITY CONCEPTS

Definition

Up to this point, programs have been discussed and developed on a "stand-alone" basis; that is, the programs were constructed to exist on their own without direct interaction with any other program. In most applications, however, a program does not exist alone. An application is normally composed of several program segments, each of which performs an individual task within the complete application program. Each program segment is a "module" or a functional piece of the entire program structure.

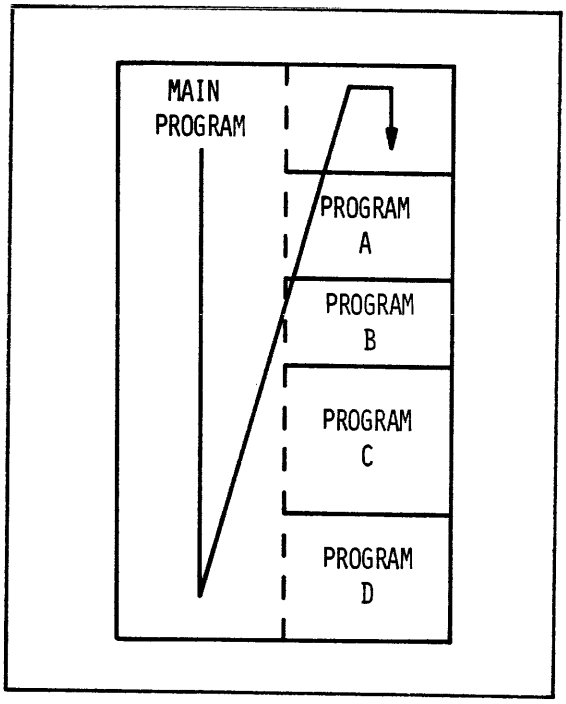


Figure 8-1. A Collection of Program Modules in an Application Program

Advantages

By structuring programs into functional modules, several advantages are gained. First, there is memory efficiency. A function can be defined once in an individual program module. Whenever that function is needed in the overall application program, that one program module can be called to perform its task. This is more memory efficient than rewriting the function every place it is required. Figure 8-1 illustrates this relationship between the collective program segments or modules in an application program.

A second advantage of program modularity lies in ease of debugging. It is easier to debug the smaller program modules than to debug a large complete application program. The debugging task can be limited to smaller programs that are designed to perform a limited function. The individual program module can be tested and proved correct before it is added to the overall application.

Furthermore, program modularity allows the programmer to exercise a greater degree of flexibility. Program segments can be written to perform a given function. If these functions need to be changed, only the individual program segments need to be changed. If functions must be deleted or added, individual program segments can be deleted or added. Also, the modules created for one program might be needed in subsequent programs. They could then be "dropped" into the new programs as needed, with minimal or no restructuring.

It is feasible that a user could have a selection of program modules to choose from as an application is being designed. He could identify the functions needed by the application program, choose the program modules to accomplish these functions, and then simply piece them together with interfacing code, much like a hardware designer now designs with TTL building blocks.

8.3 SUBROUTINES

The most common example of the implementation of program modularity is the subroutine. The use of subroutines promotes modularity and economy of memory and allows applications to be debugged more easily.

Definition

A subroutine is a closed set of instructions (and perhaps data areas and constants) which is called by another program or subroutine for the purpose of performing a specific task. The decision of defining a piece of code as a subroutine is sometimes arbitrary, but normally a piece of code is a candidate for becoming a subroutine when its function is required at more than one point in an application.

Usages

Chapter 5 explains that the programs in a system's memory are grouped into three broad classifications: a supervisor module, one or more I/O modules, and one or more task modules. Normally it may be expected that each I/O module will be a separate subroutine devoted to the needs of a specific I/O device or I/O devices of like characteristics. Likewise the task modules will normally be subroutines or a collection of smaller subroutines. It is possible to have subroutines inside subroutines. A subroutine may call other subroutines which call other subroutines in turn, and so forth.

Characteristics

A subroutine is limited in size and, normally, is defined for the purpose of performing a specific function or, possibly, several functions. In most cases, though, the function of the subroutine should be limited to one specific task.

A subroutine is defined to be a function which is needed at several places in the overall application. Therefore, a subroutine can be called from several different places in the application program. Because a subroutine can be called from several places, it tends to be more general in nature than the in-line code it replaces.

As illustrated in Figure 8-2, a subroutine has an entry point and an exit point with a body of code in between to perform the assigned function.

Entry Point. The entry point of a subroutine normally is the first executable instruction in the subroutine. It is possible to design a program segment with more than one entry point, but good programming practice limits a subroutine to only one entry point. With only one entry point, the ease of debugging is greatly improved.

Exit Point. In addition to an entry point, a subroutine has an exit point where control is returned to the program that called it. Again, as with entry points, a subroutine could be designed to have more than one exit point, but good programming discipline limits a subroutine to only one exit point.

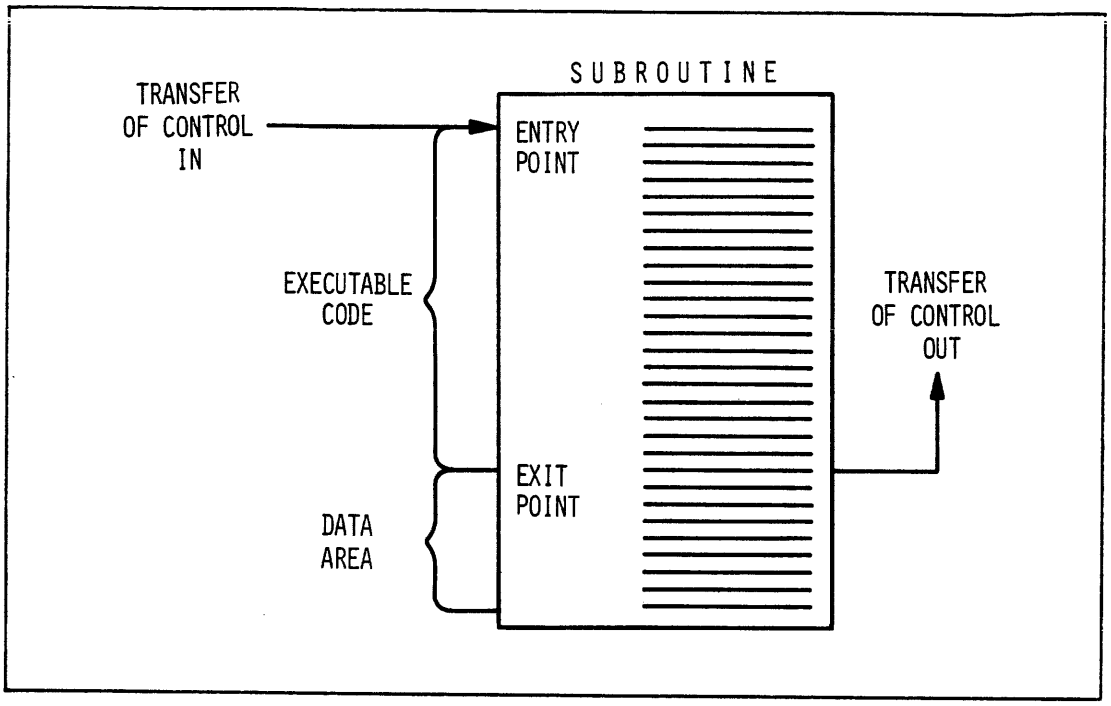


Figure 8-2. The Structure of a Subroutine

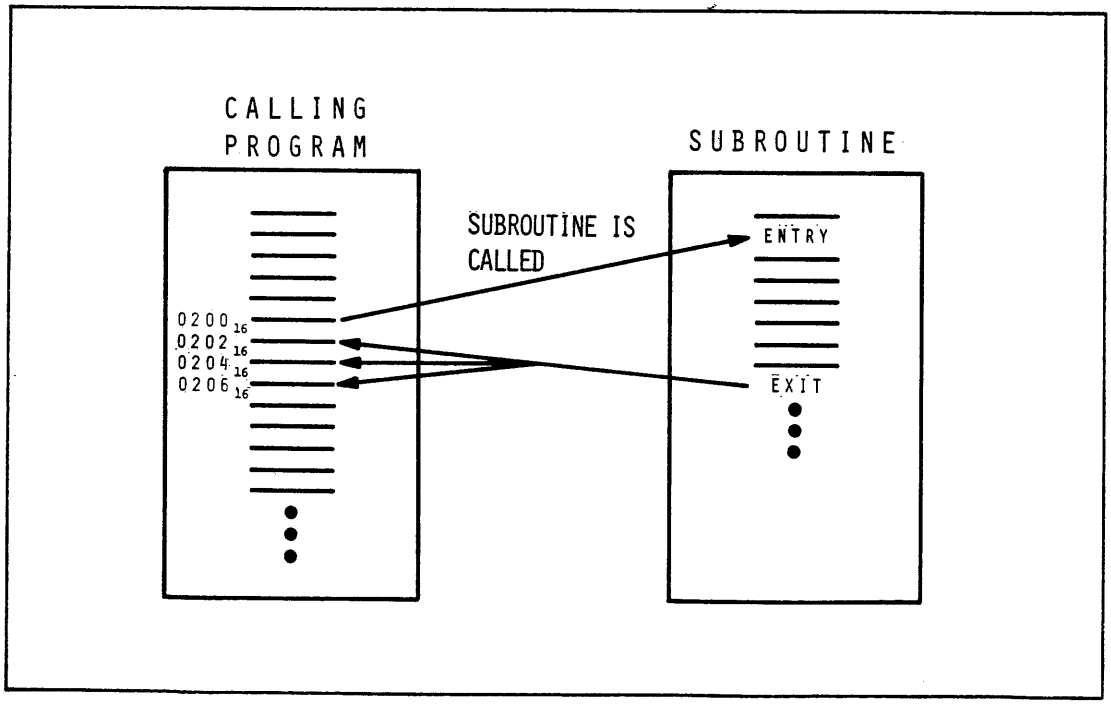


Figure 8-3. A Subroutine Can Return Control to One of Several Places in the Calling Program

Because a subroutine can be called from several different places in the application program, a mechanism must exist to remember the point from which it was called so that control can be returned to the specific program that called it. Virtually every processor--whether it be a big main frame, a minicomputer, or a microprocessor--has an architecture and instruction set that allows subroutines to be called and the return address saved in an area available to the subroutine.

Depending upon the results desired, the subroutine can be designed to return control to any one of several places. As shown in Figure 8-3, a program is called from location 0200_{16}, but the construction of the subroutine is such that a return to the calling program can be at locations 0202_{16}, 0204_{16}, or 0206_{16}. The specific location to which the subroutine returns control in the calling program depends upon the results produced by the subroutine. An example of this will be seen later in this chapter when the UNIBUG monitor calls are discussed.

The return address can be considered to be a piece of data passed to the subroutine.

Data Passing. A subroutine processes data and normally produces action and/or other data needed by the calling program or a subsequent program. For example, a subroutine may perform the function of determining the average value of a given set of values. The subroutine receives the individual values as input data and produces the average value as output data.

Conventions have been adopted for data to be passed to subroutines from a calling program and to allow data to be returned to a calling program. Some of the more common methods include

- Passing the data via working registers
- Passing the address of the data
- Passing the data in defined areas of memory
- Passing the data on a common stack, if there is one.

Data passed to the subroutine normally includes the return address, which is needed to return control to the correct program module.

As a subroutine is designed, the programmer must decide how the input data is to be passed to the subroutine and then write the subroutine to accept the data in this manner.

If a stack is available, the convention may be adopted to place the data on the stack prior to calling the subroutine. Once the subroutine has been called, it can extract the data from the stack, calculate results, and, perhaps, place the results back on the stack before returning control to the calling program. This technique is illustrated in Figure 8-4. In this figure, the calling program places data on the stack (circle 1) and then calls the subroutine

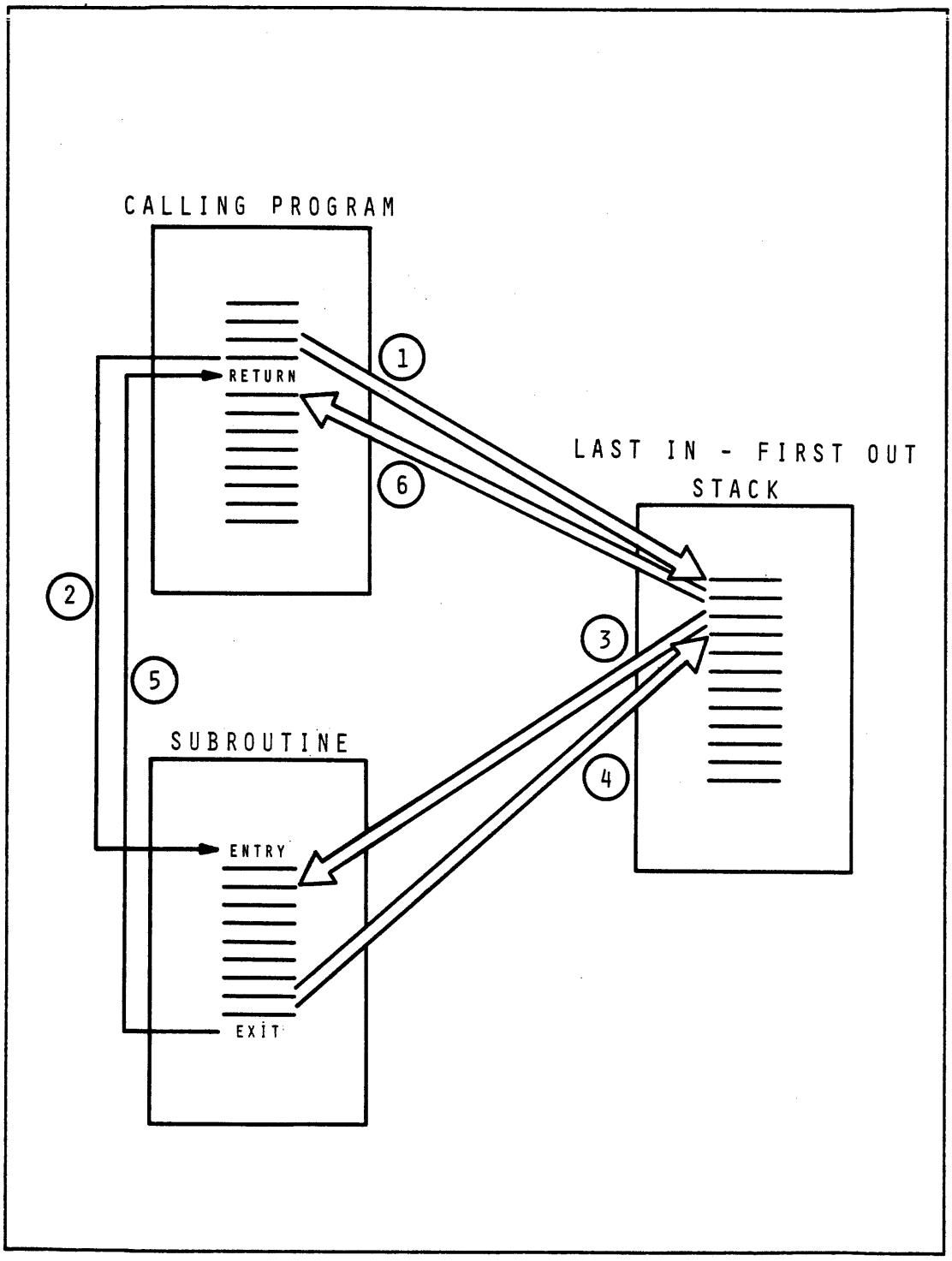


Figure 8-4. Passing Data on a Common Stack

(circle 2). The subroutine removes the data from the stack (circle 3), processes it, and places the result back on the stack (circle 4). The subroutine returns control to the main program (circle 5) which then extracts the result from the stack (circle 6).

If a stack is available, it is very convenient to at least pass the return address to a subroutine on the stack. A microprocessor may have in its instruction set one or more instructions that cause a transfer of control to a subroutine while automatically placing the return address on the stack. Likewise, there may be an instruction that allows a subroutine to automatically take a return address from a stack and return control to that address.

Another common technique (as shown in Figure 8-5) is to specify that data will be passed in specific, defined locations in memory. For example, a subroutine, when it is entered, may be designed to always look for data in memory location 100_{16}. Likewise, the subroutine may pass data back to the calling program by placing it in a specific, defined location.

In Figure 8-5, the main program places a data value in memory location X (circle 1) and calls the subroutine (circle 2). The subroutine reads the data (circle 3), processes it, places the result in memory location Y (circle 4), and returns control to the main program (circle 5). The main program reads the result in memory location Y (circle 6) and continues its processing.

An area of memory used to pass data between individual programs is called a "buffer." Rather than using one area of memory as in the previous example, the program that gathers the data may have several memory areas, or buffers, to choose from. One is chosen. Data is gathered and placed in it. When the program calls the subroutine, the program passes the address of the buffer to the subroutine.

This method is shown in Figure 8-6. In some cases, the contents of the buffer may be a collection of addresses which point the subroutine to the data.

Another method of passing data (including return addresses) is to use the working registers. As shown in Figure 8-7, data can be passed to the subroutine in the registers and the subroutine can return other data in the registers.

The architecture of the 9900 family of microprocessors provides for the use of more than one set of working registers. A subroutine can use the same set of working registers as the program which called it, or, there is the option for the subroutine to use its own set of registers, separate from the calling program.

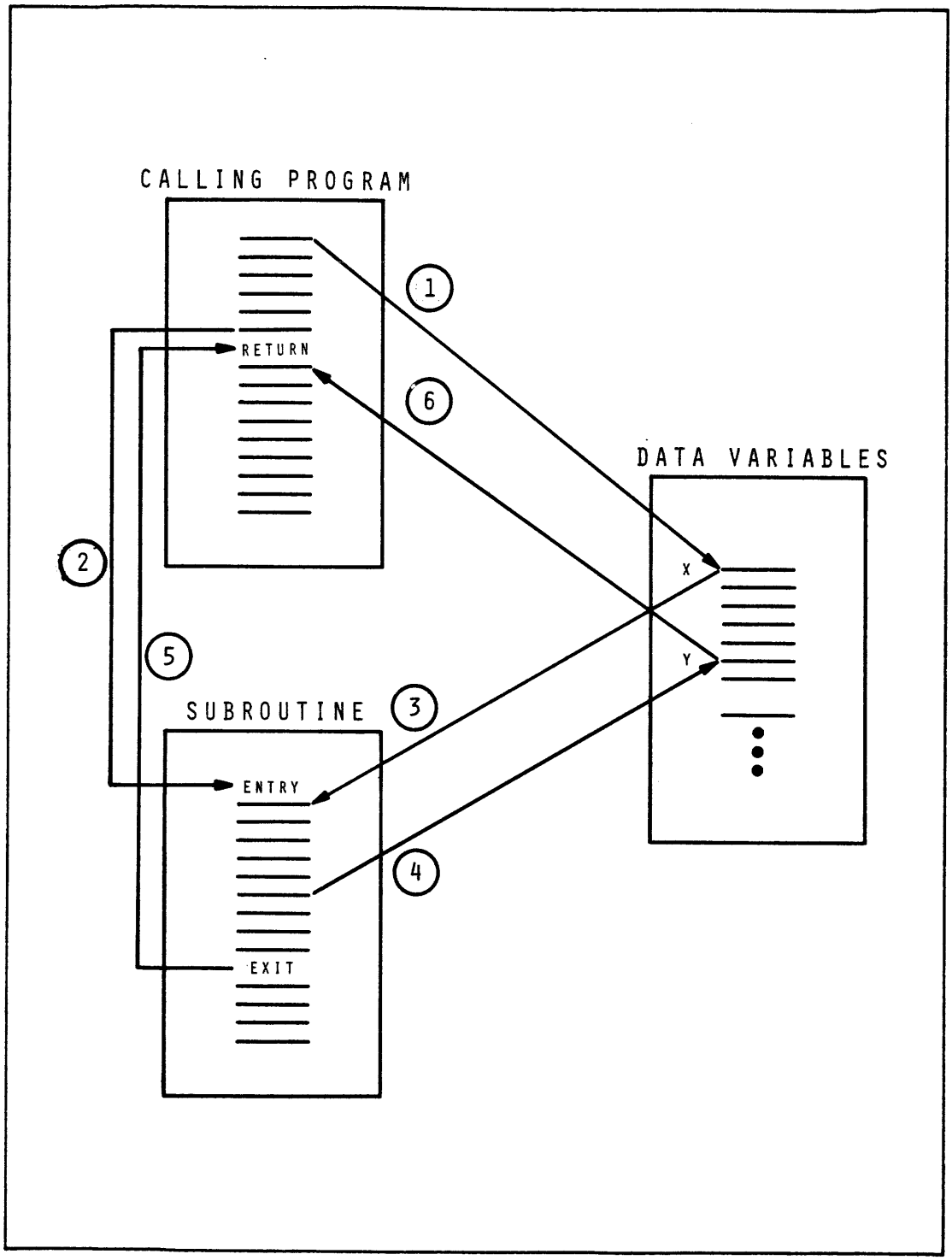


Figure 8-5. Passing Data in a Defined Area of Memory

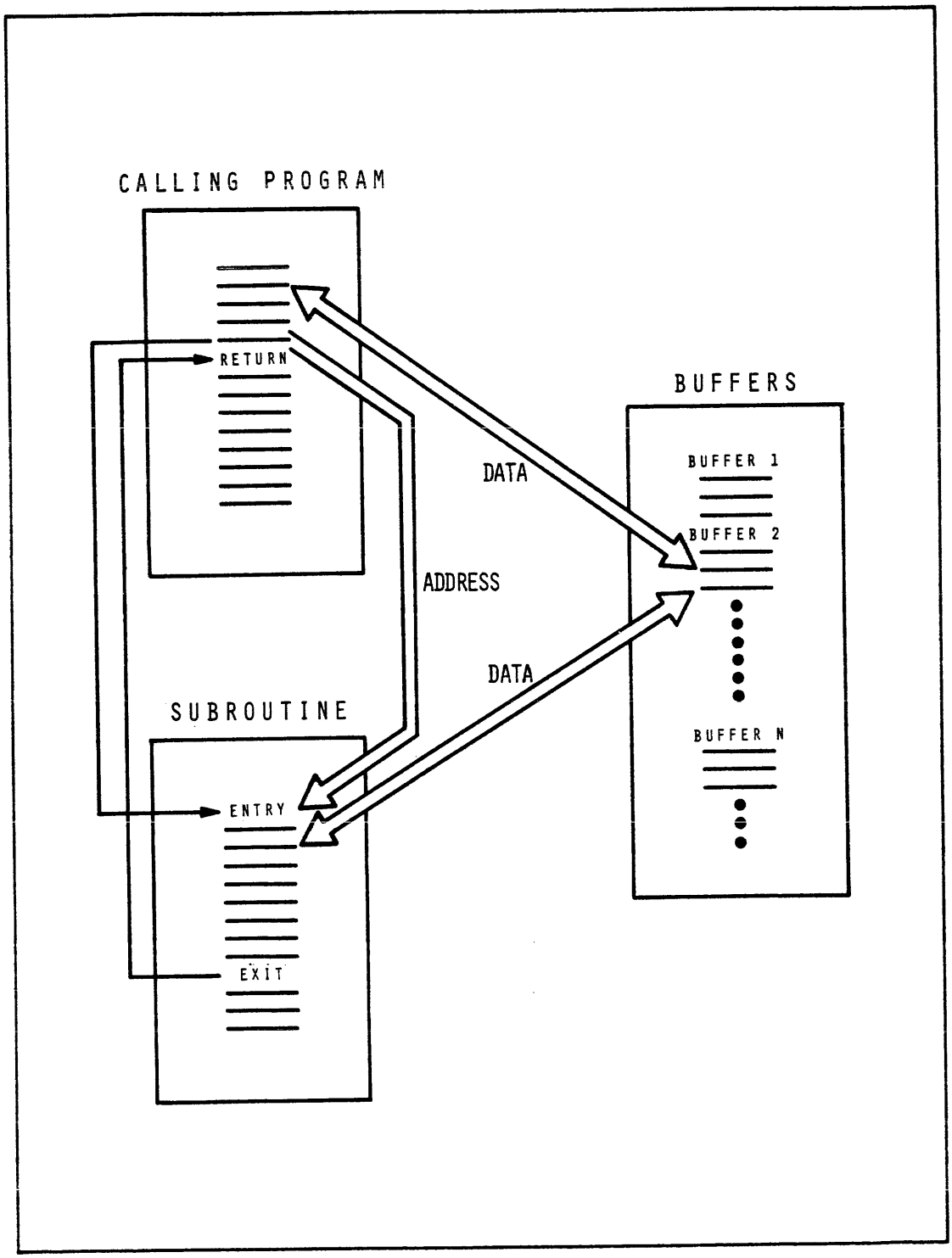


Figure 8-6. Passing Data in a Buffer

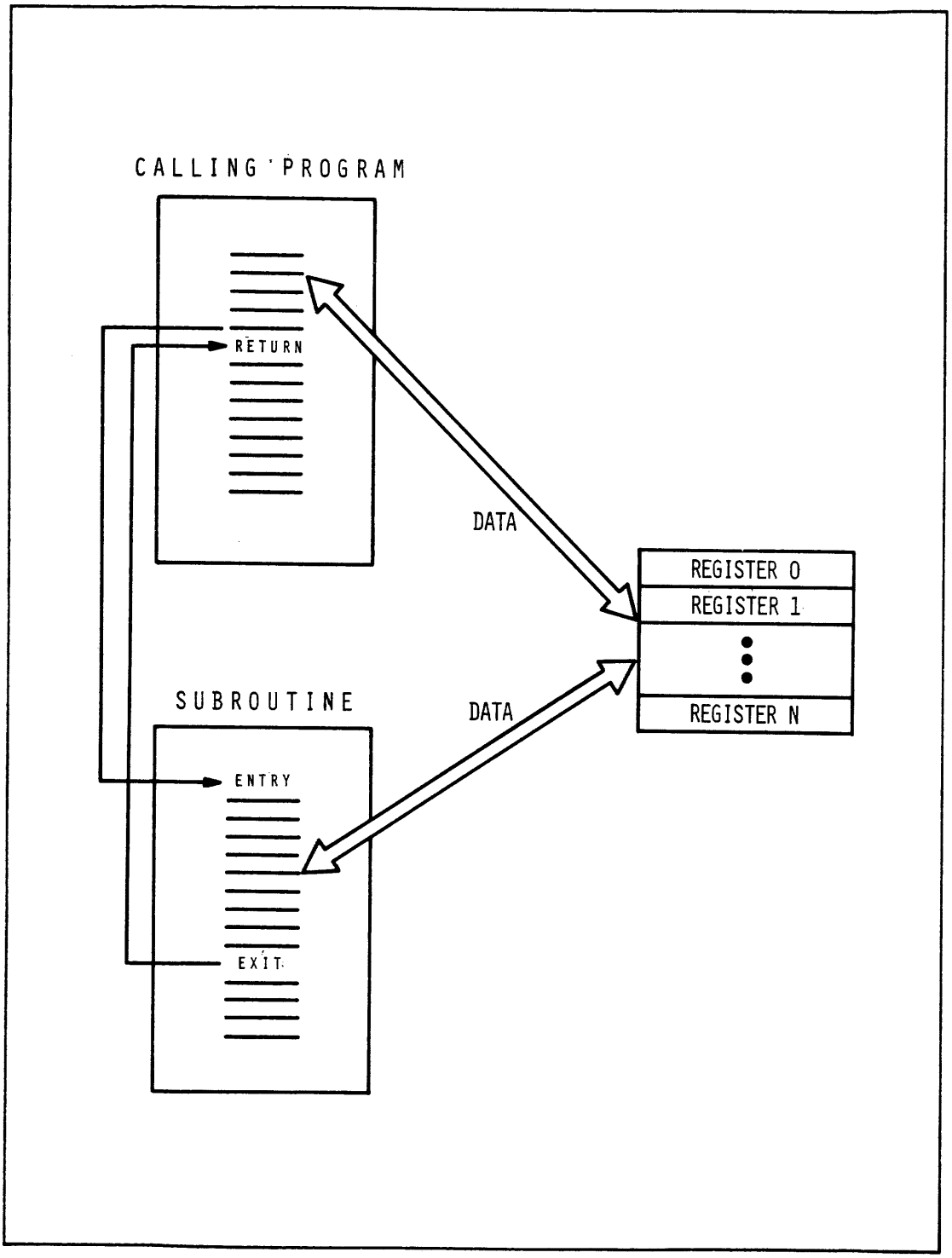


Figure 8-7. Passing Data in the Working Registers

Relation of Subroutines to Past Program Examples

In previous chapters, program examples have been stand-alone programs; that is, they were self-contained. The programs were not called by other programs, so, by definition, they were not subroutines.

It is more likely, however, that specific program segments are written to be a part of an overall application that has several other functions. There is the need then for interaction between the various program segments.

As an example, consider the traffic-light controller in the program example of Chapter 7. It is likely that this program may be simply a part of a larger traffic-light controller application. There may be other programs that perform such functions as counting the traffic in the lanes, a clock program to coordinate the timing of the traffic lights according to the time of day, a program to log the traffic activity by time of day and, perhaps, a diagnostic program that periodically checks the status of the equipment. In such a traffic-light controller, the program example in Chapter 7 might be a subroutine. This subroutine might be called from another program and have data passed to it which might, for example, define the number of cars in each lane. The subroutine could then modify the timing of each lane depending upon the traffic load.

Likewise, the cycle generator program example in Chapter 6 could be included in an application as a subroutine and called by other programs to produce various tones or provide duty-cycle control for various devices. In fact, the program example in this chapter uses the cycle generator program as a subroutine.

With the 9900 family of microprocessors, there is the option of allowing each subroutine to have its own set of working registers. Transferring control from one program to another program having its own set of registers is called "context switching."

8.4 CONTEXT SWITCH

One of the distinguishing characteristics of the 9900 family of microprocessors, is its memory-to-memory architecture. One of the advantages of this architecture is the flexibility to use a number of different working register sets located in memory. It is possible for each program to have its own set of working registers.

Specifically, this architecture allows the processor to respond to an interrupt request rapidly. This fast interrupt response is a result of the ability to assign a unique set of registers to an interrupt service routine. Upon an interrupt, control is transferred to the interrupt service routine with its own set of working registers by means of a context switch.

Definition

To explain context switching, it is first necessary to explain what is meant by a "context." Perhaps it would help to replace the word "context" with the word "environment."

A program has an environment or context. Consider Figure 8-8, which shows a program for the TMS 9980A in its environment. There are essentially four items that constitute a program's environment.

First, a given program occupies memory space. The defined area of memory that the program occupies is a part of that program's environment.

Second, with the 9900 series of microprocessors, the area of memory set aside for the working registers is also a part of the program's environment. This is generally one of the first things that must be defined in a program. Within the processor, the workspace pointer contains the address of the working registers; therefore, the content of the workspace pointer (WP) is a part of a program's environment or context.

Instructions are being executed one at a time in a program. The current instruction to be executed at a given point in time is a third part of the program's environment. The program counter (PC) is the processor's internal register that keeps track of the address of the current instruction. Therefore, the contents of the program counter at a particular point in time is a part of a program's environment.

Finally, as a program is being executed, the individual status bits in the status register are being set or reset to reflect the current status resulting from the execution of the instructions in the program. Therefore, the contents of the status register (ST) is the fourth part of the program's environment.

It can be said, then, that the four items, which constitute a program's environment are

- Its memory space
- The contents of the workspace pointer
- The contents of the program counter
- The contents of the status register.

Normally the first of these four items, the memory space of a program, is rigid. The program is defined to occupy an area in memory, and once the program is located (either in RAM with a loader or programmed in ROM) it is not subject to change. However, the last three items are dynamic; they can change as a result of the system's execution.

Context switching or changing from one program environment to another, means switching from one program to another in memory by changing, or allowing to change, the three internal registers

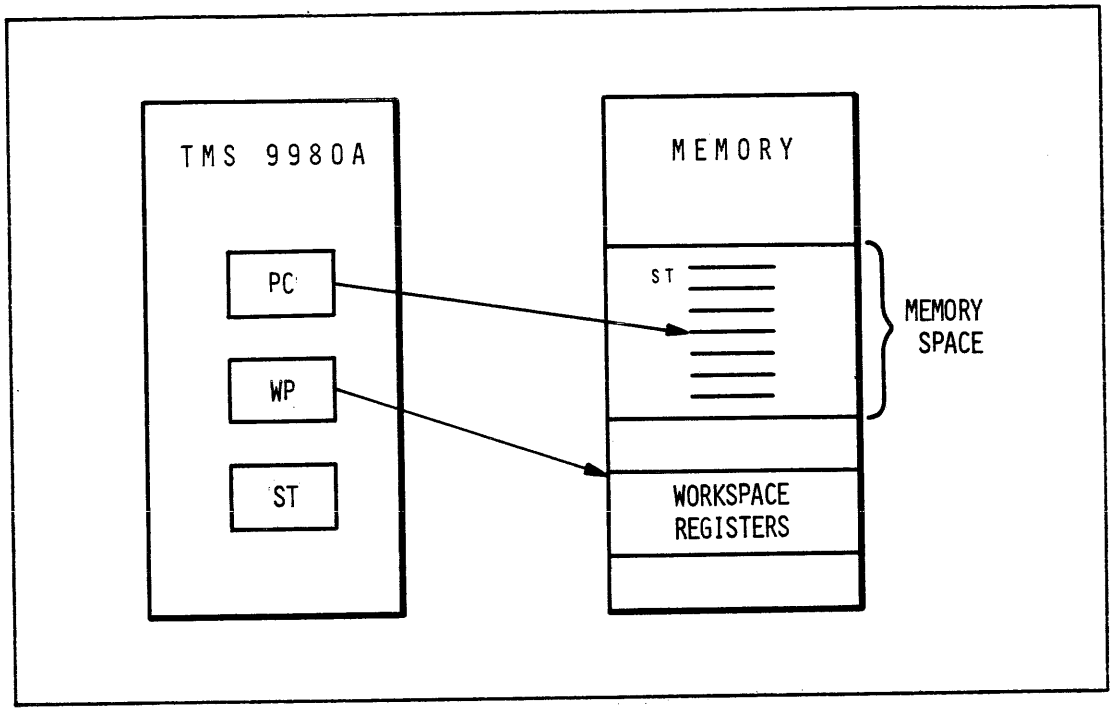


Figure 8-8. A Program In Its Environment

of the 9900 family processor: the workspace pointer, the program counter, and the status register. Also included within the concept of context switching is the act of saving the first program's environment so that it can be restored at some later point in time.

Usage

Context switching can be employed to allow each program module to have its own set of registers; a program does not have to share its registers with another program. This is especially important with an interrupt service program. The interrupt service program is not required to save the interrupted program's registers and therefore can have a faster interrupt response time.

Context Switch Initiators

Regardless of whether hardware or software initiates a context switch, the TMS 9980A processor must have the address of the new program to which it will switch and the address of the working registers for that program. With every context switch, a two-word vector is used to identify these two addresses. Also, with every context switch, the TMS 9980A processor saves the old program's environment in the new workspace area.

Hardware-Initiated Context Switch. To better understand the common elements in a context switch, consider a hardware-initiated context switch. A system's hardware may include external interrupts; that is, events outside the processor that request the immediate attention of the processor. (The concept of interrupts, vectored interrupts, and priority interrupts are discussed in earlier chapters.) The 9900 family of processors, employs prioritized, vectored interrupts. The processor is capable of receiving a number of interrupts. Each interrupt is assigned a specific code which serves to identify the interrupt as well as to assign it a relative priority in relation to other interrupts. Specifically, the TMS 9980A implements six levels of interrupts (RESET, LOAD, and user levels 1 through 4). RESET has the highest priority followed by LOAD and then user level 1 to level four (lowest priority).

Refer to Figure 8-9. There are three input pins on the TMS 9980A (IC0, IC1, and IC2) which identify the presence of an interrupt request and its code. The previous chapter discusses the details of how this level code is generated by the hardware. The codes for these various levels are shown in Figure 8-10. Notice that although there are eight possible values with these three lines ($2^3 = 8$), only six interrupt levels are defined. One value (111_2) is used to indicate the "no interrupt" state. The values 000_2 and 001_2 have redundant functions; they both define the RESET interrupt.

The TMS 9980A processor logic (microcode) uses these three input lines to recognize when an interrupt request is being made. If the code present on these input pins represents a level which is less than or equal to the value in the low order four bits of the status register, the interrupt request is granted and a context switch is made following the completion of the currently executing instruction and prior to fetching the next instruction.

The low-order four bits of the status register are used to prevent interrupts of insufficient priority. These bits are automatically affected when an interrupt occurs and can also be changed under program control.

The level represented by the three-bit code on the interrupt input lines is used by the processor to determine the location of the two-word vector for the interrupt context switch. Figure 8-11 shows the memory map for the TMS 9980A interrupt vectors. There is a pair of words (four bytes) associated with each vector. The first word of each pair contains the address of the working register set (that is, the value to be placed into the workspace pointer). The second word contains the address of the program to service the interrupt (that is, the value to be placed into the program counter). Each interrupt level (level zero or RESET, levels 1 through 4, and LOAD) has its own unique address for its two-word vector.

Consider the example of a level-two interrupt as depicted in Figure 8-12. Assume that program A is executing when the interrupt request occurs. Program A occupies an area of memory beginning

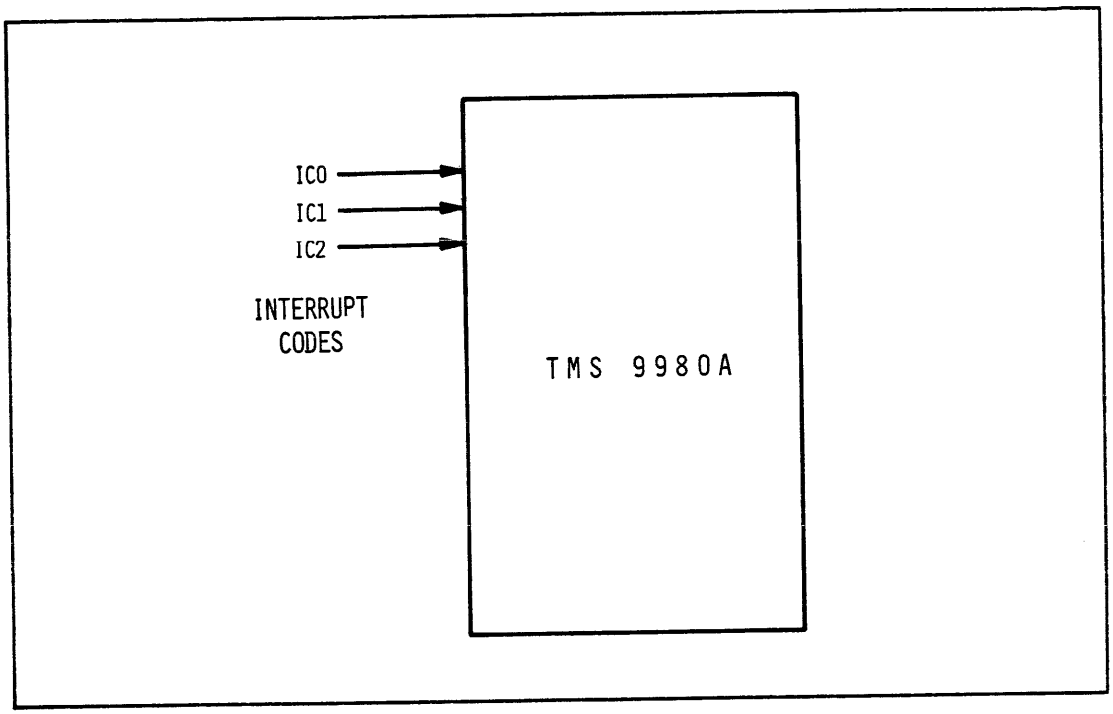


Figure 8-9. Interrupt Code Inputs to TMS 9980A

IC0	IC1	IC2	FUNCTION
0	0	0	RESET
0	0	1	RESET
0	1	0	LOAD INTERRUPT
0	1	1	LEVEL 1 INTERRUPT
1	0	0	LEVEL 2 INTERRUPT
1	0	1	LEVEL 3 INTERRUPT
1	1	0	LEVEL 4 INTERRUPT
1	1	1	NO INTERRUPT

Figure 8-10. Interrupt Level Codes

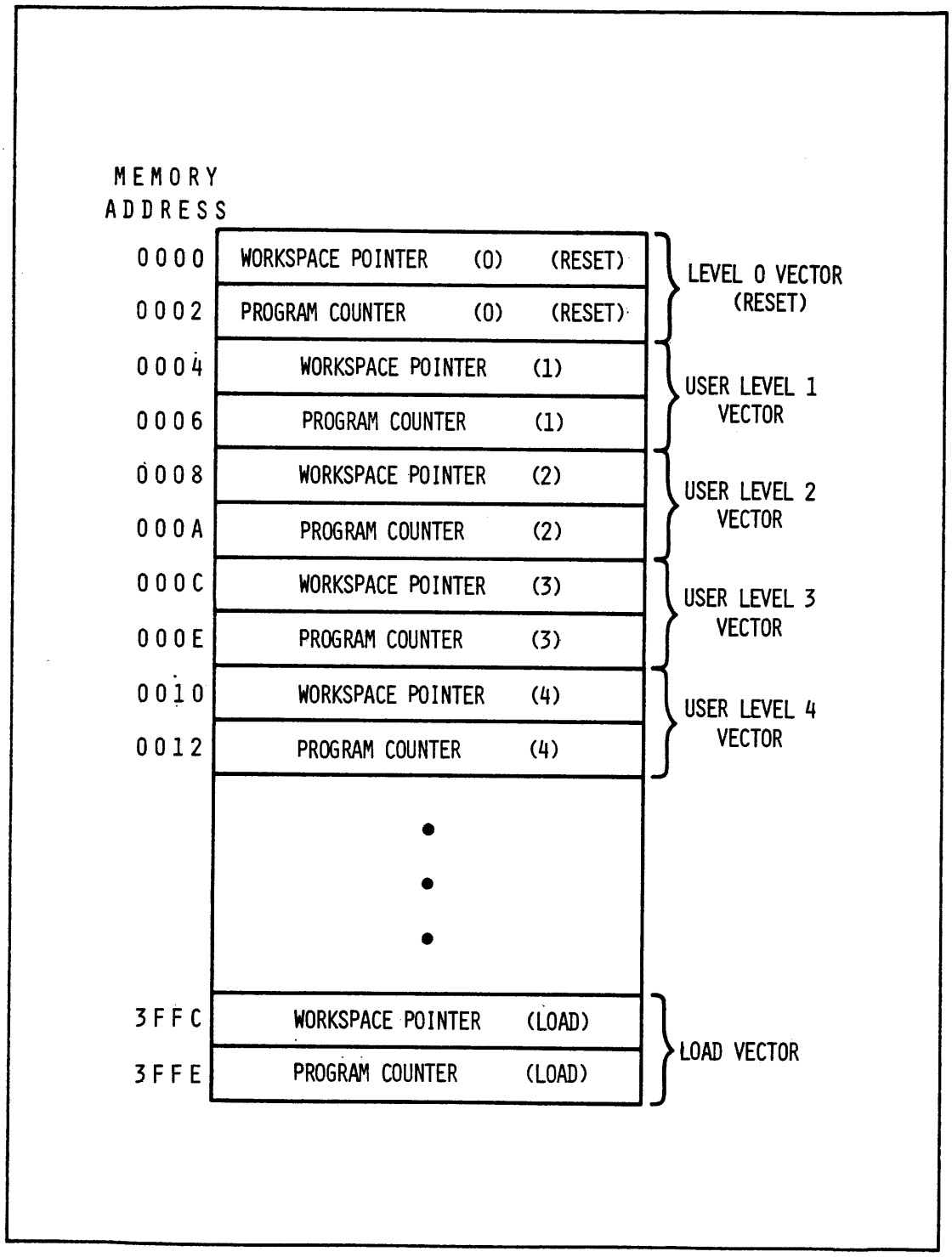


Figure 8-11. Memory Map for TMS 9980A Interrupt Vectors

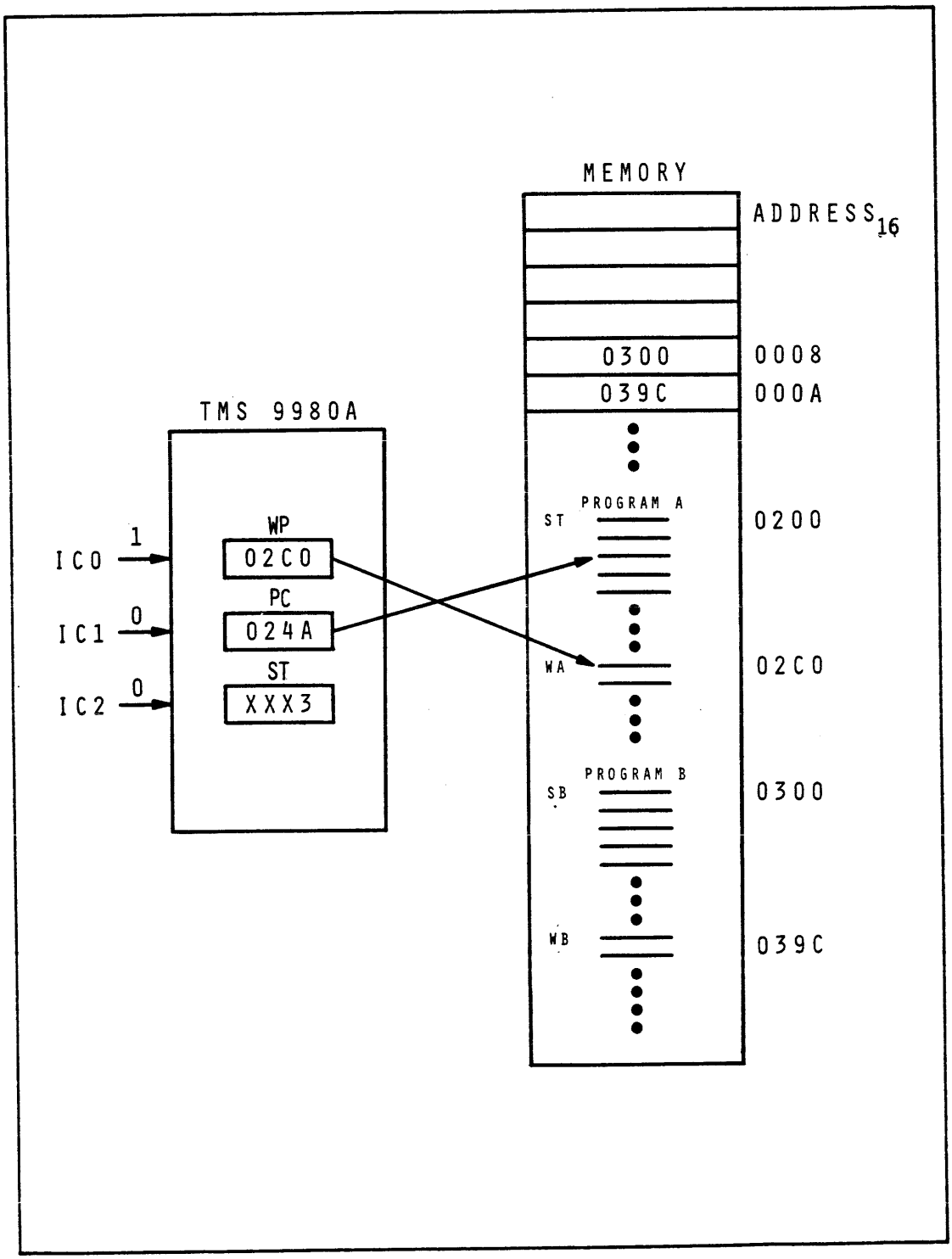


Figure 8-12. A Level-2 Interrupt Immediately Before a Context Switch

at address 0200_{16}. Program A's workspace register begins at address $02C0_{16}$. As the interrupt request is made, the program is executing an instruction, and the program counter contains $024A_{16}$ (the address of the next instruction). After completing the current instruction and just before the next instruction is fetched from memory, the processor examines the code on lines IC0 through IC2. If this code was 111_2, it would indicate no interrupt request was being made, and the processor would continue its normal program execution sequence.

Assume, however, that the code 100_2 appears on those lines. The processor interprets this code as a level-two interrupt request (see Figure 8-10). The processor compares this level to the value contained in bits 12 through 15 (the low-order four bits) of the status register. If the code on IC0 through IC2 represents a level equal to or less than the value in the status register, the processor performs a context switch. (Smaller numbers have priority over larger numbers.)

In this example, suppose the low-order four bits of the status register contained the value 0011_2 (or 3_{16}). A level-two interrupt (represented by the code 100_2 on IC0, IC1, and IC2) is less than 3; therefore, the interrupt request is granted.

The context switch is performed as depicted in Figure 8-13. The processor uses locations 0008_{16} and $000A_{16}$ as the two-word vector for the level-two interrupt. The processor temporarily saves the current value in the workspace pointer ($02C0_{16}$) internally and places in the workspace pointer the value found in location 0008_{16} ($039C_{16}$). Now with the workspace pointer set to the address of the new workspace area, the processor can save the interrupted program's environment (context). The processor places the previous contents of the workspace pointer ($02C0_{16}$) into working register 13 of the new (program B's) workspace area. The contents of the program counter ($024A_{16}$) are placed in working register 14 of the new workspace area and the contents of the status register are placed in working register 15 of program B's workspace area. At this point, the interrupted program's environment has been saved in the last three registers of the new workspace area.

The processor then completes the context switch by putting the contents of memory location $000A_{16}$ (the second word of the level-two vector) in the program counter (PC). (In this example, that value is 0300_{16}.) Finally, the value in the low-order four bits of the status register is set to one (0001_2), which is one less than the current interrupt level. The context switch is now complete, and the processor begins executing program B: a new program with a new set of working registers. Figure 8-13 shows the new environment after the context switch is made. The low-order four bits of the status register have been set to accept only interrupts of level 1, RESET or LOAD, and not levels 2, 3, or 4.

Notice that four bits are set in the status register to indicate the current level of interrupt. (The current level of interrupt

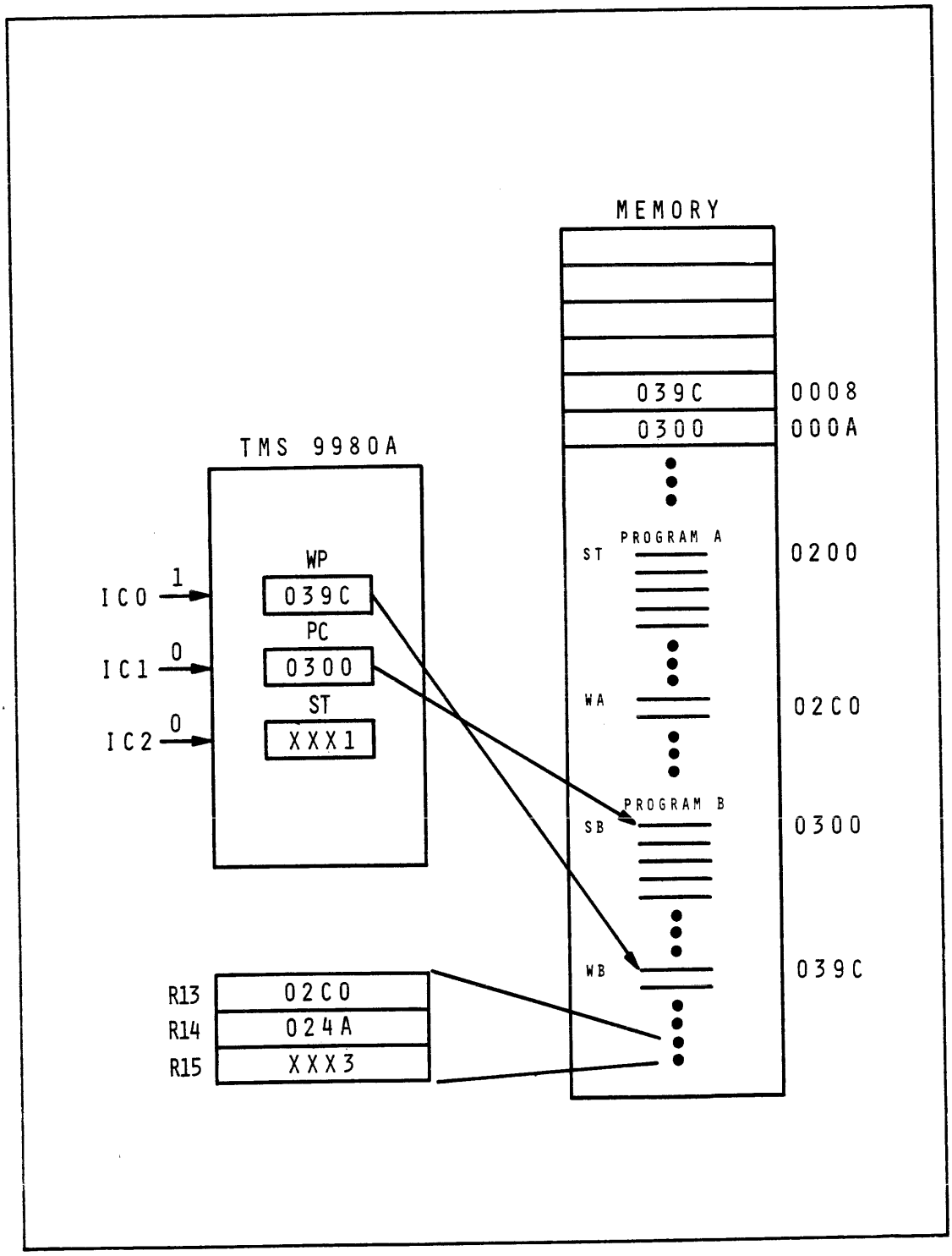


Figure 8-13. A Level-2 Interrupt Immediately After a Context Switch

is one greater than the value in status bits 12 through 15.) Only three bits, not four, are required for the TMS 9980A, because only six levels of interrupt are available; however, four bits are used to provide compatibility with the 9900 microprocessor, which accepts 16 levels of interrupt.

Note that there is still the possibility for an interrupt of higher priority to interrupt the new interrupt level-two program. In such a case, another context switch (or a "nested" context switch) would be made using the two-word vector of the new interrupt level.

It is possible through software to redefine this hardware priority structure by redefining the four-bit interrupt-mask value in bits 12 through 15 of the status register.

The question may arise: Although the old program environment has been saved, how is the context switch reversed? To reverse a context switch, the processor takes the value in working register 15 and places it in the status register; the value in working register 14 is placed in the program counter, and the value in working register 13 is placed in the workspace pointer. Thereby, the context switch is reversed. There is a specific instruction in the TMS 9980A's instruction set (RTWP) which causes this to happen automatically.

A hardware-initiated context switch has been discussed for the purpose of illustrating key elements in all context switches. The key elements include

- A two-word vector containing the address of the new workspace area and the address of the new program
- The old program's environment (the contents of the WP, PC, and ST) just prior to the context switch which is saved in working registers 13, 14, and 15, of the new workspace area.

Software-Initiated Context Switch. It is also possible to initiate a context switch from software as well as from hardware. There are two instructions in the TMS 9980A's instruction set that allow a context switch to be initiated from software:

- XOP
- BLWP.

The first of the two instructions is the extended operation (XOP) instruction. The XOP instruction can be thought of as a "software interrupt" because it functions somewhat like a hardware interrupt.

The XOP instruction has two operands associated with it. The second operand is a number in the range 0 through 15. This number (called the XOP code) identifies the location of the two-word vector for the XOP instruction. This number may be thought of as the "level" for the XOP, as a level is used to identify a hardware

interrupt. Each XOP code has a two-word vector at a specifically defined memory location.

Figure 8-14 shows the memory map of the two-word vectors for an XOP instruction located from memory address 0040_{16} (XOP 0 vector) through memory address $007F_{16}$. Whenever an XOP instruction is executed, the processor uses the code associated with the XOP instruction to identify the location of the vector for that XOP. The processor uses the first word of the vector as the address of the new workspace area (the value to be placed in the workspace pointer) and the second word as the address of the new program (the value to be placed in the program counter). The previous program's environment is saved in the last three registers of the new workspace area and in the same order as discussed with the hardware-initiated context switch.

There are some functional differences between an interrupt context switch and an XOP context switch, however. First, there is no priority associated with XOP codes; that is, the code associated with a given XOP is used simply for the purpose of identifying the location of the vector for that XOP. The execution of the XOP does not affect the low-order bits in the status register, and the processor makes no priority distinction of one XOP code from another.

Secondly, the XOP context switch causes one additional thing to happen that an interrupt context switch does not. With an XOP context switch, the processor causes a value defined by the first operand in the XOP instruction to be placed in working register 11 of the new program's workspace area. This allows a parameter to be passed to the called program from the calling program. For example, if a program called by an XOP sorted data numerically, the first operand could be the address of the data to be sorted.

Figure 8-15 is an example of a XOP context switch. An XOP 2 instruction at location 0228_{16} causes a context switch to a new program starting at location $032E_{16}$. As a result of the context switch, another set of working registers is assigned to the new program. The figure shows the contents of the WP, PC, and ST immediately before and after the context switch.

There is another instruction that can also invoke a context switch. It is the Branch and Load Workspace Pointer (BLWP) instruction. The BLWP context switch has the same general characteristics as the other context switching methods: it has a two-word vector associated with it which is used to identify the new workspace area and the new program; and it causes the workspace pointer, program counter, and status register to be saved in the last three registers of the new workspace area. The BLWP instruction has the unique characteristic, however, of allowing the two-word vector to be defined anywhere in memory. The operand associated with the instruction is the address of the two-word vector. There is no level or code or any other vector-identifying operand.

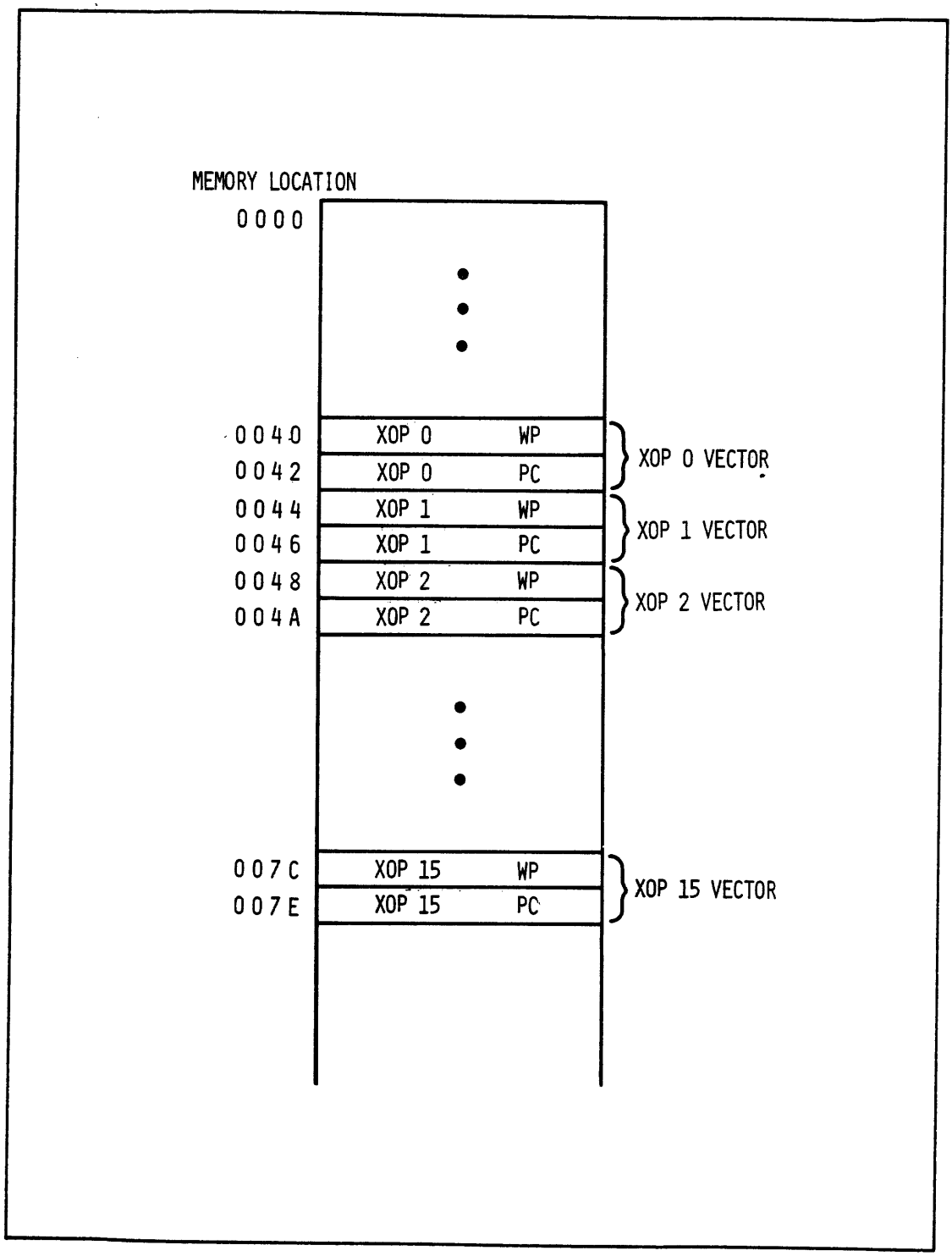


Figure 8-14. Memory Map for the XOP Vectors

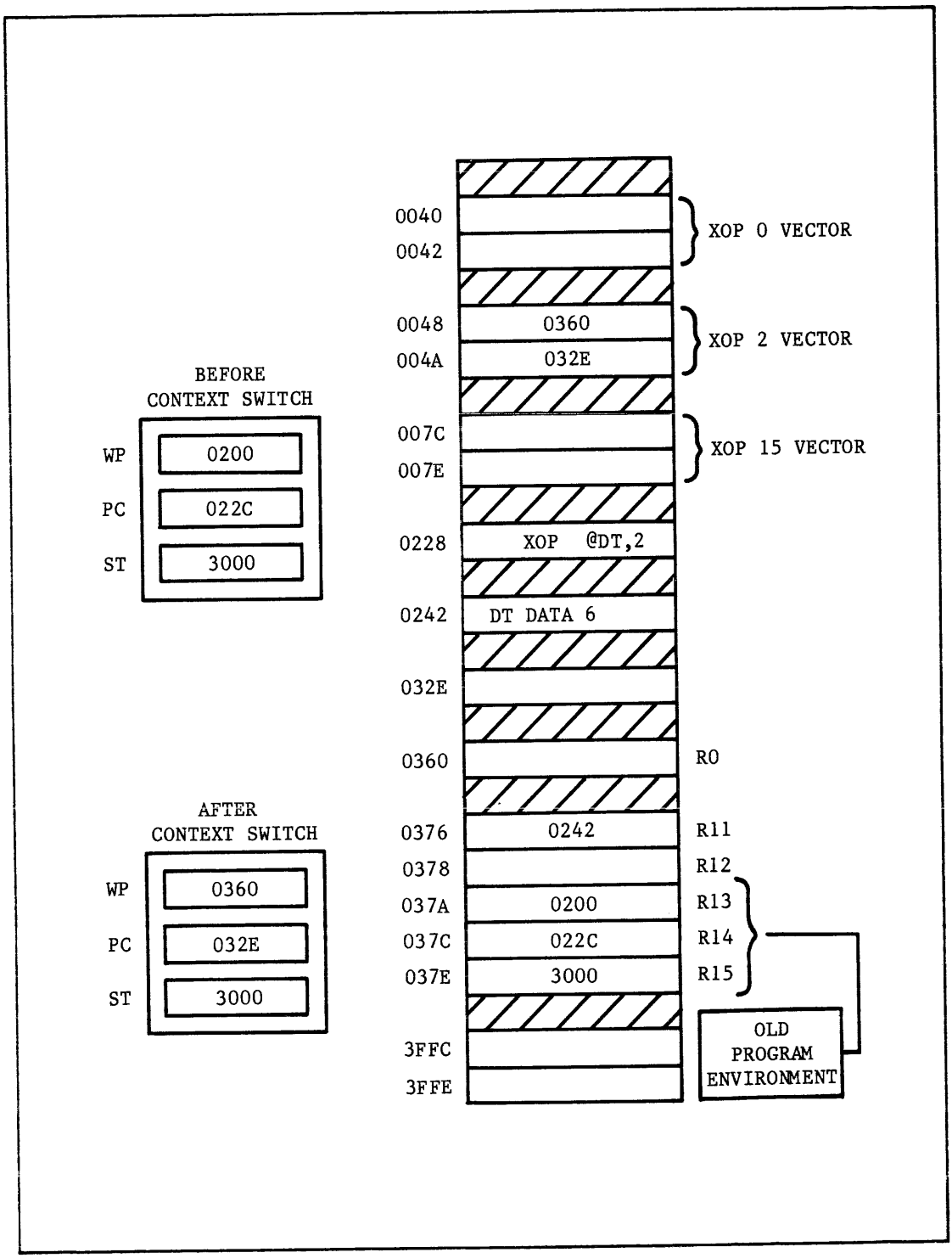


Figure 8-15. Example of an XOP 2 Context Switch

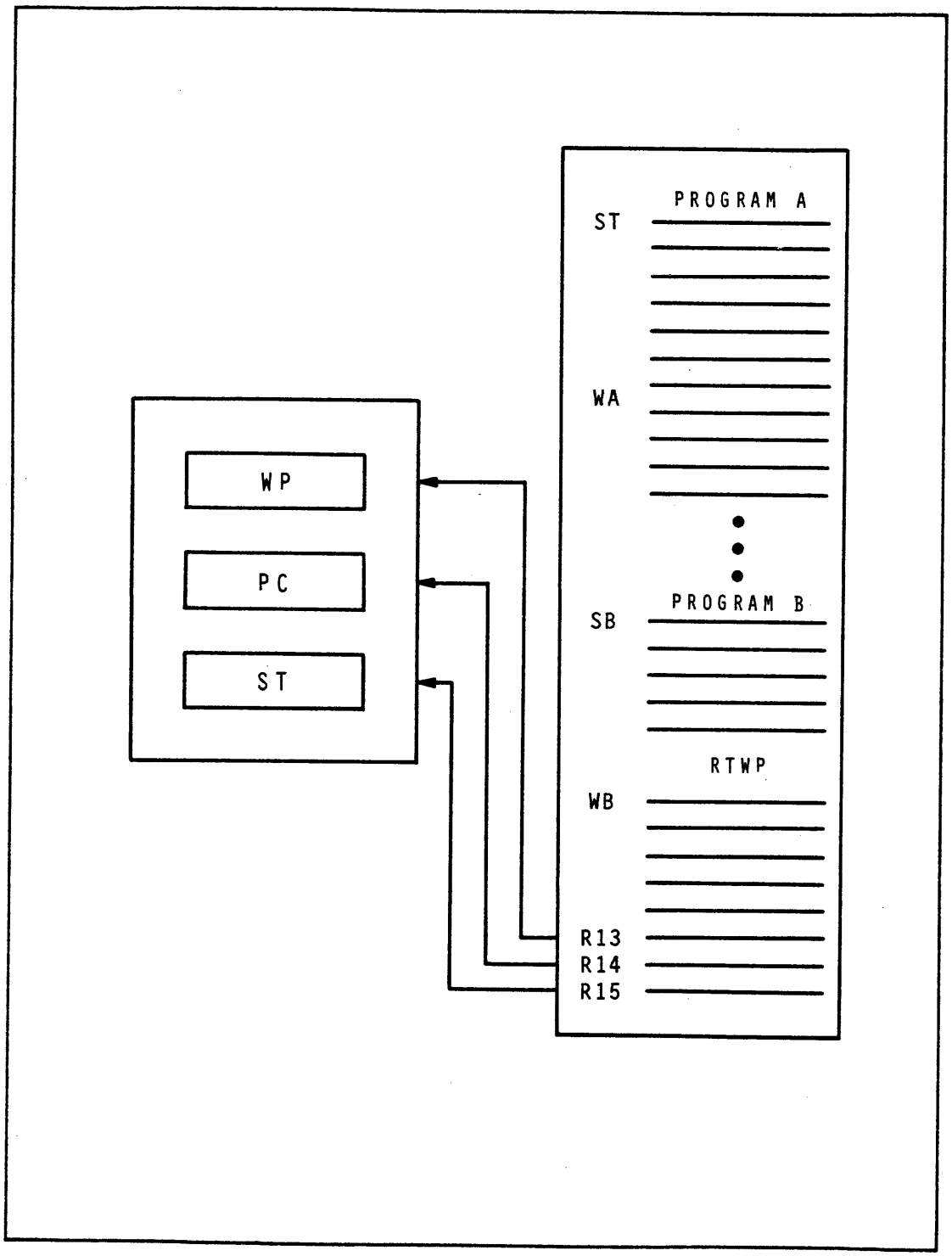


Figure 8-16. The RTWP Instruction Reverses a Context Switch

Both the XOP instruction and the BLWP instruction are described in detail in the next instruction subset.

Returning from a Context Switch

The return from a context switch is not complicated. The issuance of one instruction, the Return with Workspace Pointer (RTWP) instruction, causes a context switch to be reversed. This instruction can be used to return from a context switch originated by either hardware or software. The execution of the RTWP instruction causes the processor to place the contents of workspace register 15 in the status register, the contents of workspace register 14 in the program counter, and the contents of workspace register 13 in the workspace pointer. Figure 8-16 illustrates this. This action reverses a context switch and returns control to the original program at the same point and with the same environment that existed immediately prior to the context switch.

8.5 INSTRUCTION SUBSET 6

At this point, the last subset of TMS 9980A instructions are introduced. This will complete the description of the TMS 9980A's instruction set.

This subset consists of three groups. The first is composed of instructions which can be employed for making subroutine calls. One of these, the BL instruction, is used to make a subroutine call while maintaining the same set of registers. The other instructions in this group, the XOP and the BLWP, cause a context switch to be made. The final instruction in the group, RTWP, is used to return from a context switch.

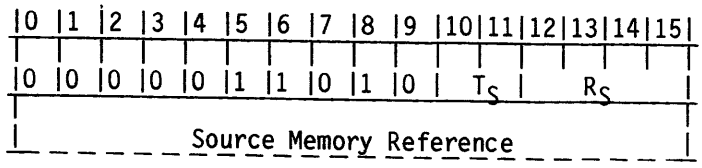
A second group is composed of two instructions that have the power of subroutines. Each of these instructions accomplishes alone what many other processors require a subroutine to do. These are the multiply and divide instructions. Indeed, an identifying feature of the 9900 microprocessor family is the single instruction multiply and single instruction divide.

The last group of instructions in the subset includes the execute (X), LIMI, STWP, and STST instructions. The LIMI instruction is used to change the interrupt mask in the status register, while the STWP and STST instructions permit the manipulation of the workspace pointer register and status register.

BRANCH AND LINK

BL

CODE: BL S

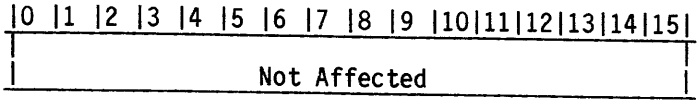


0680_{16}

RESULT: (PC) → (R11)
S → (PC)

Length: 1 or 2 words

STATUS REGISTER:



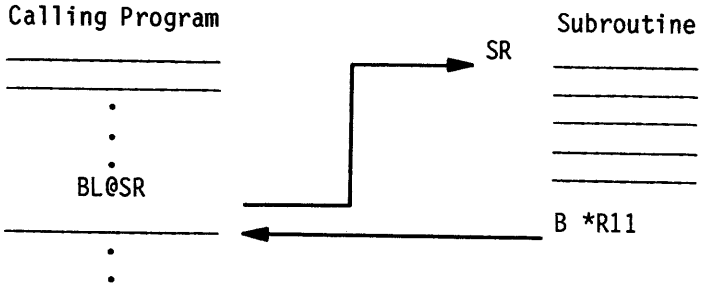
OPERATION:

Transfer control to the source operand and save the address following the BL instruction in working register 11.

NOTES:

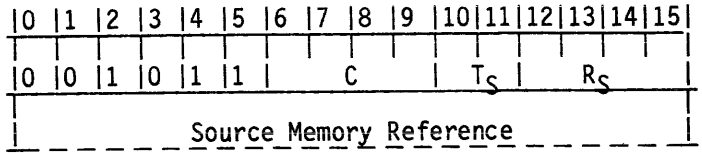
Used as a subroutine call. Causes a transfer of control to occur with return linkage. To return to the instruction following the BL subroutine call, the subroutine can execute a branch indirect through register 11 (B *R11).

Example:



EXTENDED OPERATION XOP

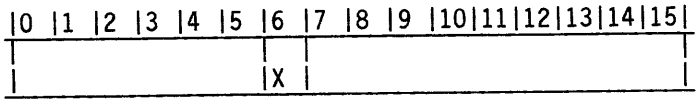
CODE: XOP S,C $2C00_{16}$



Length: 1 or 2 words

RESULT: $(40_{16} + 4 \times C) \rightarrow (WP)$
$(42_{16} + 4 \times C) \rightarrow (PC)$
(old WP) → (new R13)
(old PC) → (new R14)
(old ST) → (new R15)
S → (new R11)

STATUS REGISTER:



OPERATION:

Perform a context switch using a two-word vector located in an area of memory between addresses 0040_{16} and $007F_{16}$ (inclusive). The address of the two-word vector is determined by C. C is multiplied by four and added to 40_{16} to determine the address of the first word of the two-word vector pair. A context switch is made using the contents of the first word of the two-word vector as the address of a new set of working registers and the contents of the second word as the address of the program to which a transfer of control is made. The current contents of the WP, PC, and ST when the XOP instruction is executed are stored in registers 13, 14 and 15, respectively, of the new set of working registers. In addition, the effective address of the source operand is placed in register 11 of the new workspace registers.

NOTES:

Can be used to simulate a hardware interrupt.

Example: XOP @AB,3

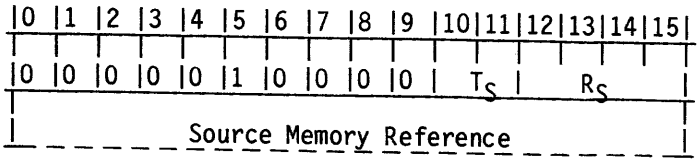
A context switch is made using the contents of location $004C_{16}$ as the address of the new working registers set and the contents of $004E_{16}$ as the address of the program to which control is transferred. The address of the current workspace, the address following the instruction and the contents of the status register are saved in working registers 13, 14, and 15 of the new working registers set. The address of AB is placed in working register 11 of the new workspace area.

Instruction Summary 8-2

BRANCH AND LOAD WORKSPACE POINTER

BLWP

CODE: BLWP S

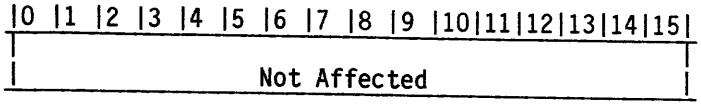


Length: 1 or 2 words

RESULT:

(S) → (WP)
(S + 2) → (PC)
(old WP) → (new R13)
(old PC) → (new R14)
(old ST) → (new R15)

STATUS REGISTER:



OPERATION:

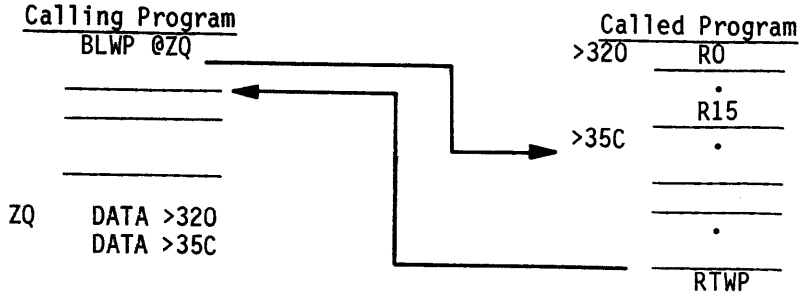
Perform a context switch using a two-word vector determined by the source operand. The source operand determines the address of the first word of the two-word vector. A context switch is made using the contents of the first word of the two-word vector as the address of a new set of working registers and the contents of the second word as the address of the program to which a transfer of control is made. The current contents of the WP, PC and ST when the BLWP instruction is executed are stored in registers 13, 14, and 15, respectively, of the new set of working registers.

NOTES:

Can be used to perform a context switch using a two-word vector defined anywhere in the memory address space.

Example: BLWP @ZQ

A context switch is made using the contents of ZQ as the address for a new set of working registers and the contents of location ZQ + 2 as the address of the program to which a transfer of control is made.



Instruction Summary 8-3

RETURN WITH WORKSPACE POINTER

RTWP

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: RTWP 0380_{16}	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0

Length: 1 word

RESULT: (R13) → (WP)
(R14) → (PC)
(R15) → (ST)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	L>	A>	EQ	C	OV	OP	X						I0	I1	I2	I3

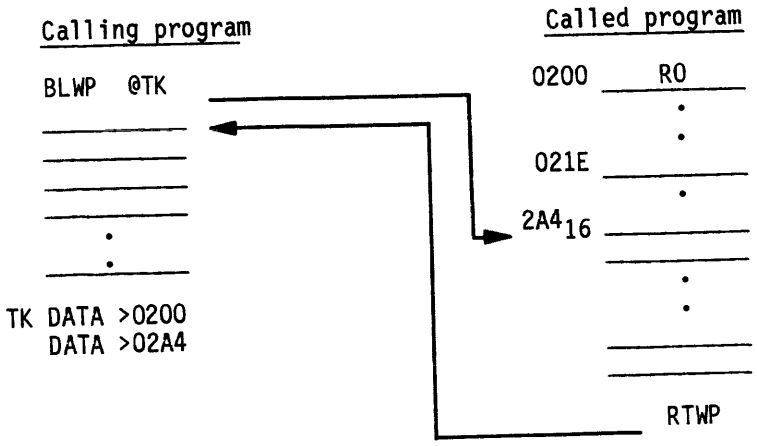
OPERATION:

Reverses a context switch. The contents of the workspace pointer, program counter, and status register are replaced by the contents of working registers 13, 14, and 15, respectively, of the current workspace area.

NOTES:

The RTWP is normally used to return to the program environment existing before a context switch caused by an interrupt or by execution of an XOP or BLWP instruction.

Example:

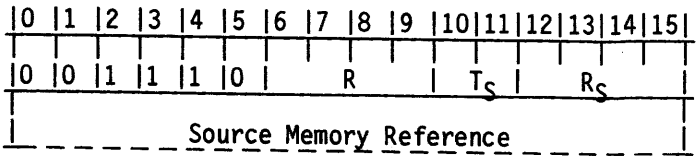


Instruction Summary 8-4

MULTIPLY

MPY

CODE: MPY S,R

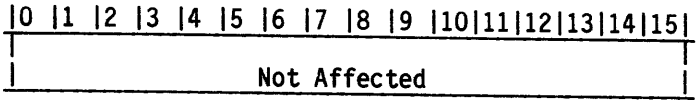


3800_{16}

Length: 1 or 2 words

RESULT: (S)*(R) ⟶ (R and R + 1)

STATUS REGISTER:



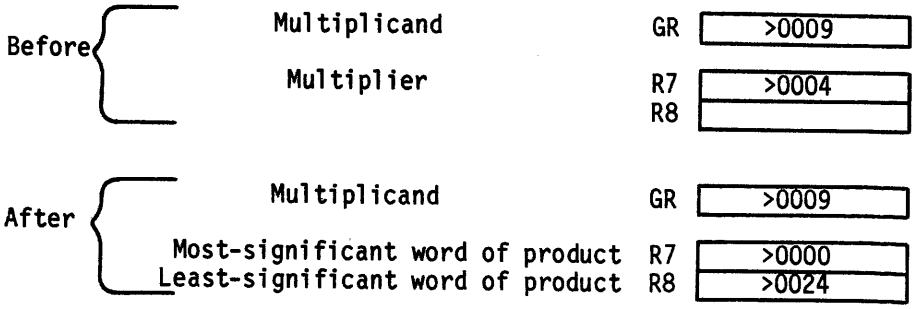
OPERATION:

Multiply the 16-bit contents of the source operand by the 16-bit contents of the destination register. The 32-bit result is placed in the destination register and the register following it. The most-significant 16 bits of the 32-bit result are placed in the destination register and the least-significant 16 bits are placed in the register following the destination register. If register 15 is the destination register, the least-significant 16 bits of the result are placed in the word following register 15. The 16-bit multiplier, 16-bit multiplicand and 32-bit product are treated as unsigned values.

NOTES:

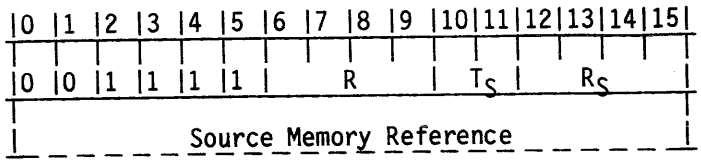
Used to perform a multiplication of absolute quantities.

Example: MPY @GR,R7
Causes the 16-bit quantity at location GR to be multiplied by the 16-bit quantity in register 7. The 32-bit result is placed in register pair 7 and 8 with register 7 containing the most-significant 16 bits.



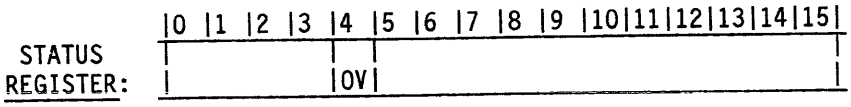
DIVIDE DIV

CODE: DIV S,R



Length: 1 or 2 words

RESULT: (R and R + 1)/(S) ⟶ (R) Quotient and (R + 1) Remainder



OPERATION:

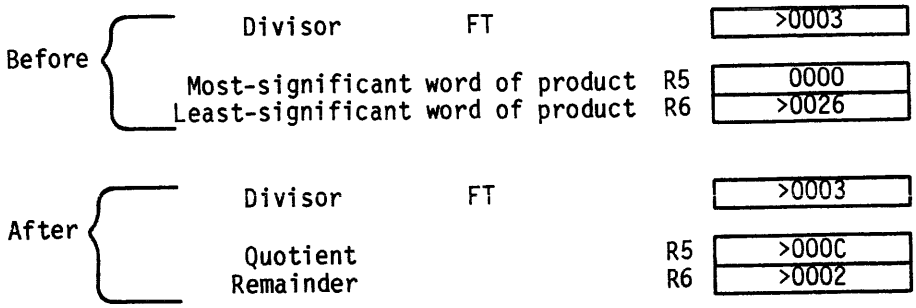
Divide the source operand into the destination operand. The source operand is a 16-bit absolute (unsigned) quantity. The destination operand is a 32-bit absolute (unsigned) quantity in a register pair. The register specified as the destination operand in the instruction is the first register of the pair. The second register is R + 1. The first register of the pair contains the most-significant 16 bits of the dividend and the second register contains the least-significant 16 bits. After execution of the instruction, a 16-bit quotient lies in the destination register (R) and a 16-bit remainder in R+1. Before the division takes place, the 16-bit source operand is compared to the 16-bit value in R. If the source operand is less than or equal to the value in R, the quotient would exceed 16 bits, in which case, the overflow status bit is set and the division is not performed.

NOTES:

Used to perform a division of absolute quantities.

Example: DIV @FT,R5

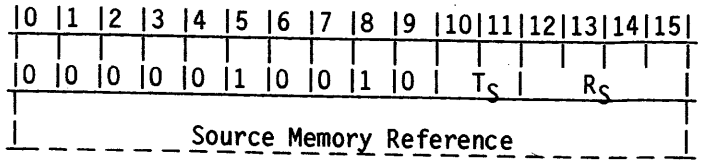
The instruction causes the 16 bit value in location FT to be divided into the 32-bit value in registers 5 and 6. Following the instruction, the 16-bit quotient is in register 5 and a 16-bit remainder in register 6.



Instruction Summary 8-6

EXECUTE X

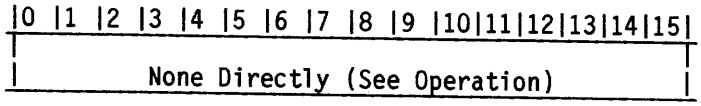
CODE: X S



Length: 1 or 2 words

RESULT: Execute the instruction indicated by the source operand

STATUS REGISTER:



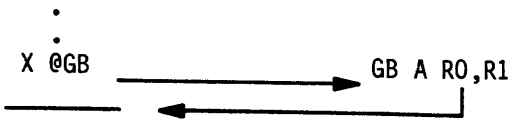
OPERATION:

Execute the instruction specified by the source operand. If the instruction specified is a two- or three-word instruction, the word or words following the execute instruction are used. The execute instruction itself does not affect the status register, but the specified instruction affects the status bits normally. If the executed instruction is a branch, the transfer of control is made. If the executed instruction is a jump and a transfer of control is made, the jump is made relative to the location of the execute instruction.

NOTES:

Example:

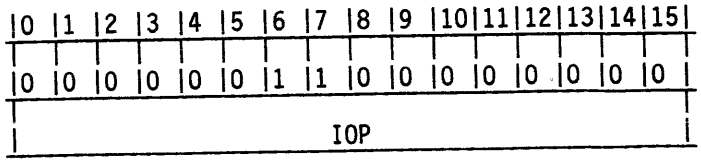
The instruction X @GB where location GB contains A R0, R1 causes the instruction at GB to be executed and control is returned to the instruction following the execute instruction.



LOAD INTERRUPT MASK IMMEDIATE

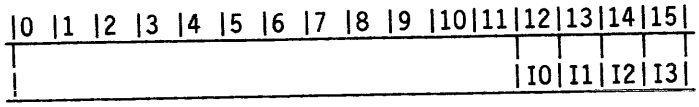
LIMI

CODE: LIMI IOP 0300_{16}



Length: 2 words

RESULT: $IOP_{12-15} \longrightarrow (ST)_{12-15}$



OPERATION:

The four least-significant bits of IOP are copied into the four least-significant bits of the status register. These four bits constitute the interrupt mask.

NOTES:

Used to set the interrupt mask which determines the levels of interrupt to be allowed.

Example:

A LIMI 2 instruction would allow only interrupt levels 0, 1, and 2 to be made, but not lower levels. The specified value in the instruction indicates the lowest priority level of interrupt to be allowed. Notice that a level 0 interrupt cannot be masked off.

STORE WORKSPACE POINTER STWP

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CODE: STWP R $02A0_{16}$	0	0	0	0	0	0	1	0	1	0	1	0			R	

Length: 1 word

RESULT: (WP) ⟶ (R)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
STATUS REGISTER:	Not Affected															

OPERATION:

The address in the workspace pointer is copied into the specified working register.

NOTES:

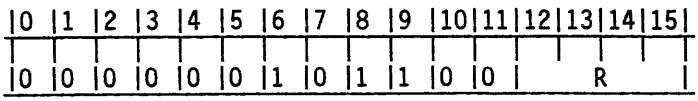
Used to store the contents of the workspace pointer.

Example:

```
STWP R8          STORE (WP) IN REG 8
```

STORE STATUS STST

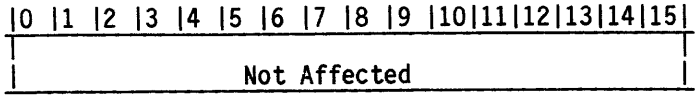
CODE: STST R $02C0_{16}$



Length: 1 word

RESULT: (ST) ⟶ (R)

STATUS REGISTER:



OPERATION:

The contents of the status register are copied into the specified working register.

NOTES:

Used to store the contents of the status register.

Example:

```
STST  R4          STORE (ST) IN REG 4
```

8.6 DEFINING AN XOP

With an understanding of the XOP instruction, a discussion is in order to see how the instruction can be employed to take advantage of functions resident in the UNIBUG monitor.

The UNIBUG firmware monitor controls the operator interface with the University Board through the terminal and display. The operator can "speak" to the monitor through the keyboard inputs, and the monitor can speak to the operator through the terminal display.

The UNIBUG monitor has the capability to read characters and display characters. These are functions which may be useful or required in user-written programs. The programs to do these functions could be written by the user, but they are already resident in the UNIBUG monitor, and it is simpler to call upon UNIBUG to perform these functions. First, however, it is necessary to understand the "protocol" or procedure for calling the monitor to peform these "utility" functions.

The user accesses the UNIBUG monitor via the XOP instruction. Recall that there are two operands with an XOP instruction. If an XOP is used to call UNIBUG, the second operand (the XOP number) defines to the monitor what function is to be performed, and the first operand defines the data to be included in the operation.

Figure 8-17 shows the relationship between a user program and the UNIBUG monitor.

The user program calls the UNIBUG monitor to perform a function by executing an XOP instruction. A context switch is made to the subroutine in the monitor where the function is performed. The subroutine performs the requested function and then executes an RTWP instruction to return control to the user program.

When calling the monitor, the second operand defines the function to be performed. Each XOP number is associated with a given function. The first operand identifies to the subroutine in the monitor the location of the data, where the data is to be placed, or other information. Any of the five general addressing modes can be employed for the first operand. It is often convenient, however, to use register addressing because most of the functions require only a limited amount of data which can be contained in a register.

8.7 USER ACCESSIBLE UTILITIES (UNIBUG Firmware)

Consider now the functions that are available within the UNIBUG monitor and the details for invoking these functions.

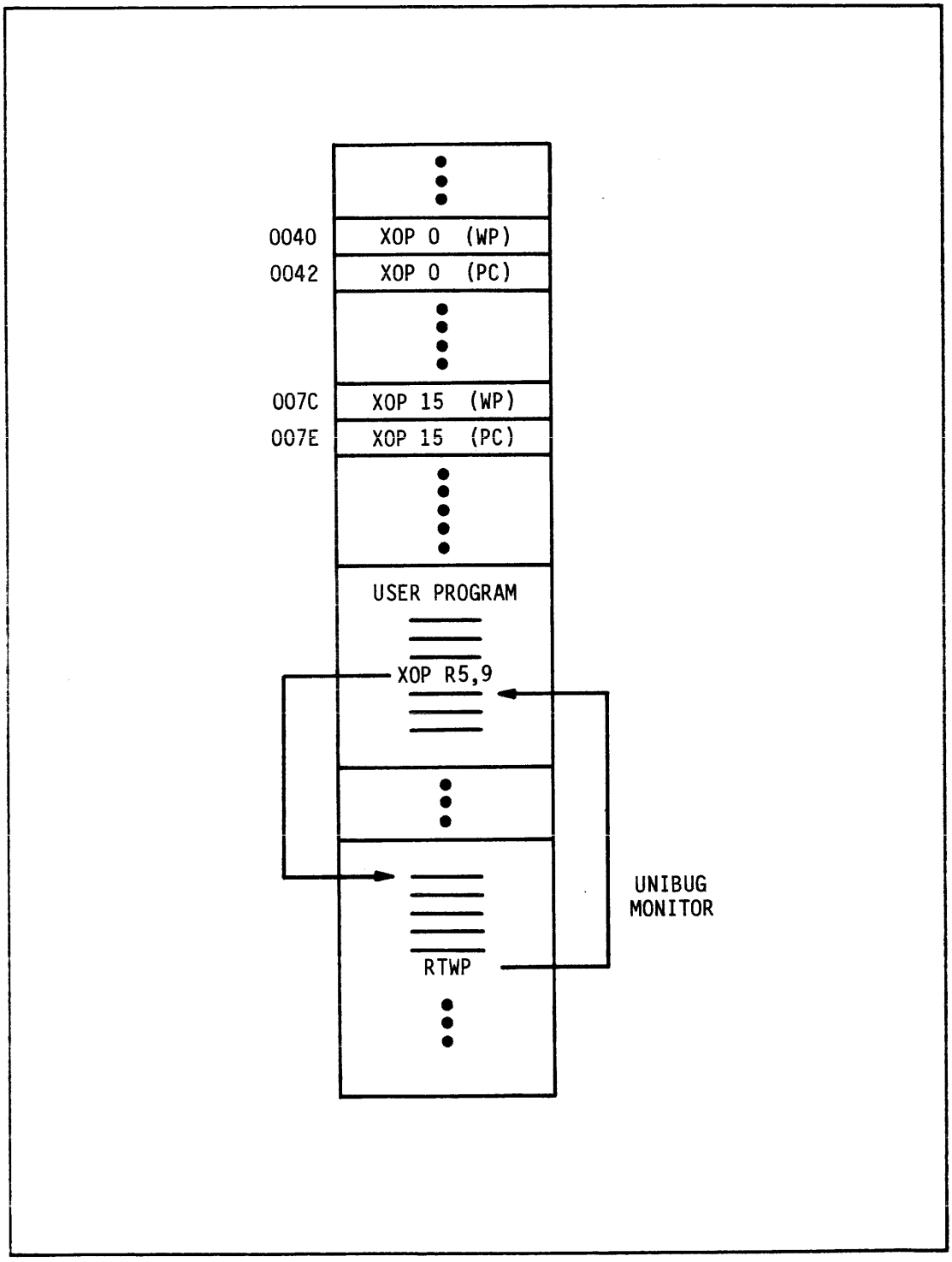


Figure 8-17. User Interface to the UNIBUG Monitor

Summary of Utility Functions

In summary, these are the functions available with the UNIBUG monitor:

XOP 8	Write one hexadecimal character to the terminal.
XOP 9	Read a hexadecimal word from the terminal.
XOP 10	Write four hexadecimal characters to the terminal.
XOP 11	Echo a character received from the keyboard to the display.
XOP 12	Write a character to the terminal.
XOP 13	Read a character from the terminal.
XOP 14	Write a message to the terminal.

The functions that are used to output to the on-board display (XOP 8, XOP 10, and XOP 14) must be followed by an input function in order to turn on the display.

The following section describes the details of interfacing to the UNIBUG monitor using these XOP's.

XOP 8 - Write One Hexadecimal Character to the Terminal

XOP 8 allows a user program to write a hexadecimal character to the display.

The format of this monitor call is:

```
XOP  S,8
```

S, or the source operand, can be a register, a symbolic address, or can be referenced using any one of the five general addressing modes. The hexadecimal value to be displayed is contained in the least-significant four bits of the source operand. Notice that the monitor converts this four-bit hexadecimal value into an ASCII character representation before displaying it.

The general format is the same for all the monitor calls. The source operand points to the data, and the second operand (XOP number) defines the function to be performed.

As an example, suppose the operator wants a hexadecimal F to show on the terminal's display. This can be accomplished with the following XOP.

```
      XOP   @FL,8     DISPLAY "F"
      XOP   R0,13     ACTIVATE DISPLAY AND RECEIVE CHAR
              .
              .
              .
FL    DATA  >000F
```

XOP 9 - Read a Hexadecimal Word from the Terminal

XOP 9 allows up to four hexadecimal digits to be read from the keyboard followed by a valid terminating character. A valid terminating character is a carriage return, space, or comma. Two words follow the XOP 9. The first work defines the address where control is transferred if no characters are entered (other than a valid terminaing character). The second word defines where control is transferred if something other than a hex digit or a valid terminating character is entered. Normally, control is returned by UNIBUG to the instruction following these two words.

The following is an example of the structure of a monitor call to read a hexadecimal word.

```
      XOP   @HW,9           occupies locations 0320(16) and 0322(16)
      DATA  NL              occupies location 0324(16)
      DATA  ER              occupies location 0326(16)
      <next instruction>    occupies location 0328(16)
        .
        .
        .
HW    DATA  0
      DATA  0
```

When the XOP instruction is executed, the last four hex digits entered on the keyboard prior to the terminating character are placed in the word labeled HW.

The terminating character is placed in the left byte of the next word. Program control is returned by UNIBUG to the address of the instruction following the XOP call (location 0328_{16}).

Recall from the explanation of the UNIBUG monitor or from direct experience that when entering a hexadecimal word (four digit) value on the keyboard, any number of hexadecimal digits can be entered, but the monitor retains only the last four digits prior to the terminating character. If less than four digits are entered, leading zeroes are returned.

If the XOP call in the example had been made and the operator simply entered a valid terminating character but no hex digits, UNIBUG would have returned control to the address contained in the word immediately following the XOP call. In this example, control would have gone to location NL. If the XOP call had been made and something other than a valid hex digit or terminating

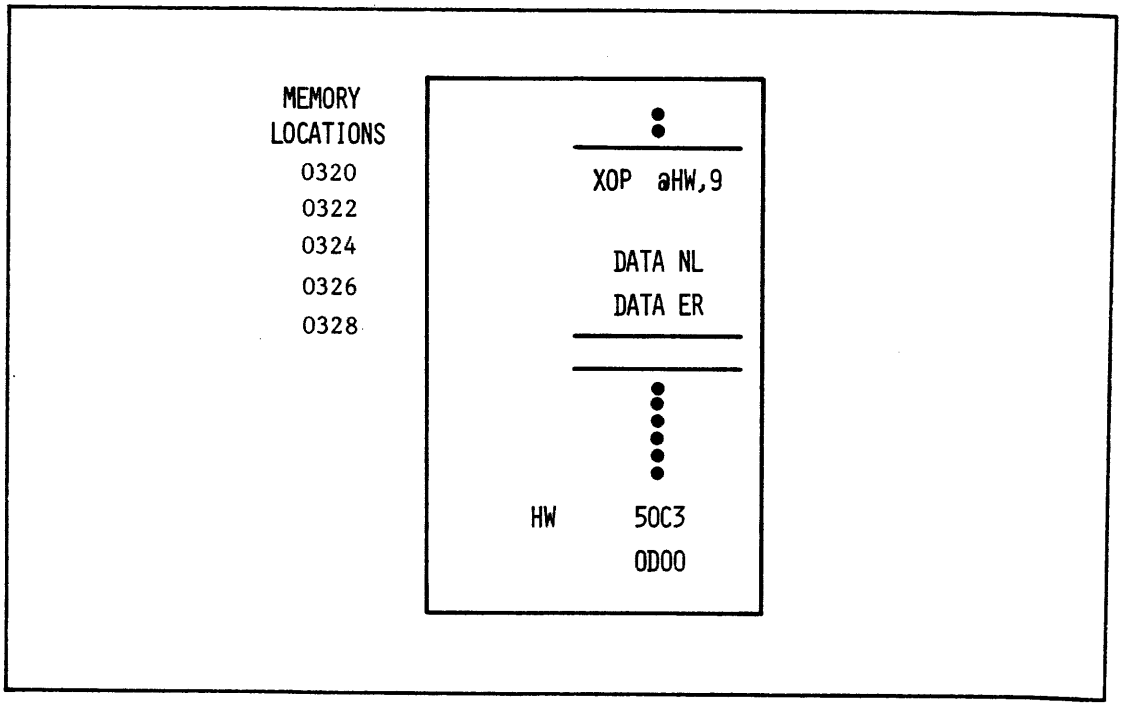


Figure 8-18. Results of an XOP 9 Monitor Call

character had been entered, UNIBUG would have returned control to the address contained in the second word following the XOP instruction. In this example, control would have gone to location ER.

Figure 8-18 shows the results of an XOP 9 monitor call where the operator entered 5, 0, C, and 3 followed by a carriage return. Notice that the monitor converts the ASCII representation of the hex digits entered on the keyboard into a four-bit binary value.

XOP 10 - Write Four Hexadecimal Characters to the Terminal

XOP 10 allows four hexadecimal digits to be written to the display. For example, assume register 9 contains the value $1AD3_{16}$. The following instruction

```
XOP R9,10        DISPLAY THE HEXADECIMAL DIGIT IN R9
XOP R0,13        ACTIVATE DISPLAY AND RECEIVE CHAR
    .
    .
```

would cause the characters 1, A, D, and 3 to appear on the display.

Notice the monitor subroutines must first convert each four-bit hex digit into ASCII character representation before displaying it.

XOP 11 - Echo a Character Received from the Keyboard to the Display

XOP 11 causes an ASCII character read from the keyboard to be automatically "echoed" or displayed.

For example, the XOP instruction in the program shown below causes characters to be read from a keyboard and automatically displayed. The program saves the characters in a buffer. As each character is entered, it is examined to see if it is a period. If so, program control is transferred to location FN.

```
    LI    R7,BF       PUT BUFFER ADRS IN R7
RD  XOP   *R7,11      READ AND ECHO A CHAR
    CB    *R7,@PE     IS IT A PERIOD?
    JEQ   FN          YES
    INCT  R7          NO - UPDATE BUFFER ADRS
    JMP   RD          GO READ ANOTHER CHAR
```

PE is a byte address containing the value $2E_{16}$ (ASCII code for period). The program assumes the parity of the received character to be reset (0). (If the parity of the received character was unknown, it would be necessary to precede the CB instruction with an instruction to force the parity bit to ZERO. For example, a SZCB @PM,*R7 could be used, where byte location PM contains 80_{16}.)

XOP 12 - Write a Character to the Terminal

XOP 12 causes a character to be written to the display (or terminal). Suppose the left byte of register 0 contains the value $4D_{16}$.

The sequence of instructions

```
XOP R0,12          DISPLAY THE CHAR IN R0
XOP R1,13          ACTIVATE DISPLAY AND RECEIVE CHAR
```

would cause the ASCII character M to be displayed.

XOP 13 - Read a Character from the Terminal

Conversely, XOP 13 causes a character to be read from the cassette. For example, an XOP @IN,13 instruction would cause a character received from the keyboard to be placed in byte address IN.

XOP 14 - Write a Message to the Terminal

XOP level 14 causes a string of characters (a message) to be written from a program to the display. The source operand in the XOP instruction points to the byte address of the first character. The monitor outputs the first character followed by the character at the subsequent byte address followed by the next character and so forth until it encounters a byte containing the value 0. A zero byte terminates the message output and causes a return to be made to the instruction following the XOP call.

For example, the following program would cause the message ACTIVATE to appear on the display.

```
        XOP   @MS,14          DISPLAY PROMPTING MESSAGE
        XOP   R0,13           AWAIT ACKNOWLEDGMENT
          .
          .
          .
MS TEXT 'ACTIVATE'
```

The following program example illustrates the use of several of these UNIBUG functions.

8.8 PROGRAM EXAMPLE: MORSE CODE TRANSLATOR

Program Description

This program example illustrates how programs can be structured in modular pieces, including subroutines. The main program (described

here) accepts an input character from a keyboard, translates the character into Morse code, and "transmits" the Morse code on the TM 990/189 speaker.

The cycle generator program (the program example in Chapter 6) is used to drive the speaker. It is modified to make it a subroutine.

The main program uses a UNIBUG monitor call to read the character input on the keyboard by the operator. The character is checked to ensure that it is alphabetic. Should the character be nonalphabetic, the message INVALID will be displayed on the terminal. Otherwise, the character will be translated into Morse code using a "table look-up" technique.

The alphabetic characters A through Z have sequential values from 41_{16} through $5A_{16}$. This arrangement allows a simple check to be made to determine that an input character is alphabetic by ensuring that the received value is in the range 41_{16} through $5A_{16}$.

If the character is alphabetic, its ASCII code value can be converted into a relative value (from A) easily by subtracting 41_{16}. The resulting relative value may be used as an "index" into a table of Morse code characters.

The look-up table is structured so that each word in the table corresponds with an ASCII alphabetic character. The first word in the table is the Morse code for an A. The second word is the Morse code for a B, and, finally, the last word in the table is the Morse code for a Z.

The relative value of each ASCII alphabetic character is an index into the table. For example, an ASCII character A has the value 41_{16}. If 41_{16} is subtracted from the value, the result is an index of zero, which indicates the first entry in the conversion table. Likewise, by subtracting 41_{16} from 42_{16} (the ASCII value for B), an index of one is produced which points to the second entry in the table, and so on.

Each entry in the table is one word long. (Therefore, the index value produced above must be doubled to derive a relative address into the look-up table.) The low-order (left) byte in the word contains the Morse code for a character. Each character in Morse code is composed of long and short "elements." A short element is a dot. A long element is a dash. In a byte, a dot is represented by a binary ZERO digit, and a dash is represented by a binary ONE digit. The first element of the Morse character is the least-significant bit. The second element is the next least-significant bit and so forth. This structure allows an easy identification to be made of each element by performing a one-bit right shift and checking the resulting carry state to determine if it is a dot or dash.

The value in the left byte of each word of the look-up table indicates the number of Morse code elements in the character. There is a maximum of four elements for the alphabetic characters, so that all the character translation information could have been contained in one byte. However, this type of table structure eases the programming task somewhat.

If an operator inputs a carriage return, this indicates to the program that the operator is finished, and a return is made to the UNIBUG monitor.

Program Design

Figure 8-19 is a flowchart of the program. Figure 8-20 is a listing of the program. Refer to the listing for the following discussion.

The EQU statement labeled MN identifies to this program the address of the UNIBUG entry point. Since this program will utilize the cycle generator program from Chapter 6 as a subroutine, the program must also have identified to it the address of the cycle generator program. This is accomplished with the EQU statement labeled WC. There is a second entry point into the cycle generator, D1, that also needs to be identified. This is done with the EQU statement labeled D1. (Refer to the Program Example in Chapter 6.)

The program is originated at location 0280_{16} (with an AORG statement). By beginning at this location, the program can reside in memory along with the cycle generator subroutine without memory-location conflict since the cycle generator program occupies memory locations 200_{16} to $24A_{16}$.

The workspace pointer is set at location $2A0_{16}$ and the CRU software base address is set in register 12 at location $20C_{16}$.

These same two steps are taken in the cycle generator program, but because this program is the main program and gets control first when execution begins, these two functions are done here in the main program rather than in the cycle generator subroutine.

At locations $2A8_{16}$ and $2AC_{16}$, registers 0 and 1 are initialized to a value of ten. These values are passed to the cycle generator subroutine to cause a 1-kHz tone to be produced when the Morse characters are transmitted.

At locations $2B0_{16}$ (labeled AA) and $2B2_{16}$, register 6 is cleared and XOP level 11 is used to make a monitor subroutine call to read and echo a character from the keyboard.

When a character is entered, UNIBUG returns control to the instruction at location $2B4_{16}$ with the received character in the left byte of register 6.

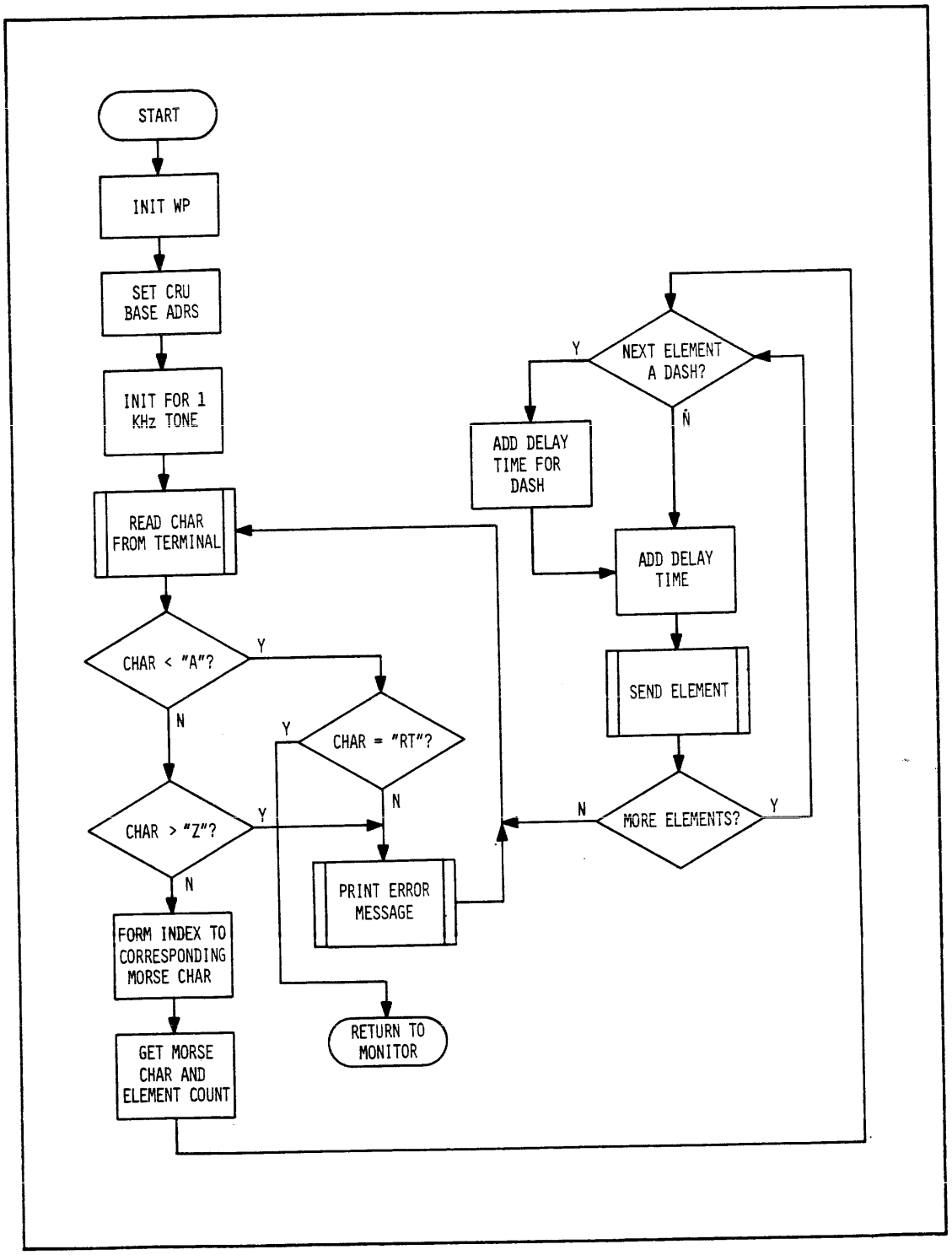


Figure 8-19. Morse Code Program Flowchart

```
0001                    *  INPUTS:
0002                    *    CHARACTERS RECEIVED FROM KEYBOARD
0003                    *  OUTPUTS:
0004                    *    TIMING VALUES FOR WAVE CYCLE SUBROUTINE
0005                    *  REGS. USED
0006                    *    R0 - CYCLE "ON" TIME FOR WAVE CYCLE SUBROUTINE
0007                    *    R1 - CYCLE "OFF" TIME FOR WAVE CYCLE SUBROUTINE
0008                    *    R2 - LENGTH OF TONE COUNTER
0009                    *    R4 - DELAY COUNT FOLLOWING EACH ELEMENT
0010                    *    R6 - ELEMENT COUNT FOR MORSE CODE CHARACTER
0011                    *    R7 - HOLDS MORSE CODE CHARACTER
0012                    *    R12 - BASE CRU ADRS FOR OUTPUT DEVICE
0013                    *  ADDRESS EQUATES
0014          31C2      MN   EQU  >3000      MONITOR ENTRY ADRS
0015          0228      WC   EQU  >0228      ADRS OF "WC" IN WAVE CYCLE PGM
0016          0236      D1   EQU  >0236      ADRS OF "D1" IN WAVE CYCLE PGM
0017                    *
0018 0280               WS   BSS  32         WORKSPACE AREA
0020 02A0 02E0          ST   LWPI WS         INIT WP
     02A2 0280
0021 02A4 020C               LI   R12,>43C   SET ADRS OF SPEAKER IN R12
     02A6 043C
0022 02A8 0200               LI   R0,10      SET UP R0 AND R1 FOR
     02AA 000A
0023 02AC 0201               LI   R1,10      1 KHZ TONE
     02AE 000A
0024 02B0 04C6  AA           CLR  R6         CLEAR BOTH BYTES OF R6
0025 02B2 2EC6               XOP  R6,11      READ A CHAR FROM TERMINAL
0026 02B4 9806               SZCB R6,@PM     FORCE PARITY BIT TO ZERO
     02B6 5806
0027 02B8 9806               CB   R6,@A      IS CHAR LESS THAN AN "A"?
     02BA 0308
0028 02BC 111D               JLT  CC         YES-NOT ALPHABETIC
0029 02BE 9806               CB   R6,@Z      IS IT GREATER THAN A "Z"?
     02C0 030A
0030 02C2 151D               JGT  ER         YES-NOT A VALID CHAR
0031 02C4 06C6               SWPB R6         FORM INDEX IN R6
0032 02C6 0226               AI   R6,>FFBF   (>FFBF = ->41)
     02C8 FFBF
0033 02CA 0A16               SLA  R6,1
0034 02CC C1E6               MOV  @MT(R6),R7  GET MORSE CODE IN RIGHT BYTE
     02CE 031A
0035 02D0 C187               MOV  R7,R6      SET UP ELEMENT COUNT
0036 02D2 0986               SRL  R6,8       IN R6
0037 02D4 04C2  BB           CLR  R2         INIT CYCLE COUNT TO ZERO
0038 02D6 0917               SRL  R7,1       SHIFT NEXT ELEMENT TO CARRY
0039 02D8 1702               JNC  DT         SHORT OR LONG
0040 02DA 0222               AI   R2,>29B    LONG
     02DC 029B
0041 02DE 0222  DT           AI   R2,>14D    SHORT
     02E0 014D
0042 02E2 06A0               BL   @WC        GO TO WAVE CYCLE PGM
     02E4 0228
0043 02E6 0204               LI   R4,>1A0A   >1A0A = 6666
     02E8 1A0A
0044 02EA 0202               LI   R2,1       ONLY ONE TIME THRU LOOP
     02EC 0001
0045 02EE 06A0               BL   @D1        GENERATE DELAY BETWEEN ELEMENTS
     02F0 0236
```

Figure 8-20. Listing of Morse Code Program

```
0046 02F2 0606        DEC  R6          ELEMENTS REMAINING?
0047 02F4 16EF        JNE  BB          YES
0048 02F6 10DC        JMP  AA          NO-GO INPUT ANOTHER CHAR
0049 02F8 9806  CC    CB   R6,@CR      IS CHAR A CARRIAGE RETURN?
     02FA 030C
0050 02FC 1303        JEQ  EX          YES-RETURN TO MONITOR
0051 02FE 2FA0  ER    XOP  @MS,14      OUTPUT AN ERROR MSG
     0300 030E
0052 0302 10D6        JMP  AA          AND GO READ ANOTHER CHAR
0053 0304 0460  EX    B    @MN         RETURN TO MONITOR
     0306 3000
0054                  *
0055                  *  DATA CONSTANTS
0056                  *
0057 0308 4100  A     DATA >4100       ASCII CODE FOR "A"
0058 030A 5A00  Z     DATA >5A00       ASCII CODE FOR "Z"
0059 030C 0D00  CR    DATA >0D00       ASCII CODE FOR CARRIAGE RETURN
0060 030E   49  MS    TEXT 'INVALID '    ERROR MESSAGE
     030F   4E
     0310   56
     0311   41
     0312   4C
     0313   49
     0314   44
     0315   20
0061 0316 0000        DATA 0           TERMINATOR FOR MESSAGE STRING
0062 0318 8000  PM    DATA >8000       PARITY MASK FOR ASCII CHARS
0063                  *
0064                  *  ASCII-TO-MORSE TRANSLATION TABLE
0065                  *
0066 031A 0202  MT    DATA >0202       A
0067 031C 0401        DATA >0401       B
0068 031E 0405        DATA >0405       C
0069 0320 0301        DATA >0301       D
0070 0322 0100        DATA >0100       E
0071 0324 0404        DATA >0404       F
0072 0326 0303        DATA >0303       G
0073 0328 0400        DATA >0400       H
0074 032A 0200        DATA >0200       I
0075 032C 040E        DATA >040E       J
0076 032E 0305        DATA >0305       K
0077 0330 0402        DATA >0402       L
0078 0332 0203        DATA >0203       M
0079 0334 0201        DATA >0201       N
0080 0336 0307        DATA >0307       O
0081 0338 0406        DATA >0406       P
0082 033A 040B        DATA >040B       Q
0083 033C 0302        DATA >0302       R
0084 033E 0300        DATA >0300       S
```

Figure 8-20. Listing of Morse Code Program (Continued)

```
 0085 0340 0101        DATA >0101      T
 0086 0342 0304        DATA >0304      U
 0087 0344 0408        DATA >0408      V
 0088 0346 0306        DATA >0306      W
 0089 0348 0409        DATA >0409      X
 0090 034A 040D        DATA >040D      Y
 0091 034C 0403        DATA >0403      Z
 0092              *
 0093      02A0        END  ST
NO ERRORS
```

Figure 8-20. Listing of Morse Code Program (Concluded)

The SZCB instruction at location $2B4_{16}$ forces the parity bit of the received character to ZERO. Only the parity bit is affected; all other bits in the ASCII character are unaffected. This is done because the comparison characters used to check for alphabetic entries assume a ZERO parity bit.

The CB instruction at location $2B8_{16}$ is the first part of an edit check to verify the entered characters are alphabetic. If that comparison causes an arithmetic-less-than condition, control passes to location CC where the character is checked to determine if it is a carriage return. If so, control is returned to the monitor. Otherwise, the error message, INVALID is displayed by the XOP 14 instruction at location $2FE_{16}$.

If the received character is not less than the ASCII value of A, another check is made at location $2BE_{16}$ by comparing the received character to a Z. If the ASCII value of the received character is greater than the value of Z, this indicates the character is not alphabetic and is not a carriage return; therefore, program control passes to location ER where the error message is displayed, and the program waits for another character to be entered.

If the character is alphabetic, the SWPB instruction at location $2C4_{16}$ places the ASCII character value in the right byte of register 6. Then at location $2C6_{16}$, a hexadecimal 41 is subtracted from the value to produce a relative value for the character. The SLA instruction at location $2CA_{16}$ multiplies the resulting value by two to form an index value in register 6 to the word entries in the Morse code look-up table.

Using this index value, the MOV instruction at location $2CC_{16}$ puts the table entry into register 7. At location $2D0_{16}$, the entry is copied to register 6. At location $2D2_{16}$, the entry in register 6 is shifted right eight bit positions so that now only the element count remains in register 6. At this point, the elements are right justified in register 7, with the first element in the least-significant-bit position and register 6 contains the number of Morse code elements in the character.

At location $2D4_{16}$, (labeled BB), register 2 is cleared in preparation for forming a delay count for the cycle generator subroutine. The delay count is one of two values, depending upon whether the element to be transmitted is a dot or dash. A dash is three times as long as a dot. In this program, a dash is one second long; therefore, a dot is 333 milliseconds long.

The current element is identified by right shifting the rightmost bit of register 7 into the carry status bit (location $2D6_{16}$) and then checking the state of the carry bit (location $2D8_{16}$). If the carry bit is ONE, this indicates the bit is a dash, and at location $2DA_{16}$, the value $29B_{16}$ is added to the zero value in register 2.

This AI immediate instruction at location $2DA_{16}$ is followed by a second AI instruction at location 2DE (labeled DT) to form

the total delay count for a dash element. If the element had been a dot (indicated by a ZERO bit), the AI instruction at location $2DA_{16}$ would have been skipped and only the value $14D_{16}$ would have resulted in register 2.

At location $2E2_{16}$ there are one of two values in register 2: either $14D_{16}$ (333_{10}) or $3E8_{16}$ ($29B_{16} + 14D_{16} = 3E8_{16} = 1000_{10}$). The value in register 2 is the number of milliseconds duration for the tone: 333 milliseconds for a dot, 1000 milliseconds (1 second) for a dash. Recall that registers 0 and 1 were both loaded with the value ten earlier (at locations $2A8_{16}$ and $2AC_{16}$) to establish a 1-kHz tone for the cycle generator program. Each cycle is one millisecond long so that the value in R2 determines the tone duration in <u>milliseconds</u>.

At location $2E2_{16}$ the BL instruction makes a subroutine call to the cycle generator program.

The BL instruction transfers program control to the cycle generator program and stores the address of the next instruction in this program in register 11 of the common workspace area.

The cycle generator program must be modified to make it a subroutine by changing the B instruction at the program's exit point to a B *R11 instruction. This modification will cause a return to be made by the cycle generator subroutine to the address contained in register 11.

With the above modification, the cycle generator program will return control to the LI instruction at location $2E6_{16}$. There, the value $1A0A_{16}$ is placed in register 4, and a 1 is placed in register 2 (location $2EA_{16}$). Another subroutine call is made to the cycle generator program (location $2EE_{16}$) at entry point D1 to cause a 333-millisecond pause after each element. Morse code requires an intermediate pause after each element in a character equivalent to the length of a dot. Notice that $1A0A_{16}$ equals 6666_{10}, which will cause an off time in the cycle generator subroutine of 6,666 50-microsecond periods (or 333 milliseconds).

Following this delay, program control is returned by the cycle generator subroutine to the DEC instruction at location $2F2_{16}$ where the count of elements in register 6 is decremented by one. If the result of this decrement is zero, this indicates all elements have been sent, and the JMP instruction at location $2F6_{16}$ transfers program control to location AA where another character is read from the keyboard.

If the element count is still greater than zero, the JNE instruction at location $2F4_{16}$ transfers program control to location BB where the next element in register 7 is analyzed and transmitted.

Notice the TEXT assembler directive at location $30E_{16}$ which causes the characters I, N, V, A, L, I, D, and "space" to be assembled into their ASCII character code at sequential byte addresses. The

zero value in the left byte of location 316_{16} is the message terminator for the XOP 14 at location $2FE_{16}$ which displays the character string. (This DATA directive at location 316_{16} is not really necessary with the UNIBUG assembler since the TEXT directive with the UNIBUG assembler automatically puts a zero byte after the last character.)

The structure of the Morse code translation table beginning at location MT has been discussed. By analyzing this table, confirm that there are two elements, a dot followed by a dash, in an A and that there are four elements in an F, two dots followed by a dash and another dot.

There are other valid characters in the Morse code set besides alphabetic characters. If it is desired for the program to accept other characters in addition to alphabetic characters, it is necessary to add the characters to this table and to modify the validity checking portion of the program.

Program Operation

Assemble this program along with the cycle generator program. (Remember to change the B instruction at the exit of the cycle generator program to a B *R11.) Set the program counter to the address of ST in this program and run it. Try entering on the keyboard both valid and invalid characters. The program can be terminated by entering a carriage return.

8.9 SUMMARY

The judicious use of subroutines promotes program modularity. The advantages of program modularity include memory economy, easier debugging, and increased flexibility.

In this chapter, the structure of subroutines is discussed. With the 9900 family of microprocessors, subroutines can be structured to have their own set of working registers. The architecture of the family allows a context switch to be made from one program with a set of working registers to another program with its own set of registers. Context switching can be made from software by using two specific instructions in the TMS 9980A's instruction set (BLWP and XOP) or as the result of an external interrupt from hardware.

The final subset of the TMS 9980A's instruction set is introduced. This subset includes the single instruction multiply and single instruction divide.

The subroutines within the UNIBUG monitor that are available to the user are described. These subroutines provide useful functions that are often needed in a user's program. Some of these functions are used in a program example.

The program example illustrates not only the UNIBUG monitor calls (which are made with an XOP context switch) but also the use of a subroutine (the cycle generator program) which is called without a context switch.

8.10 EXERCISES

1. How do subroutines relate to the concept of program modularity?

2. Why is a return address important to a subroutine?

3. Why is it important that a subroutine be restricted to a specific function?

4. What particular feature of the TMS 9980A's register architecture contributes to an improved interrupt response time?

5. What two instructions in the 9900 microprocessor family's instruction set cause a context switch? How are these instructions alike? How are they different?

6. What are the addresses of the two words used as a vector for a level-4 interrupt?

7. Assume that the following memory locations contain the following values:

Location (hexadecimal)	Contents (hexadecimal)
0000	3240
0002	32C0
0004	1208
0006	35F0
0008	1706
000A	2CDE
000C	21EC
000E	3870
0010	14A2
0012	14E0
0014	0300
0016	0320

Further assume that immediately before a level-3 interrupt context switch is made, the TMS 9980A's internal registers contain the following values:

(WP) = 024E
(PC) = 047C
(ST) = 1004

What are the contents of these three internal registers immediately following the context switch?

8. What memory locations would be used to save the contents of the three internal registers as a result of the above context switch?

9. What binary value must appear on input lines IC0, IC1, and IC2 to request a level-3 interrupt?

10. Consider the following sequence of instructions.

```
BL   @SR
DATA AA
DATA BB
JMP  ER
JMP  GD
```

Write a subroutine at beginning location SR to add together the two 16-bit, signed values passed by the calling program as a result of the subroutine call. Assume the two values are at addresses AA and BB. Place the result in working register 0 and return control to the fourth word following the subroutine call instruction. If overflow occurs as a result of the addition, return control to the third word following the subroutine call.

11. Modify the above subroutine assuming the subroutine call was made by a BLWP instruction, rather than the BL instruction. Return the result in working register 0 of the calling program.

12. What are the addresses of the two words used as a vector for an XOP 11 instruction?

13. Assume the low-order four bits of the status register contained the value D_{16} immediately before an XOP 9 instruction. What would be the value in these four bits immediately following the XOP instruction?

8.11 LAB EXPERIMENTS

1. Modify the program example to output Morse code on an LED, rather than the speaker. Recall that the CRU software base address for the rightmost user addressable LED is 20_{16}.

2. Modify the program example to accept additional characters besides alphabetic characters.

3. Modify the program example to allow an operator to select the transmission rate. Use the UNIBUG monitor calls to prompt the operator for the desired transmission rate and to read the answer. The transmission rate could be expressed as a relative speed on a scale of one to ten or a selected rate of words per minute.

In determining the rate, five characters are considered to be a word. During the transmission, there should be a space following each element equal to the dot time. Following each character, there should be a space equal to three dot times. A word should be followed by a space equal to seven dot times.

4. Modify the program example to send automatically a sequence of characters (such as a station identification) when a special key is entered.

5. Modify the program example to accept a character from the terminal keyboard and display the Morse code representation for the character as a sting of ONE's and ZERO's.

6. Modify the program example to use the TMS 9901 for timing purposes rather than the cycle generator subroutine.

7. Write a subroutine using the multiply instruction to perform a signed multiply. Write another subroutine to perform a signed divide.

8. Write a program utilizing the UNIBUG monitor calls to read a hexadecimal digit from the keyboard and display the binary equivalent on the four LED's.

9. Write a program to read a series of characters from the TM 990/189 terminal and echo each received character on the display. If a nonalphabetic character is received, write the message INVALID to the display and return control to the monitor.

10. Write a program to accept a series of four-digit hexadecimal values from a keyboard. Convert each received hexadecimal value to decimal and display the converted value. Upon receiving a value terminated by a minus sign, return control to the monitor.

CHAPTER 9

SOFTWARE ENGINEERING

9.1 INTRODUCTION

This chapter describes "software engineering," a term used to indicate that there are disciplines associated with the structuring of software just as there are disciplines associated with "hardware engineering."

One of the first steps toward successful software engineering is to determine which functions of an application are to be implemented in software rather than hardware. Factors affecting these hardware-versus-software decisions are of primary importance in programming.

After the software functions have been determined, other software engineering disciplines which can increase development speed and reduce software errors include top-down design, structured programming, and modular programming. The basic principles of modular programming are described in the previous chapter. This concept includes dividing the application software into separate program functions, or modules.

Once programs have been divided into functional modules, they must be developed and must ultimately reside together in the system's memory. Methods that can be employed to link these modules together in memory are discussed.

A system may employ interrupt-driven I/O. With the addition of interrupts to a microprocessor system, a new spectrum of considerations confronts the system designer. These considerations are discussed along with the concept of "real-time" system design and real-time operating systems. The concepts of real-time system software design are illustrated in a program example.

9.2 HARDWARE/SOFTWARE TRADEOFFS

A microprocessor system, like any computer system, is a combination of both software and hardware. A system is composed of many individual functions. The system designer often has a choice of implementing specific functions in hardware or in software. The choice of which functions to implement in hardware and which to implement in software is often dictated by cost/performance tradeoffs.

Generally, as a system's performance increases, so does its cost. Figure 9-1 illustrates this relationship.

Cost/Performance Tradeoffs

Many microprocessor-based designs represent products that are profit-oriented. These products must be designed toward the least cost possible, consistent with required performance. As a general rule, it is often less expensive to implement functions in software, but faster performance can be gained by implementing them in hardware.

As a simple example of the decision flexibility a system designer may have, consider a cycle-generation function similar to the program example in Chapter 6.

Suppose a system designer is given the task of generating a variable-tone generator for a microprocessor-based system. One approach might be "a hardware solution" illustrated in Figure 9-2.

The system designer could use a hardware "black box" tone oscillator to drive a speaker. The tone oscillator produces a variable tone according to an analog value presented to it. The tone oscillator can be turned on and off with a simple switch.

This "hardware approach" would work well and might even be desirable if the frequency of the desired tone needed to be very precise.

A more cost-effective "software approach," as illustrated in Figure 9-3 could be used. With this approach, the speaker can be controlled by turning a simple transistor amplifier on and off. The software in the system's memory controls the amplifier. By controlling (through software) the time period that the amplifier is on and off, the tone and its duration can be controlled.

The software approach also reduces production cost of the system (tone oscillator and A/D converter are relatively expensive components). However, there could be a performance limitation to the software approach. For example, in the cycle generator program from Chapter 6, 50 microseconds was the smallest half-cycle period that could be requested. With this limitation, the highest tone that can be generated is 10 kHz. If it becomes necessary to produce tones of higher frequencies, it is then necessary to implement a hardware approach (or, possibly, to use a faster version of the processor).

Run-Time Speed Versus Development Speed

When discussing performance, speed is the primary criterion of measurement. There are two different speeds to be considered, however. There is, first, "run-time speed;" that is, how fast

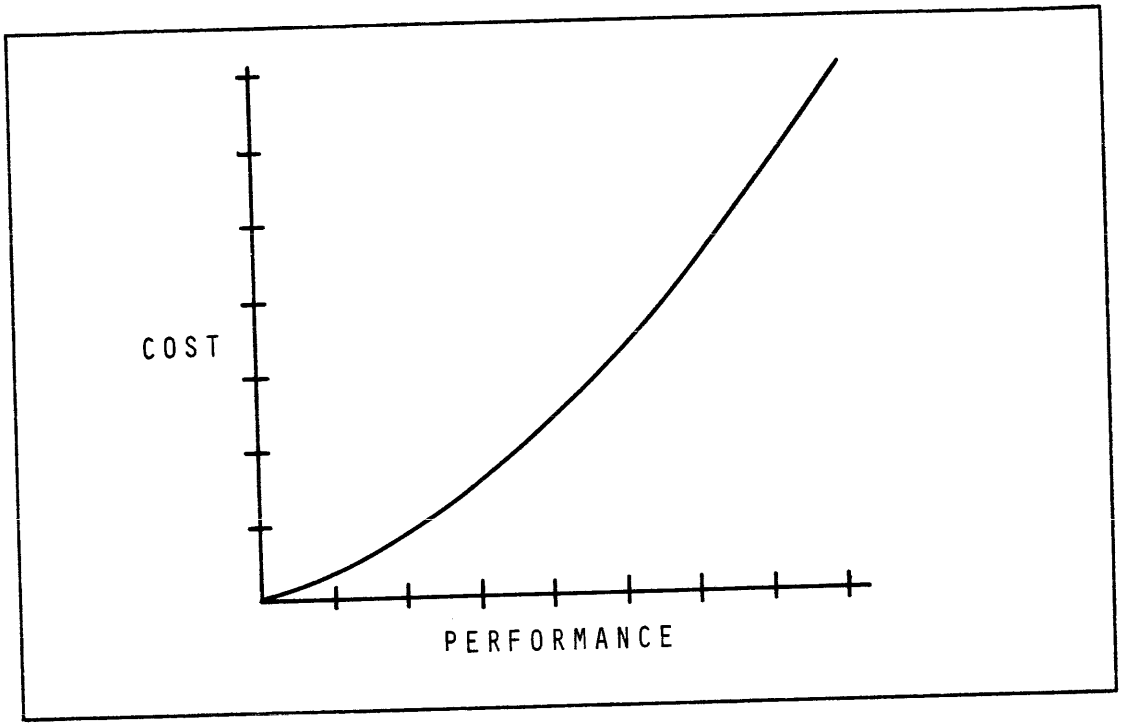


Figure 9-1. System Cost versus Performance

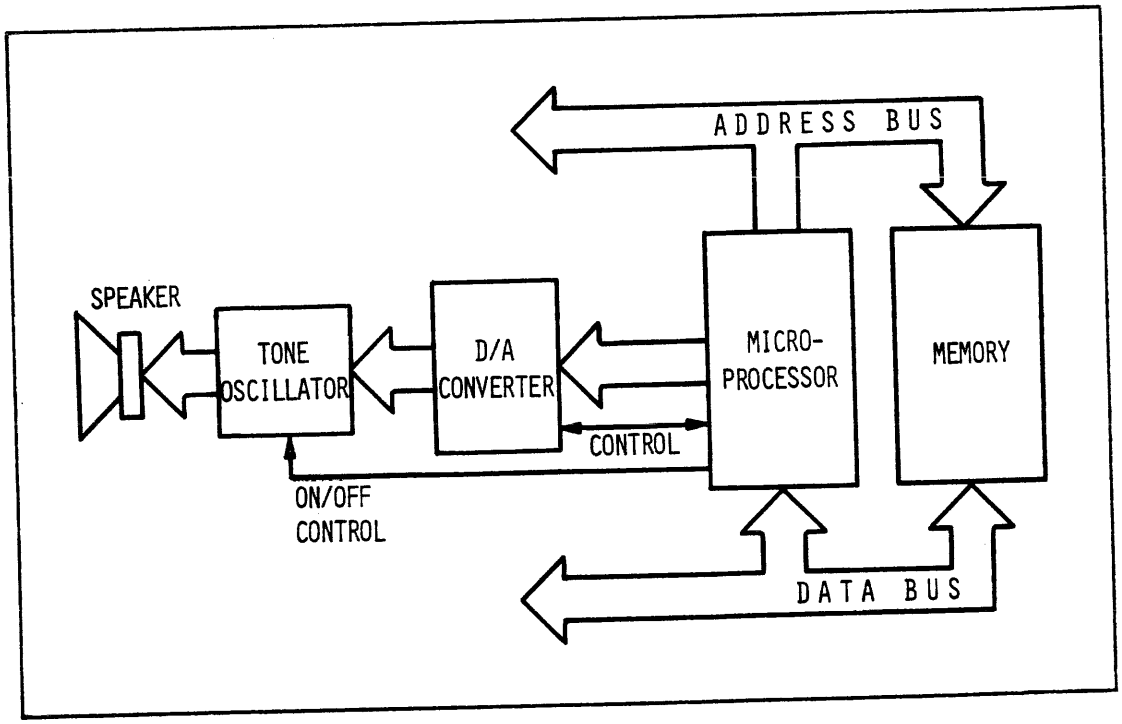


Figure 9-2. "Hardware Approach" For Tone Generator

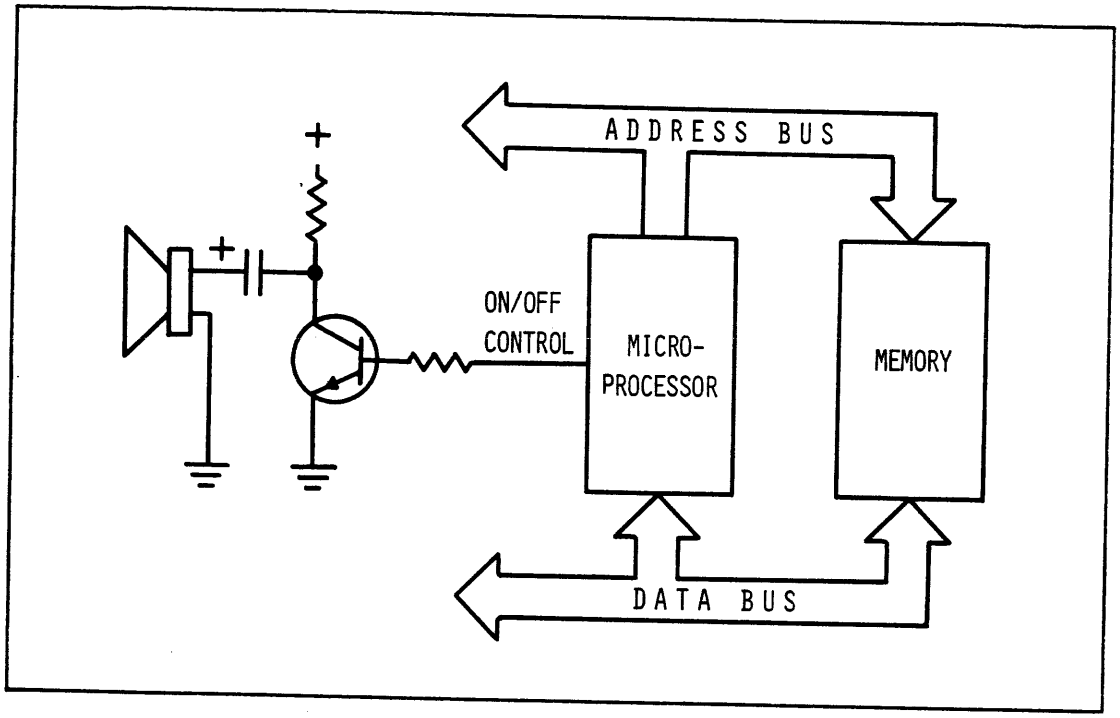


Figure 9-3. "Software Approach" For Tone Generator

the system runs when it is operating. Another "speed" to consider is how fast the system (especially the software) can be developed. Run-time speed is affected by hardware/software considerations. Development speed is also affected by these same variables.

The level of language chosen for the software can significantly impact the development time necessary to define the software functions.

As described in Chapter 4, the levels of language generally available for software development range from machine language to assembly language to high-level language. Usually, machine language is used only for very limited size applications or for temporary "patching" of programs under development.

For most systems, the choice lies between assembly language and high-level language.

Development time normally proceeds faster with high-level language, but less-efficient machine code is generated. The resulting less-efficient code takes longer to execute at run-time and is less memory efficient (takes more memory space) than the same functions designed with assembly language. With microprocessor systems that are production oriented, the system designer often chooses assembly language to minimize the production costs (by reducing the number of memory components required) and also to increase the throughput

of the system. Assembly language also gives the programmer direct control of the hardware. High-level language does many bookkeeping tasks for the programmer, and "surprises" can arise when the language handles a situation differently than the programmer expects.

Currently, the majority of larger volume production systems are created by use of assembly language. Memory components are becoming less expensive and high-level language translators are becoming more efficient and easier to use in writing application programs; thus, the trend is for more application software to be written in high-level language.

Flexibility

Independent of the language chosen, however, maximum flexibility is usually gained by a software approach rather than a hardware approach.

Other Considerations

There are other considerations that affect the hardware/software tradeoff decisions. One of these is the experience of the persons involved with the design. If the individuals developing a system have more experience in hardware, the problem solutions tend to be conceived as hardware solutions. The converse is true for persons with software backgrounds. Another consideration is the availability of existing software or hardware to perform a function. If the solution to a problem already exists, it may be more practical to continue with the existing solution.

9.3 STRUCTURING THE SOFTWARE

Three software engineering methodologies that are especially important in terms of potential benefits to the development of software are

- Top-down design
- Structured programming
- Program modularity.

Top-Down Design

Concept. Top-down design is the process of breaking down a large problem into a number of smaller problems. With top-down design, the software engineer begins with a primal description of the application software. From this broad, basic description, the general functions that must be included in the application are defined. Then the individual program modules required to perform these general functions are defined. The software design flow is from the very general to the more specific.

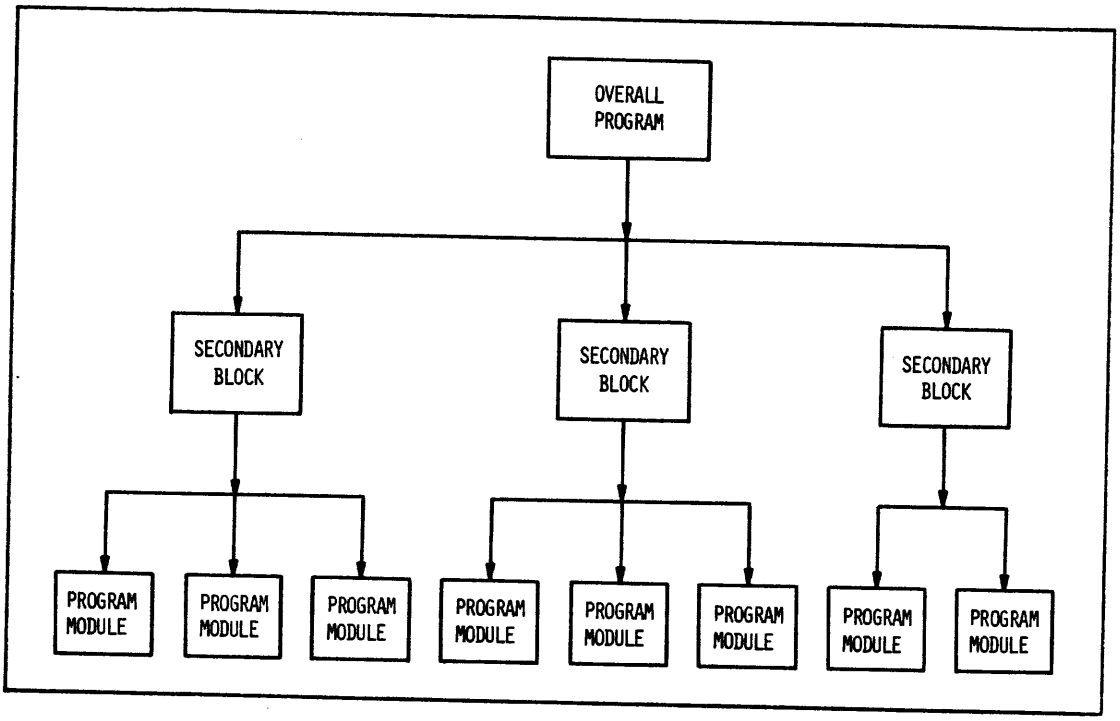


Figure 9-4. A Block Diagram of a Top-Down Design

The design process is expressed as a block diagram with the structure shown in Figure 9-4. The top block describes the overall application. The secondary blocks describe the general functions included in the application, and the lower level blocks define the specific program modules required by each of the functions. There could be several levels of blocks (depending upon the complexity of the application) and the design process continues until the program blocks are broken down into manageable pieces.

Advantages. The use of top-down design offers several advantages.

First, it establishes a framework for a structured, systematic approach toward both developing and testing the software. As each separate block is developed and tested, the software designer has a good yardstick to measure the portion of the task that has been completed and the portion remaining to be completed.

Second, by first defining and testing the overall systems logic, the system designer can detect and correct fundamental system design errors that would otherwise be much more difficult and time-consuming to fix after the entire system (including all the individual program modules) has already been developed.

Third, a top-down design promotes program modularity, which tends to allow the writing, testing, and integration of a system to be done in smaller, more manageable sections. The modules can then be written individually by members of a programming team.

Fourth, a top-down design makes it more likely that the individual program modules can be tested and debugged in an applications environment; it is less likely that special programs will need to be developed to test the individual program modules.

Studies indicate that top-down design can improve the productivity of software designers, and the resulting software is easier to debug and understand.

Disadvantages. Like most things, however, top-down design has certain disadvantages. One disadvantage is that the designer must often make important decisions regarding the overall system design earlier in the development stage. Sometimes these decisions are made before enough information has been gained to understand what problems may be encountered once the specific program modules are created.

Secondly, if the application software is going to adopt existing hardware or software, it may not fit into the top-down design easily.

Thirdly, top-down design works best with a relatively simple program structure. Complex applications with complex interconnections do not easily lend themselves to the simpler top-down design.

The opposite of top-down design is the "bottom-up approach," where specific, individual program modules are developed first and then integrated into an overall application program environment. A bottom-up design compares to a top-down design as inductive reasoning compares to deductive reasoning.

Most important, however, is the final objective, and that is to get the application to work. It is more important that the system work correctly than to conform rigidly to some particular design philosophy. In many cases, the final design will be a combination of the top-down and bottom-up approaches.

Structured Programming

Once an application program has been designed (using either top-down design or bottom-up design), structured programming can be used for the detailed program design and the coding process.

Concept. The goal of structured programming is to reduce the debugging time (normally one of the most, or the most, time-consuming part of the development effort) by simplifying program

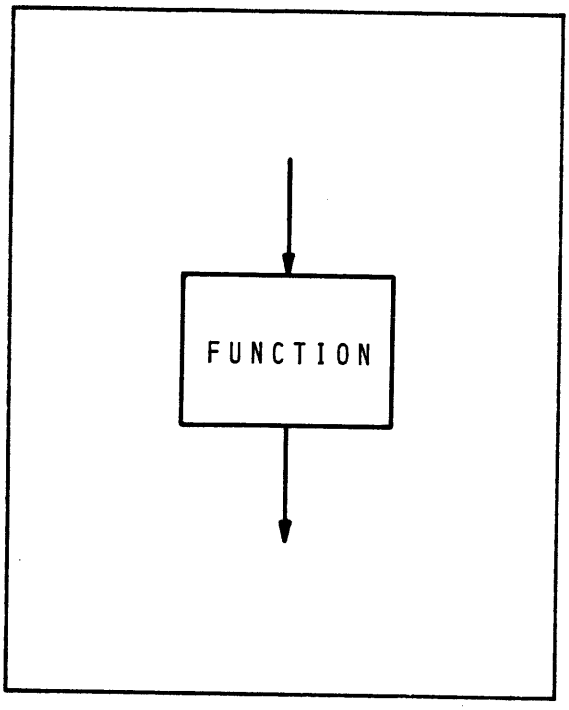


Figure 9-5. The Linear Structure

structure. By reducing the debugging time, the development time (and, normally, the development cost also) is reduced.

Structured programming can potentially reduce the debugging time in two ways. One, the adoption of structured programming techniques can reduce the number of bugs introduced into the program as it is written. Second, if a bug is detected as a program is being tested, the bug is often easier to detect and correct if structured programming techniques have been used.

The foundation of structured programming rests on two key principles:

- Only a specific set of program structures is permitted.
- Each of the program structures has only one entry point and one exit point.

Key Program Structures. There are three key program structures used in structured programming:

- Linear or simple structure
- Decision structure
- Conditional structure.

The linear structure is a set of instructions (this set may be a subroutine or a program module) that performs one function. The flow of the program is linear. No decisions are made within the function that would transfer control outside the function. Any transfer is made at the end of the structure. The linear structure is shown in flowchart form in Figure 9-5.

As an example of the linear structure, consider the subroutine in Figure 9-6 which accepts two 16-bit numbers in registers 4 and 5, adds the two numbers, subtracts a constant from the result, stores the answer in memory location RS, and returns control to the calling program.

```
AD      A       R4,R5       ADD THE TWO NUMBERS
        S       @CN,R5      SUBTRACT A CONSTANT
        MOV     R5,@RS      STORE THE ANSWER
        B       *R11        RETURN
```

Figure 9-6. Example of the Linear Structure

The decision structure provides for a test to be made and, based upon the result of the test, for different functions to be performed. The basic form of this structure is the IF-THEN-ELSE structure shown in Figure 9-7.

Based upon the result of the test, either one function or the other is performed. Notice that the programming structure has one entry point (into the decision diamond) and one exit point (after one of the two functions is performed).

A corollary of the IF-THEN-ELSE structure is the structure where a test is made, and, based upon the result of the test, a function is or is not performed. This structure is shown in Figure 9-8. Notice the similarity to the IF-THEN-ELSE structure in Figure 9-7.

An example of the decision structure is the subroutine in Figure 9-9. After the two numbers in R4 and R5 are added together, one constant is subtracted if the sum is larger than zero, or another constant is subtracted if the sum is zero or negative. The answer is then stored in memory location RS and control is returned to the calling program.

The conditional structure provides for repetition of a process (a loop). Two forms of this structure are the DO-WHILE structure and the DO-UNTIL structure. With both forms, a defined process (or function) is performed repetitively until some terminating condition allows the program to "fall out of" the loop.

The DO-WHILE structure is shown in Figure 9-10. An example of the DO-WHILE structure is illustrated by the subroutine in Figure 9-11. After the two numbers in R4 and R5 are added together, the result is compared to zero. If the result is positive, a constant is subtracted and the result is again compared to zero. As long as the result remains positive, the constant is subtracted.

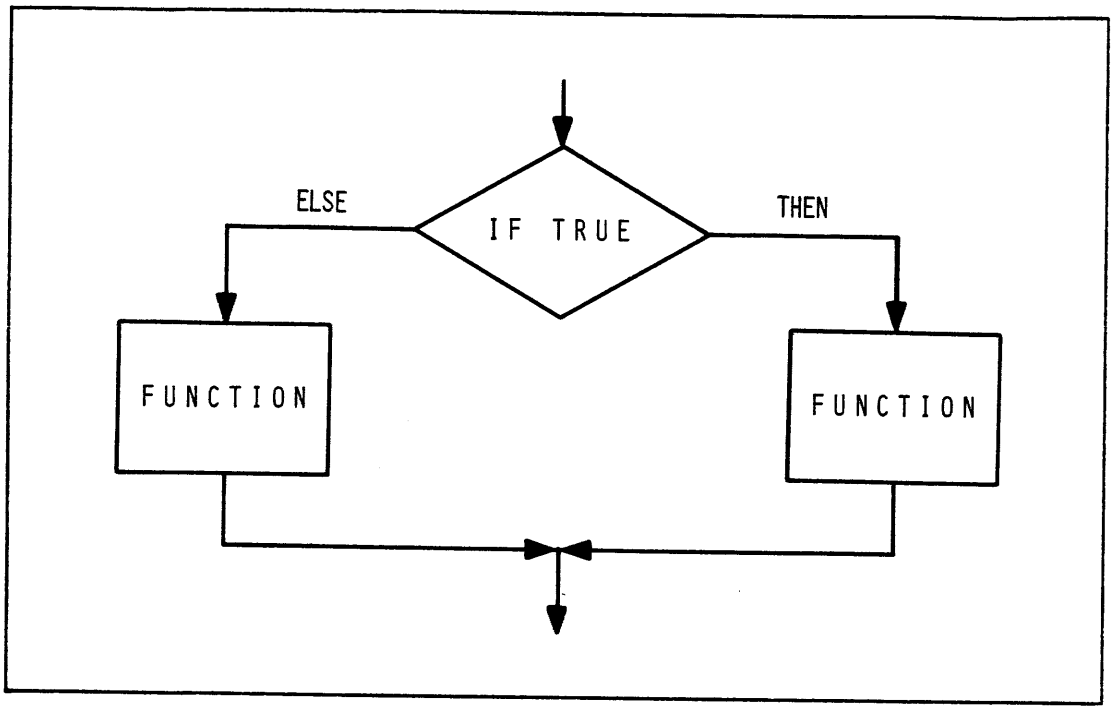


Figure 9-7. The IF-THEN-ELSE Structure

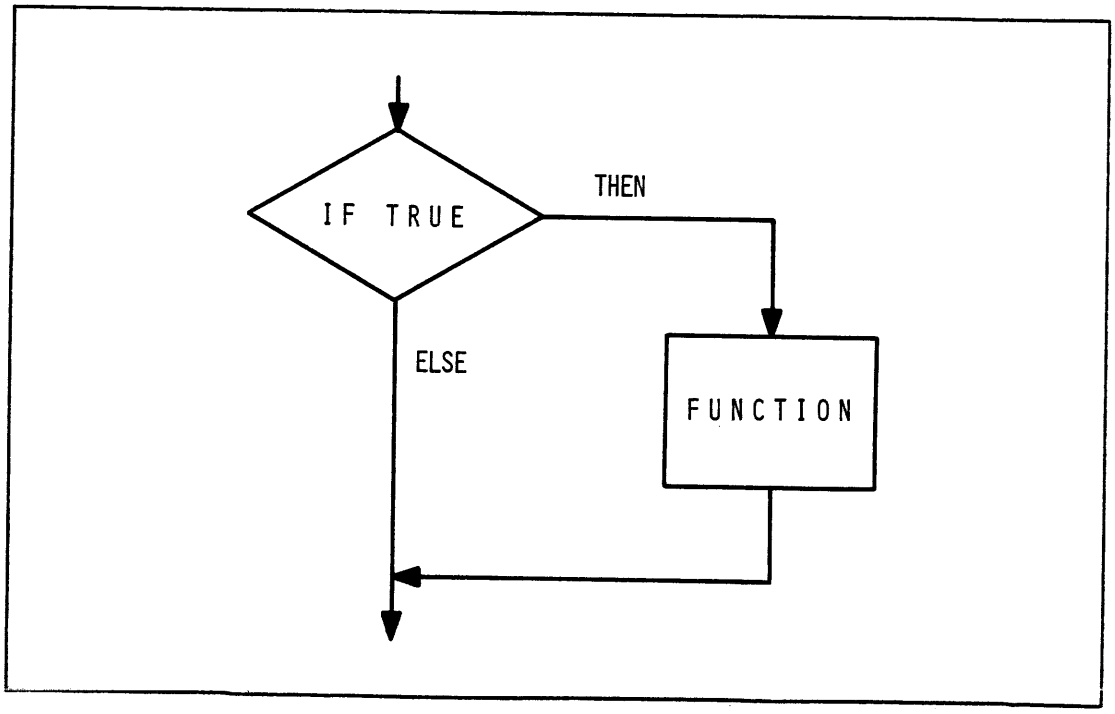


Figure 9-8. The IF-THEN-ELSE Structure With Only One Function

```
AD      A       R4,R5       IS R4 PLUS R5 GREATER THAN ZERO?
        JGT     LG          YES - GO TO LG
        S       @C1,R5      NO-ADD A CONSTANT
        JMP     ON            AND GO STORE ANSWER
LG      S       @C2,R5      ADD ANOTHER CONSTANT
ON      MOV     R5,@RS      STORE THE ANSWER
        B       *R11        RETURN
```

Figure 9-9. Example of the Decision Structure

When the result becomes negative, the answer is stored in memory location RS and control is returned to the calling program.

With the DO-WHILE loop, the condition is tested before the process is performed. The DO-UNTIL program structure is similar except that the process is performed first and then a test is made to determine if the process is to be repeated. With the DO-UNTIL structure (as shown in Figure 9-12), the process is performed at least once.

Figure 9-13 is an example of the DO-UNTIL structure. After the two numbers in R4 and R5 are added together, a constant is subtracted from the sum. After the subtraction process, the result is compared to zero. If the result is positive, the constant is subtracted again and again until a negative result occurs and then the negative number is stored in memory location RS. Notice that with this DO-UNTIL structure the constant is subtracted at least once.

As with the linear sequence and the decision sequence, there is only one entry point into, and only one exit point out of, the conditional sequence.

The three basic programming structures can be combined (or nested) to build more complex programs. Figure 9-14 shows how several individual programming structures can be combined to produce a more complex program.

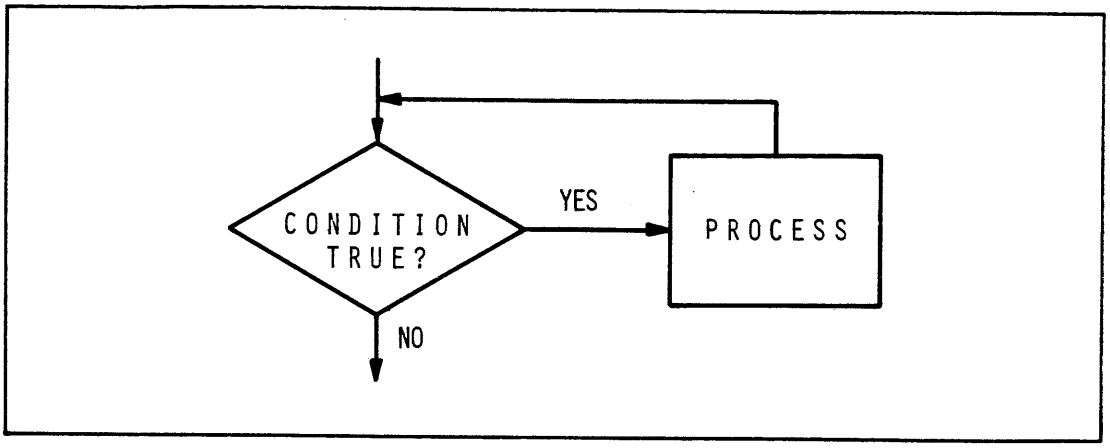


Figure 9-10. The DO-WHILE Structure

Advantages. The two greatest advantages of structured programming are reduced debugging time and better documentation. Reduced debugging time results from several things. By using structured programming techniques, the designer is less likely to introduce errors into the program when it is written. The logic of the program can be "desk checked" (analyzed mentally) more easily since there is only one entry point and one exit point for each structure. During testing, if a bug is found, the exact problem is easier to detect. Also, the documentation produced with structured programming makes the program logic easier to understand, especially by someone other than the software designer.

Disadvantages. Just like top-down design, however, structured programming does have certain disadvantages. First, it requires discipline and often requires more effort to write (especially as the techniques are being learned where nonstructured techniques have been used in the past). Second, a program developed using structured methods often requires more memory and executes slower than if the program were developed in its most performance-efficient form. Finally, some programming languages are better suited for structured programming than others. Assembly language generally does not lend itself as readily to structured programming as does high-level language. Among the high-level languages that do promote structured programming well is Pascal. The Pascal language is drawing an increasing amount of interest as a programming language for microprocessors. High-level languages, including Pascal, are becoming more widely used for microprocessor applications as the cost of memory components declines and the efficiency of the language translators improves.

```
AD    A      R4,R5      ADD THE TWO NUMBERS
SB    JLT    ST         IF RESULT NEGATIVE, TERMINATE
      S      @CN,R5     ANSWER NOT NEGATIVE, SUBTRACT CONSTANT
      JMP    SB         REPEAT
ST    MOV    R5,@RS     R5 NEGATIVE - STORE RESULT
      B      *R11       RETURN
```

Figure 9-11. Example of the DO-WHILE Structure

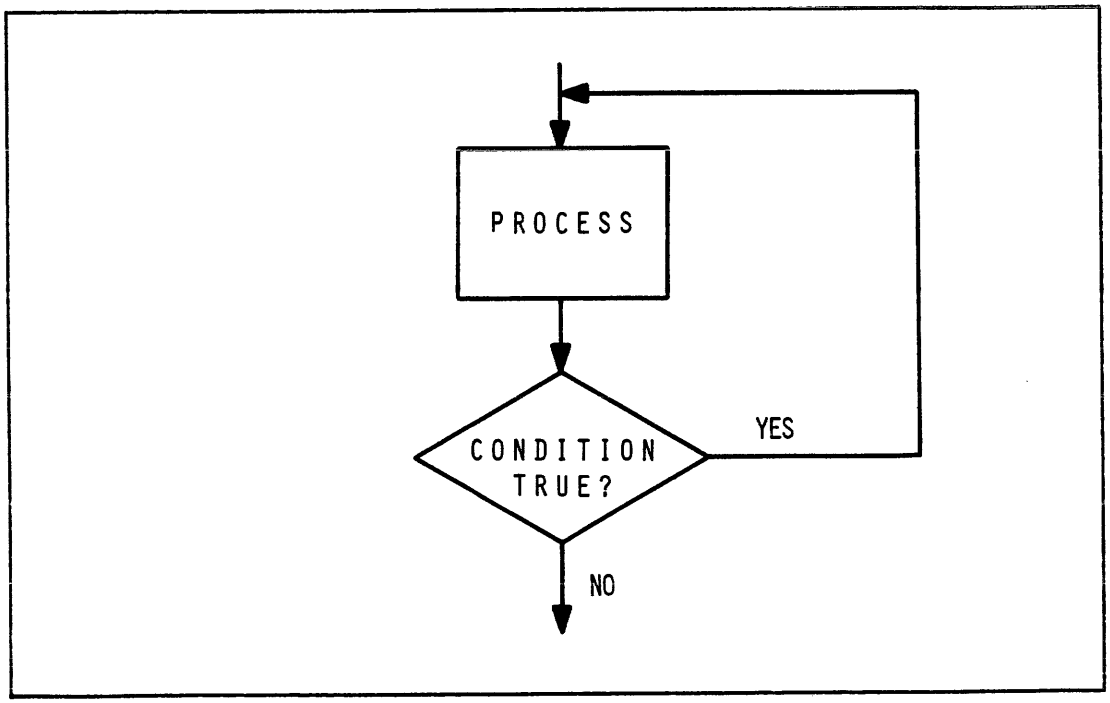


Figure 9-12. The DO-UNTIL Structure

```
        A       R4,R5       ADD THE TWO NUMBERS
SB      S       @CN,R5      SUBTRACT A CONSTANT
        JLT     ST          IF RESULT NEGATIVE, TERMINATE
        JMP     SB            OTHERWISE, REPEAT
ST      MOV     R5,@RS      STORE RESULT
        B       *R11        RETURN
```

Figure 9-13. Example of the DO-UNTIL Structure

Program Modularity

Along with top-down design and structured programming, a third concept that is important to software engineering is program modularity. The concept of program modularity, its advantages and disadvantages are discussed in the previous chapter. The use of top-down design promotes program modularity, and the adoption of structured programming techniques can simplify the debugging of individual program modules.

All of the above software engineering disciplines contribute to improved documentation. Thorough documentation should be a goal of all system development efforts.

9.4 LINKING PROGRAM MODULES

Most application programs are composed of several individual program segments or modules. All of these individual program modules must reside in memory along with data constants and data-storage areas. The separate program modules cannot occupy the same memory space, of course, but they should reside in memory as compactly as possible with as little intervening memory space as practical. This keeps the amount of memory space required (and, hence, the number of memory components required) as small as possible. A combination of techniques for fitting these program modules together in memory effectively is the subject of the following discussion.

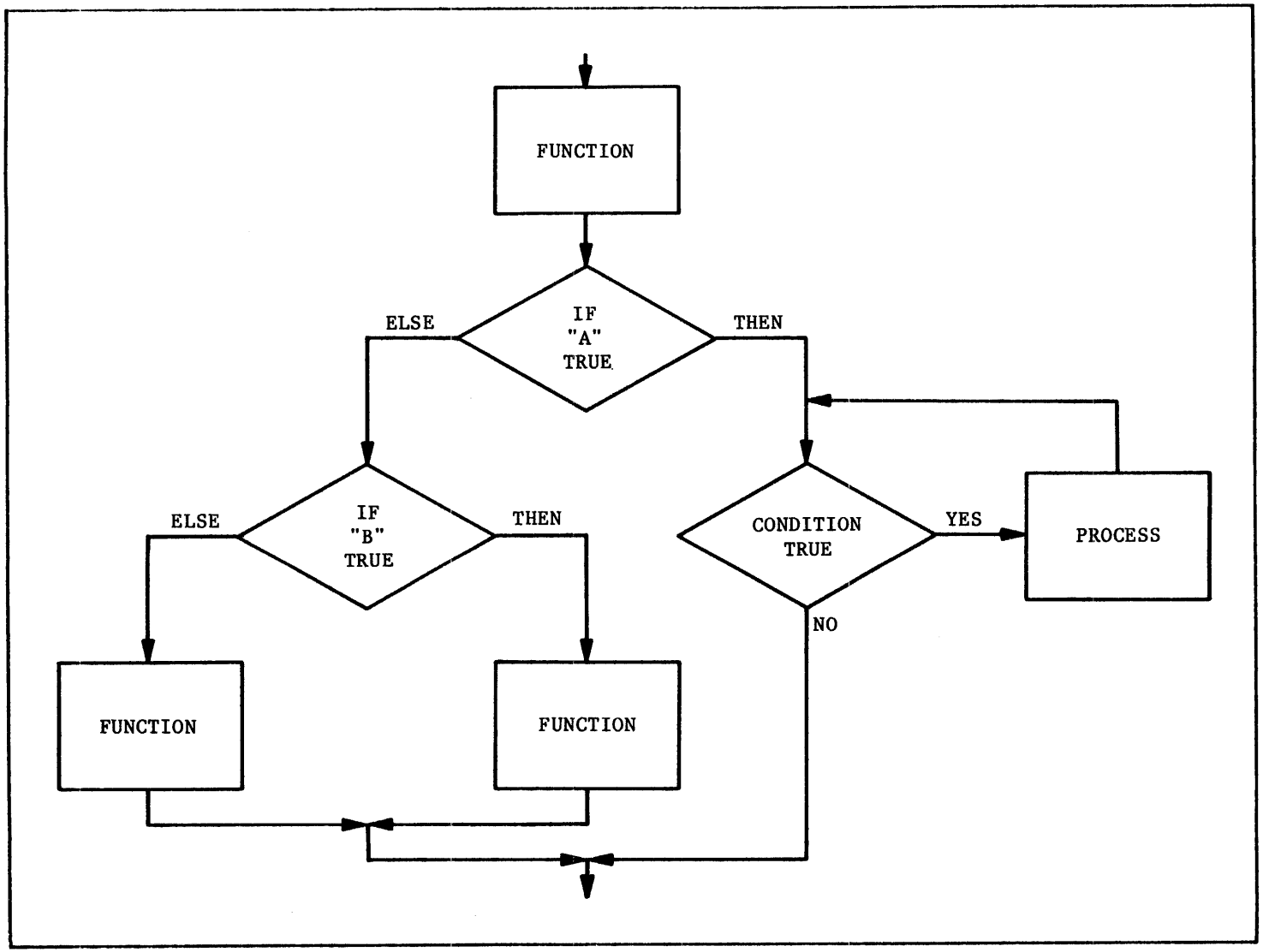


Figure 9-14: Individual Programming Structures Combine to Form a More Complex Program

ROM/RAM Division

Most application programs created for microprocessors reside in nonvolatile memory (ROM) once the design is completed. The instructions are placed in ROM so that they will remain intact and not have to be reloaded when power is removed. Data constants (that is, data that does not change) can also be placed in ROM.

Most programs do, however, require work areas. These are used to store data values that may change and, in the case of the TMS 9980A, one or more workspace register areas that must be in read/write memory (RAM).

In addition, an application may employ memory-mapped I/O in which one or more I/O devices are assigned memory locations so that they can be accessed with memory-reference addressing.

Memory Space Allocation

As a first step toward linking programs together in memory, the software designer creates a memory map of the system indicating the range of memory addresses available for ROM, RAM, and I/O. An example of a memory map for a TMS 9980A-based application is shown in Figure 9-15.

Notice that 8K bytes are allocated for ROM memory (addresses 0000_{16} through $1FFF_{16}$), 4K bytes for RAM (2000_{16} through $2FFF_{16}$), and 4K bytes for memory-mapped I/O addresses (3000_{16} through $3FFF_{16}$).

As a further refinement, the software designer allocates space within the ROM, RAM, and memory-mapped I/O space to accommodate the individual program modules, specific data areas, or individual I/O device addresses. An example of how these address spaces can be defined further is illustrated in Figures 9-16 (ROM), 9-17 (RAM), and 9-18 (memory-mapped I/O).

Program Module Memory Assignment. In Figure 9-16, memory locations 0000_{16} through $000B_{16}$ have been allocated for the three interrupt vectors: RESET, level 1, and level 2. Memory locations 0040_{16} through $007F_{16}$ have been allocated for the XOP vectors. Memory locations between 0080_{16} and 0100_{16} are used for data constants. The 512 bytes from addresses 0100_{16} to 0300_{16} are reserved for program module A. Memory locations from 0300_{16} to $1B00_{16}$ are divided between program modules B, C, D, E, F, and G. The remaining 1280 bytes of ROM space from address $1B00_{16}$ through $1FFF_{16}$ are not used in this application but are available for program expansion if additional program functions are added.

Variable Data Memory Assignment. The RAM memory map in Figure 9-17 is divided among workspace areas, variable data storage areas,

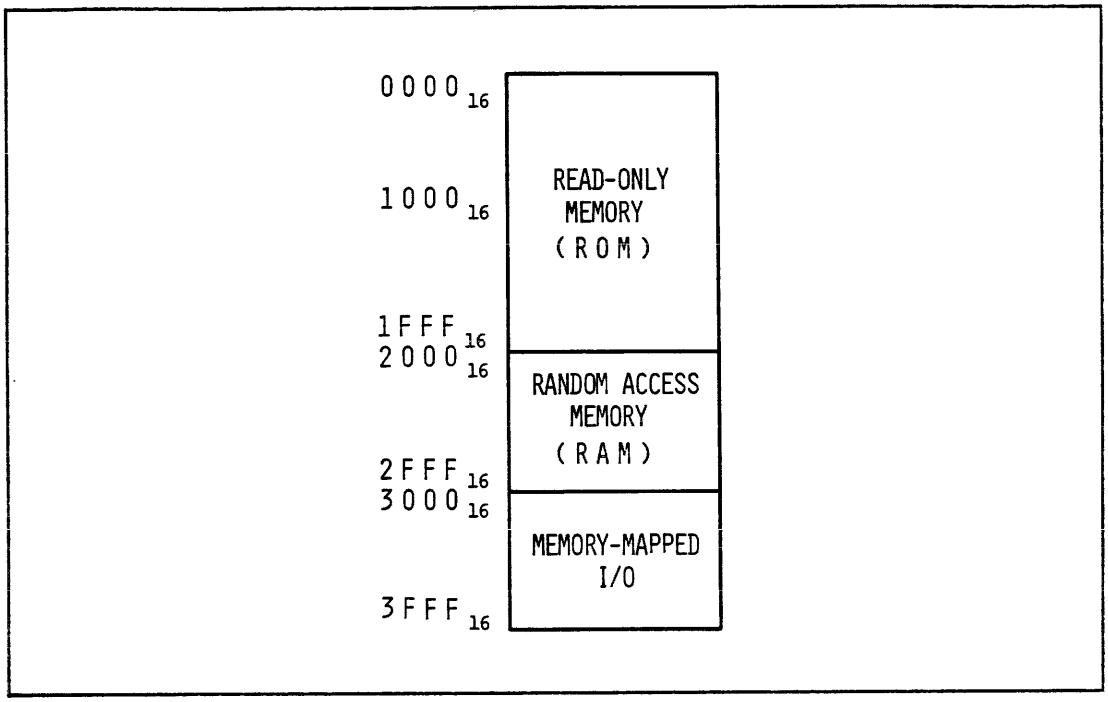


Figure 9-15. A Memory Map for a Typical TMS 9980A-Based Application

data buffers,and unused area. Memory locations 2000_{16} to 2020_{16} have been reserved for the main workspace register area. Addresses 2020_{16} to 2080_{16} are set aside for the three workspace register areas used by interrupt levels 0 (RESET), 1, and 2 interrupt-service routines.

The 128 bytes of RAM between addresses 2080_{16} to 2100_{16} have been assigned to variable data storage. The 256 bytes from location 2300_{16} to 2400_{16} have also been set aside for variable data storage. The 512 bytes of RAM between addresses 2100_{16} and 2300_{16} are used for data buffers.

Memory-Mapped I/O Memory Assignment. As shown in Figure 9-18, the memory addresses from 3000_{16} through $38FF_{16}$ have been assigned to memory-mapped I/O devices. The selection of these addresses simplifies the decoding of addresses for I/O. The SN74154 shown in Figure 9-19 is a 4-bit input/one-of-16 line-output decoder. Only when both address lines A0 and A1 are logic ONE's is the decoder enabled. Address lines A2 through A5 are used by the decoder to select each of the nine different devices. Notice that each of the devices can be selected by more than one address. For example, device A will be selected by any memory I/O address from 3000_{16} through $30FF_{16}$, and device B can be enabled by address 3100_{16} through $31FF_{16}$. Even though a device can be selected by more than one address, normally the software designer chooses one specific address to enable each device.

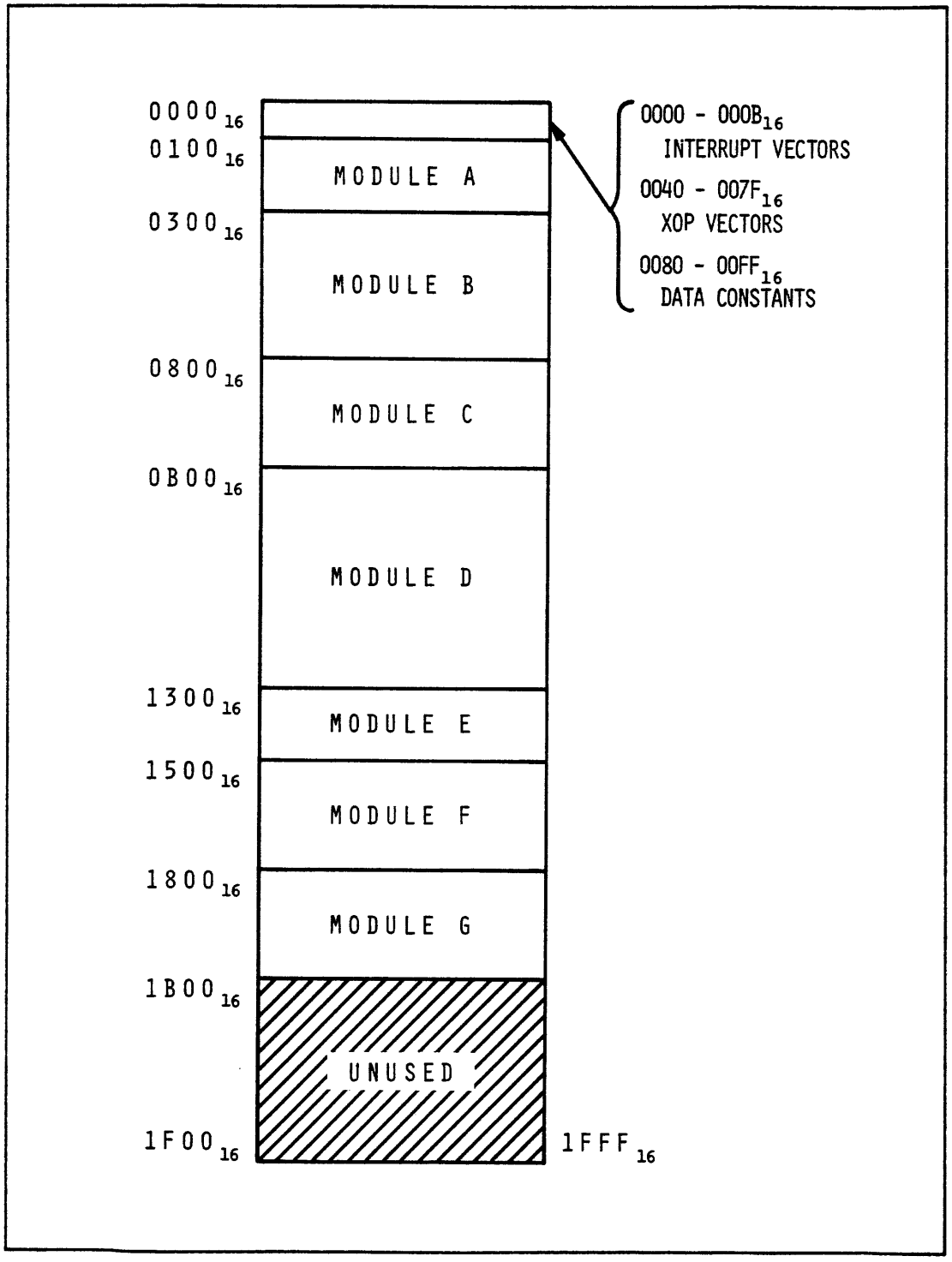


Figure 9-16. ROM Memory Space for a Typical Application

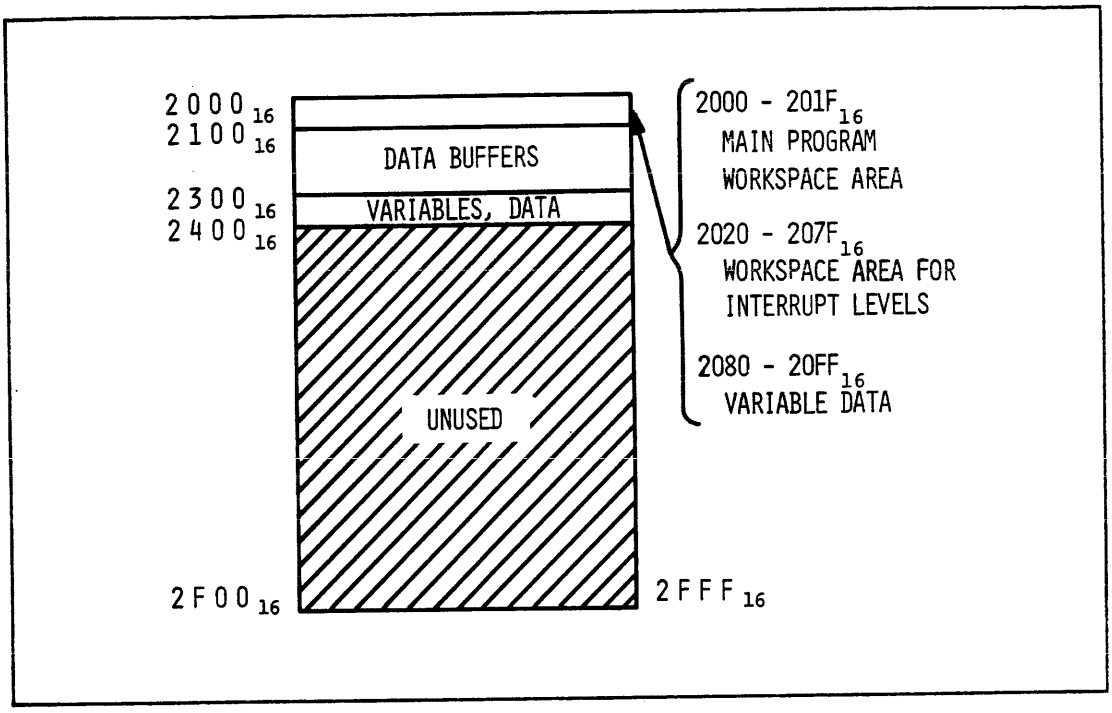


Figure 9-17. RAM Memory Space for a Typical Application

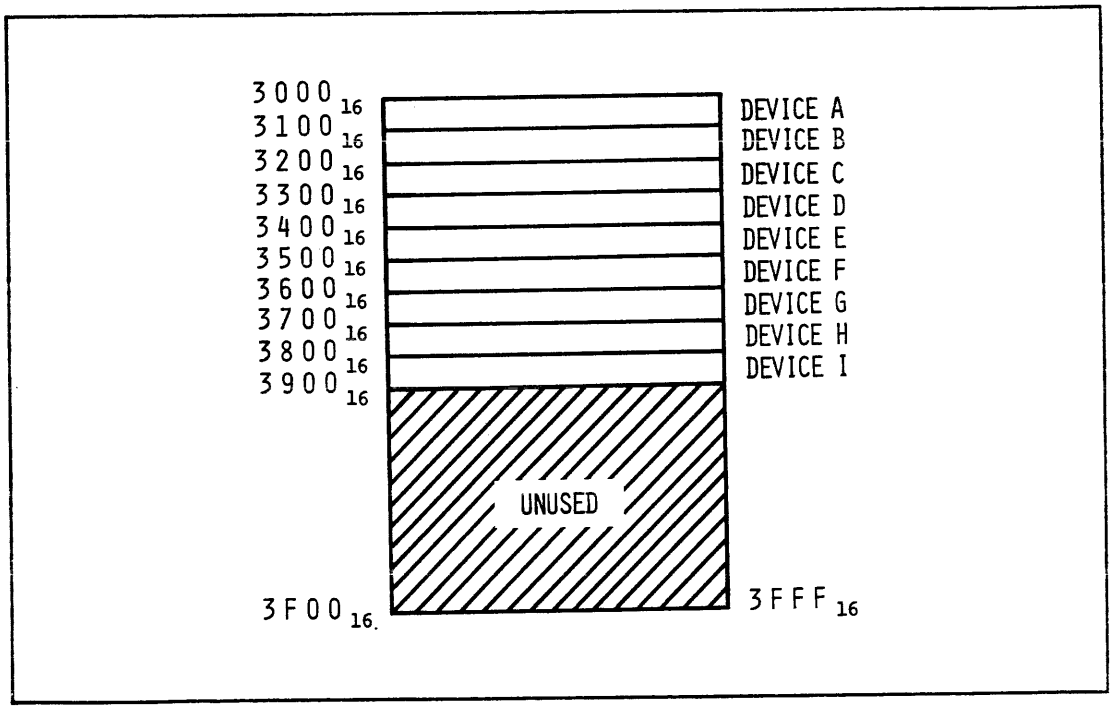
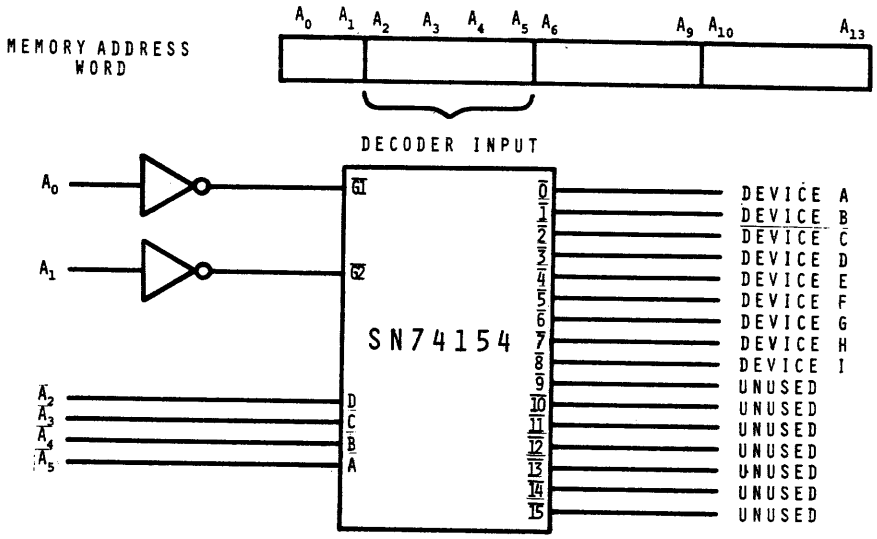


Figure 9-18. Memory Mapped I/O Area for a Typical Application



FUNCTION TABLE

INPUTS						OUTPUTS														
G1	G2	D	C	B	A	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	L	L	L	L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	L	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	L	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	L	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H
L	L	L	H	H	L	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	H	L	L	L	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	H	L	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	H	H	L	L	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	X	X	X	X	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

H = high level, L = low level, X = irrelevant

Figure 9-19. Address Line Decoding for Selecting Individual I/O Devices

Of course, with the TMS 9980A, memory space does not have to be allocated for I/O. This is necessary only if the system designer elects to use memory-mapped I/O, such as in this device decoding example.

Program Module Compaction

Once the total available memory space has been allocated among ROM, RAM, and I/O, and the available space within the ROM area and within the RAM space has been allocated for specific program modules or data, the system designer then fits the program modules or data into the defined area and resolves the intermodule linkage.

The problem of fitting the program modules into the defined memory space is relatively simple. Referring to Table 9-1, module A begins at location 0100_{16} (using an AORG 100 statement), and module B begins at location 0300_{16} (using an AORG 300 statement). Each of the program modules begins at its assigned location.

The ROM memory map in Figure 9-16 was created by the system designer using his best judgment as to the amount of memory that would be required by each of the modules. This estimate should be made toward the liberal side to assure that adequate space has been allocated for each and all of the modules. (Sometimes, however, even the liberal estimate falls short of the amount of memory a program module actually requires.) After all of the program modules are developed, the designer then knows the specific amount of memory each module requires. At this point, the system designer can redefine the memory map and readjust the origin address of each program in order to compress the total program space into a smaller range of memory. By doing so, the number of memory components required to contain the program can be reduced. This is especially important for a production system where the memory components are a production-cost item.

Suppose that after the seven program modules in the application program have been developed, they require the amount of memory shown in the length columns of Table 9-1. The length is determined by the difference between the starting address and ending address (ending address minus starting address plus 2).

Notice that all of the program modules fit into their originally assigned memory space except for module E. As it now stands, enough ROM components are necessary to accommodate the last address used by the last module (module G). If these ROM components are 1K x 8-bit components (such as TMS 2708's or TMS 4700's), seven such components are required. If 2K x 8-bit components are used (such as TMS 2716's) four such devices are required. If 4K x 8-bit components are used (such as TMS 4532's), two such devices are required. By compressing the program modules closer together, all of the modules can be made to fit into a smaller range of memory space, requiring a smaller number of ROM memory components.

Table 9-1. Program Module Address Ranges After Development

Module	Beginning Address	Last Address	Length (Hex bytes)	Length (Dec. bytes)
A	0100_{16}	0264_{16}	166_{16}	358_{10}
B	0300_{16}	$064E_{16}$	350_{16}	848_{10}
C	0800_{16}	$09FC_{16}$	$1FE_{16}$	510_{10}
D	$0B00_{16}$	$111A_{16}$	$61C_{16}$	1564_{16}
E	1300_{16}	1528_{16}	$22A_{16}$	554_{10}
F	1500_{16}	$17AE_{16}$	$2B0_{16}$	688_{10}
G	1300_{16}	$1AD0_{16}$	$2D2_{16}$	722_{10}
		Total Length:	$147C_{16}$	5244_{10}

All of the seven modules could be placed immediately one after the other with no intervening and unused memory space. In this case, the total memory space required ranges from address 0100_{16} through $157A_{16}$ (a total of $147C_{16}$ bytes). As a result, the required number of 1K x 8 ROM memory devices is reduced from seven to six, and the number of 2K x 8 devices required is reduced from four to three.

Some degree of slack between the modules might be desired, however. For example, if the product is to be tested in prototype form (limited production) before volume production, it could be anticipated that the individual modules might need to be expanded. In this case, the number of prototype read-only memory devices (most likely PROM's or EPROM's) could still be reduced, but enough intervening memory space left for a reasonable amount of module expansion.

Assume that the designer decides to pull out some of the slack but wants to leave some additional memory space at the end of each module for possible module expansion. A reasonable amount of expansion might be ten percent.

Table 9-2 shows the address ranges for the program modules with at least ten percent of unused memory following each module. (After the ten percent expansion was calculated and added to the ending address, the designer rounded the ending address value up to the next 16-byte address boundary as the origin point for the next module.)

Table 9-2. Program Module Address Ranges With Ten Percent "Slack" Between Modules

Module	Beginning Address	Ending Address (Current program length)	Ending Address (with 10% slack)	Rounded Ending Address
A	0100_{16}	0264_{16}	0288_{16}	$028E_{16}$
B	0290_{16}	$05DE_{16}$	0634_{16}	$063E_{16}$
C	0640_{16}	$083C_{16}$	0870_{16}	$087E_{16}$
D	0880_{16}	$0E9A_{16}$	$0F3A_{16}$	$0F3E_{16}$
E	$0F40_{16}$	1168_{16}	$11A0_{16}$	$11AE_{16}$
F	$11B0_{16}$	$145E_{16}$	$14A4_{16}$	$14AE_{16}$
G	$14B0_{16}$	1780_{16}	$17C8_{16}$	$17CE_{16}$

Even with the ten percent margin, the number of 1K x 8 or 2K x 8 read-only memory devices is the same as if the modules had been structured with no intervening memory space.

Although the individual program modules will ultimately reside in read-only memory, they are normally placed in RAM for testing purposes as they are developed and then converted to ROM after they are debugged.

The same considerations given to reducing the amount of memory space required by instructions and data constants in ROM can also be given to reducing the variable data-storage area in RAM.

Intermodule Communication

A problem the system designer faces in allocating memory space for the program modules, data constants, and variable data is intermodule communication. Each program module has control passed to it from other program modules. Each program module may contain subroutines that are called from other modules. The program module that passes control to another module or that calls a subroutine in another module must know the address of the module or the address of the subroutine to which control is passed.

One method of resolving this intermodule communication is to place all needed addresses in each program module. As an example,

suppose module B needs to call a subroutine in module D and also needs to call module A. Figure 9-20 shows the relevant portions of the program in module B.

Similarly, "global" data constants in ROM or data variables in RAM that are accessed by more than one program module must have their addresses defined to the program modules that access these "system-wide" constants or variables. For system-wide use, all global constants or addresses can be placed in a module by themselves.

Resolving Label Addresses with the TM 990/189 Symbolic Assembler. With the symbolic assembler on the TM 990/189, some of this intermodule communication and linkage can be resolved as the program modules are assembled. As one program module is assembled, a symbol table is created, and the addresses of all resolved labels defined in the module are recorded in this table. (See Chapter 4 for a discussion of this symbol table.) When a second (or subsequent) program module is assembled, the B option entry to the assembler allows the existing symbol table to be maintained. If the second (or subsequent) program module references a label defined in a previously assembled program module, the address is resolved automatically.

It is often necessary to pass data back and forth between different program modules or subroutines located in various program modules. The previous chapter described some of the most common techniques for data passing.

As a program is being developed, it is often convenient to save the object program in nonvolatile memory storage so that it can be reloaded when it is needed later.

The TM 990/189 includes an optional cassette interface which can be used to record an object program on a cassette tape. The object code is recorded on the tape along with other administrative information in a format that allows the data to be interpreted correctly when it is loaded later. This format is called an "absolute" object-code format because the memory addresses into which the data will be placed are absolute addresses; that is, the addresses are all specific and defined. Upon reloading, each byte of data will be loaded into the same specific, absolute memory location.

Relocatable Assemblers and Relocating Loaders. More elaborate (and usually more expensive) microprocessor development systems sometimes include "relocating" object-code loaders. The relocating loader works in conjunction with a relocatable assembler. The relocatable assembler gives the software designer the flexibility to wait until an object program is loaded into memory to assign the absolute address where the object program will begin. The relocatable assembler produces a specially formatted object program which carries information used by a relocating loader to load the program. It takes both an assembler capable of producing relocatable object code together with a relocating loader to take advantage of this flexibility.

```
        .
        .
        .
AA      DATA    >0100       POINTER TO MODULE A
SA      DATA    >0CA6       POINTER TO SUBROUTINE IN MODULE D
        .
        .
        .
```

```
        MOV     @SA,R10     GET ADDRESS OF SUBROUTINE IN R10
        BL      *R10        CALL SUBROUTINE IN MODULE D
                            (RETURN HERE FROM SUBROUTINE)
```

```
        MOV     @AA,R10     GET ADDRESS OF MODULE A IN R10
        B       *R10        GO TO MODULE D
        END
```

Figure 9-20. Example of Intermodule Communication

With these software utilities, the software designer can assemble the individual program modules in the system without usually being concerned about absolute address origins. It is only as the relocatable object programs are loaded by the relocating loader that the absolute addresses are defined according to where the object program begins in memory. Because the programs are relocatable, they can easily be loaded one right after another with no intervening memory space.

Most relocating loaders are relocating "linking" loaders; that is, they allow labels defined in one program module to be referenced from another program module. Both forward and backward references are allowed.

A relocating loader and a relocatable assembler are available with the TI AMPL software development system, which is described in more detail later in this chapter.

9.5 INTERRUPT SERVICING

As a means of review, recall that there are three general methods of I/O available with microprocessors: program-controlled I/O, direct memory access (DMA), and interrupt-driven I/O. Interrupt-driven I/O is unique among these three in that it allows an I/O device to initiate interaction with the processor. Interrupts tell a processor that immediate action is to be taken, so immediate that the processor is asked to suspend temporarily what it is currently doing in order to respond to the interrupt. Also recall that the 9900 family of microprocessors can accommodate prioritized, vectored interrupts. Specifically, the TMS 9980A on the TM 990/189 will accept six levels of prioritized, vectored interrupts.

As a further review, recall that each interrupt level has a two-word vector associated with it. Each vector is at a specific location in memory and is used by the processor to respond to the interrupt. In each two-word vector, the first word contains the address of the set of working registers to be used by the interrupt-service routine, and the second word contains the starting address of the interrupt-service routine.

The processor responds to an interrupt by performing a context switch. Control is taken away from the current program with its set of working registers and given to the interrupt-service routine with its own set of working registers. As part of the context switch, a return to the interrupted program's environment is made possible by storing the current workspace pointer, program counter, and status register (that is, the contents of these three internal registers before the context switch) in registers 13, 14, and 15 of the interrupt-service routine's set of working registers.

Interrupt Service Routines

In general, an interrupt-service routine performs the following duties.

- Saves the interrupted program's environment
- Identifies the device requiring service
- Processes the interrupt
- Resets the interrupt
- Returns control to the interrupted program.

Each of these functions is described in order.

Saving the Interrupted Program's Environment. First, an interrupt-service routine usually receives control after control has been taken away from another program by the processor. (A possible exception to this is a reset interrupt created as part of a system's initialization sequence such as when power is first applied to the system.) In many systems, the microprocessor saves the contents of the program counter (containing the address of the next instruction in the interrupted program). It is then the responsibility of the interrupt-service routine to save the other vital pieces of information so that the interrupted program's environment can be restored later. For example, it is normally necessary to save the status register and the contents of any of the working registers that will be used by the interrupt-service routine (assuming there is only one set of working registers available).

With the 9900 family of microprocessors, the interrupted program's environment is automatically saved by the processor. The processor saves the contents of the program counter and the status register, and it is not necessary to save the contents of the working registers specificially. The processor automatically switches to another set of 16 working registers to be used by the interrupt-service routine. The processor saves only the contents of the workspace pointer used by the interrupted program. With the 9900 microprocessor family, the task of saving the program environment is performed by the processor itself and does not have to be done by the interrupt-service routine.

Identifying the Device Requiring Service. The second task of the interrupt-service routine is to identify the interrupting device. Some processors have only one interrupt level (or sometimes one interrupt used for initialization and one other common interrupt which is shared by all interrupt-generating devices). With these processors, if several devices in the system are capable of generating an interrupt, the interrupt-service routine must determine which specific device caused it before it can be processed. A common technique in this case is for the interrupt-service routine to send out a special inquiry command to all devices capable of generating an interrupt, and the device which generated the interrupt

responds by sending back an identifying code. From the returned code, the interrupt-service routine identifies the interrupting device.

With the 9900 microprocessor family, however, an interrupt-generating device is assigned a unique code which is presented to the processor whenever that device makes an interrupt request. From this code, the processor can determine the identity of the interrupting device and transfer control to the specific interrupt-service routine assigned to that device. Therefore, the interrupt-service routine can immediately start processing the interrupt. It knows which device caused the interrupt because it received control. Each device has a unique interrupt vector. These vectors are maintained in the lower memory addresses, and the particular vector used depends upon the interrupt code presented to the microprocessor. Each code has a corresponding interrupt level.

With the TMS 9980A microprocessor, specifically, there are six interrupt levels (RESET, LOAD, and user levels 1, 2, 3, and 4). Normally, RESET is used for power-up and initialization, leaving five interrupt levels for interrupt-generating devices; therefore, as many as five devices can each be assigned a unique interrupt code.

If a system is designed with more interrupt-generating devices than there are interrupt levels, more than one device would have to be assigned to an interrupt level. In this case, the interrupt-service routine for that level would then have to determine which device caused the interrupt. This might be accomplished by an inquiry command to all the devices on that level which would request a further identifying code.

Processing the Interrupt. After it has identified the interrupting device, the interrupt-service routine must process the interrupt. This could be as simple as recording the fact that an interrupt occurred. For example, suppose that a real-time clock is used to measure elapsed time by generating an interrupt at periodic intervals, such as every ten milliseconds. The interrupt-service routine can simply record the fact that an interrupt occurred by incrementing the contents of a counter used to record the number of accumulated "clock ticks." This counter could then be accessed by other system routines to measure elapsed time. By storing the number of ticks in the counter when one significant event occurs and then subtracting this value from the number of ticks in the counter when a subsequent significant event occurs, a routine can determine the time period between the two events.

Another routine might use the clock-tick counter to keep track of the time of day (and the date also).

In a real-time operating system, the clock-tick counter can be used to determine times to transfer control from one task in the system to another. (More is said about real-time operating systems later in this chapter.)

Often, however, the interrupt-service routine must take further action. For example, an interrupt-service routine that sounds an alarm to indicate that a motor is overheating must also take action to shut down the motor or, perhaps, slow it down.

Often the interrupt-service routine must exchange, receive, or transmit data to or from a device. For example, a card reader can generate an interrupt indicating that a punched card has been read and that it is now ready to send the data it read from the card. The interrupt-service routine processes the interrupt by accepting the data and storing it into a buffer for processing by another routine.

An interrupt can be used to record that a data transfer has been completed. For example, a DMA controller can transfer information from a high-speed disk directly into memory. After the data transfer, the controller generates an interrupt to indicate the data transfer has been completed. The interrupt-service routine might then raise a flag to indicate to another routine that the data is in memory and ready to be processed.

Resetting the Interrupt. Normally, the interrupt-service routine must reset the interrupt; that is, it must issue a control signal which causes the interrupt request to be removed, allowing the device to issue another interrupt request later.

Returning Control to the Interrupted Program. As its final duty, the interrupt-service routine usually returns control to the interrupted program. This requires restoring the previously saved contents of the working registers, the saved contents of the status register, and the saved contents of the program counter.

Most microprocessors having interrupts also have a special instruction in the instruction set for returning control from an interrupt-service routine. The RTWP instruction serves this purpose with the 9900 microprocessor family. The RTWP instruction automatically returns control to the interrupted program and restores its environment.

Interrupt Priorities and Response Time

The structuring of an interrupt-service routine requires some additional considerations beyond that of other routines. For example, in a multiple-interrupt environment, it is possible that an interrupt-service routine can itself be interrupted. This "nested" interrupt situation can produce additional design considerations.

Nested Interrupts. With the TMS 9980A, an interrupt request is allowed to interrupt an interrupt-service routine (or, for that matter, any routine) only if the request has a sufficient priority.

For example, if a level-three interrupt context switch is made, the low-order four bits of the status register are set to the value two, indicating that only an interrupt request priority higher than three is to be permitted.

Should a later interrupt of sufficient priority occur (for example, a level-two interrupt while the level-three interrupt-service routine is executing), the processor responds automatically with a nested context switch to the level-two interrupt-service routine and the low-order four bits of the status register are set to the value one, indicating that only an interrupt of level one or higher will be permitted.

When the level-two routine is finished, an RTWP instruction reverses the context switch back to the interrupted level-three routine. When the level-three routine is completed, its RTWP instruction returns control to the program interrupted by the level-three interrupt.

Restructuring the Priority Levels. It is possible to restructure the current permissible interrupt level by altering the contents of the status register. For example, an LIMI 1 instruction in a level-three interrupt-service routine would prevent a level-two interrupt request from interrupting the level-three routine. The LIMI 1 instruction should be the first instruction in the level-three interrupt-service routine. After a context switch is made by the processor as a result of an interrupt, all interrupt requests are held off (not permitted) until after execution of the first instruction in the interrupt-service routine. This allows the interrupt-service routine to rearrange the priority structure if necessary.

Likewise, an interrupt-service routine can allow itself to be interrupted by what would normally be a lower priority interrupt level. In this case, the LIMI instruction can be placed anywhere in the interrupt-service routine where it is desired.

In some particular section of a program it may be necessary to hold off any interrupts during the execution of a set of instructions, perhaps because the program section is especially time sensitive (it must be executed within a specific time limit) or, perhaps, the program is non-reentrant (it cannot be interrupted without the risk of pertinent data being lost). In such a case, a STST instruction could be used to remember the current permissible interrupt level, followed by a LIMI 0 to turn off all interrupts. (RESET, however, cannot be prevented with the TMS 9980A. That is why RESET is normally reserved for only a very critical interrupt such as power-up, or initialization, for example, when it no longer becomes necessary to follow the normal priority structures.)

After the interrupt-sensitive section of the program is completed, the saved interrupt level can be restored.

Interrupt Response Time. One other subject should be addressed in relation to interrupt-service routines, and that is interrupt response time.

Interrupts are normally assigned to devices that require immediate attention so, as a general rule, an interrupt-service routine should respond to an interrupt as soon as possible.

Several factors affect the speed with which the response to an interrupt is made:

- The relative priority of the interrupt request
- The structure of the interrupt-service routine
- The number of machine cycles required to complete the current instruction being executed when the interrupt request is made
- The speed at which the processor is running.

First, when an interrupt request is made by a device to a TMS 9980A, it may not be permitted to interrupt because its interrupt level is not of sufficient priority, In this case, the interrupt is held off until the interrupt mask in the status register is changed to allow the interrupt request.

Secondly, even when an interrupt level is of sufficient priority to be permitted, the response time to the interrupt is dependent upon the structure of the interrupt-service routine. The routine may have other administrative things to do before it can get around to processing the interrupt.

Also, the response time is affected by the point during the execution of an instruction that the request becomes active. With most microprocessors, including the TMS 9980A, an interrupt request is not acknowledged until the execution of an instruction is completed. The earlier in an instruction that the request initially becomes active, the longer it must wait until the processor can acknowledge it. This wait time is also affected by the instruction being executed. It may be a relatively short instruction such as JMP, or a relatively long instruction like DIV.

Finally, the speed at which the microprocessor is running affects the overall response time.

Generally, the amount of time spent in an interrupt-service routine should be as small as possible. While in an interrupt-service routine, it is possible that other devices (at that same level or at another level) may have an interrupt request active which is being held off, waiting for the current interrupt-service routine to complete its processing.

Only the minimum amount of processing required should be performed in the interrupt-service routine itself and as much processing as possible allocated to other, noninterrupt-service routines.

9.6 REAL-TIME CONSIDERATIONS

The use of interrupts is often associated with "real-time" applications. A real-time system processes data as it becomes available, as opposed to a "batch" system which processes data at some later, unrelated time after the data is produced. A real-time application allows processing and decisions to be made upon incoming data in time to control the process under operation.

As an example, Figure 9-21 depicts a microprocessor system used in a factory to control a conveyor belt operation. A motor drives the conveyor belt, and the speed of the belt is determined by the amount of current supplied to the motor and the load on the belt. The system is designed to maintain the movement of the conveyor belt at a constant speed selected by an operator. It must also detect and display the current speed of the belt to the operator. In addition, if the motor becomes overheated, the system must alert the operator, and take immediate action to stop the motor. Also, it must record the number of items carried on the belt and produce a report upon demand that shows the number of items carried on the belt during each 15-minute interval of the day.

Controlling the speed of the belt is a real-time process. By detecting precisely spaced marks on the belt (using a light-sensitive detector, for example) and by employing a real-time clock to produce evenly spaced clock ticks, the processor can determine the speed of the belt and can maintain the selected speed by controlling the amount of current fed to the motor. Notice that if the processor receives information indicating the belt is going slower than its set rate, it must respond to speed up the belt. In addition, the clock ticks that are used to determine the speed of the belt are a series of interrupts from a real-time hardware clock. An overheated motor can indicate an alarm condition through an interrupt, or the condition could be detected by the processor periodically (say, every ten seconds) reading the temperature of the motor and turning it off if it becomes too hot. This latter method of detection is called polling.

The real-time clock (RTC) could be used not only to provide a time base for measuring the speed of the belt but also to record the passing of a ten-second interval to determine when it is time to read the motor temperature. Furthermore, the real-time clock can serve as a base for keeping the time of day. In conjunction with the time of day, the processor can count the number of items on the belt (with another light-sensitive detector perhaps) during each 15-minute period of the day. And, as a final function, the processor must recognize an operator input from the keyboard and respond in real-time to the operator request.

Time Measurement and Delays

The measurement of elapsed time often plays a key role in a real-time application. There are, essentially, two ways that

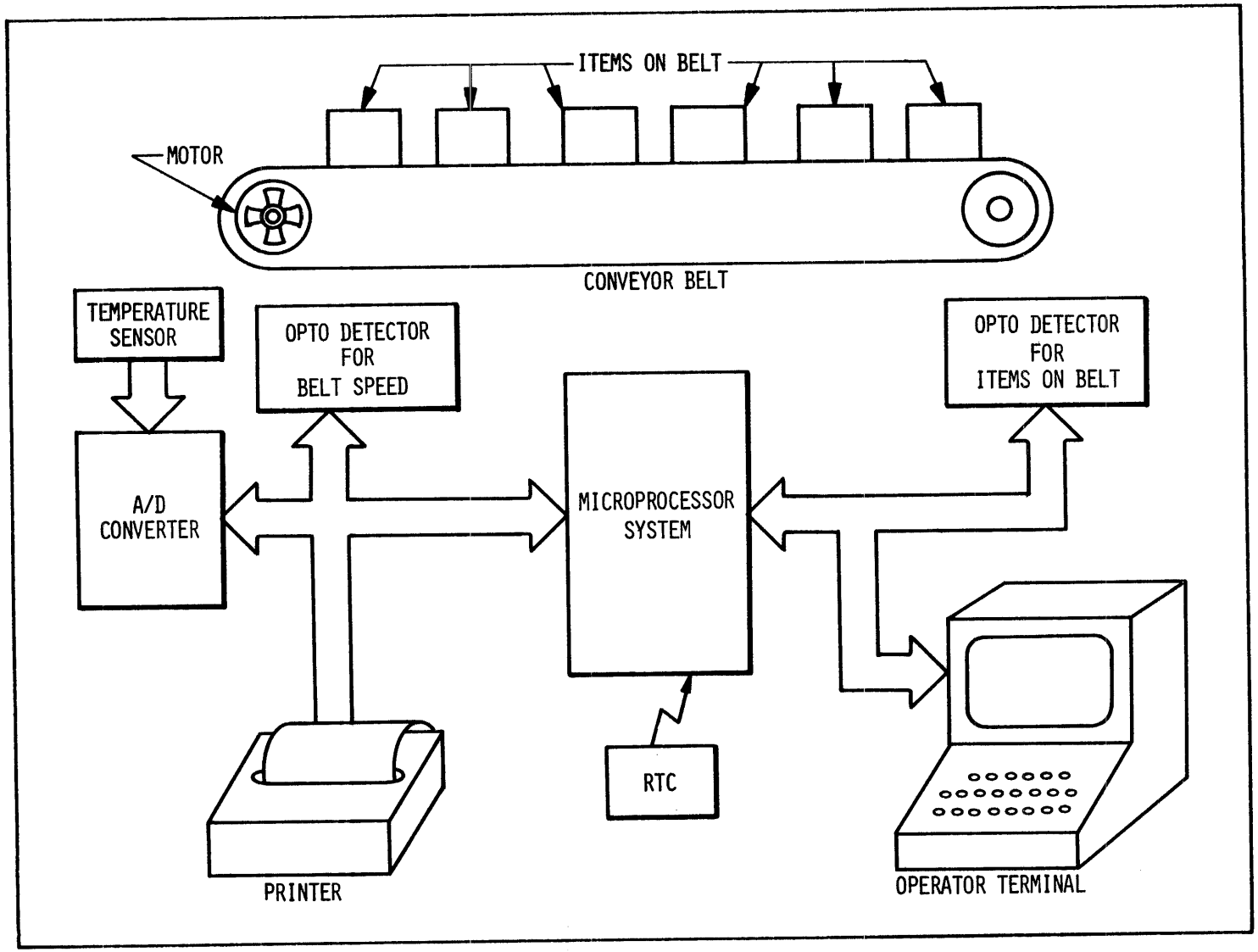


Figure 9-21. A Real-Time Microprocessor System in a Factory

elapsed times can be measured or time delays produced in a system: by program-controlled timing loops and by a hardware clock.

Program-Controlled Timing Loop. A program-controlled timing loop is a method of generating a time delay by executing a carefully selected group of instructions the collective execution time of which is calculated to require a given amount of execution time. An example of a program-controlled timing loop is the cycle generator program in Chapter 6.

There are two primary advantages of program-controlled timing loops when compared to a hardware clock. First, no additional hardware is required. Second, this is a relatively quick way of developing a simple time delay. For example, if a one-millisecond delay is to be introduced into a given process, a sequence of instructions could be structured very quickly to generate this.

However, there are several disadvantages associated with program-controlled timing loops.

First, it is more difficult to program highly accurate times. The instructions used must be very carefully selected and structured very carefully. This can take a lot of time to develop. The resulting time delay may not be highly accurate. For example, a 20-microsecond delay might be required, but with the instructions available, it may be possible to produce only a 19- or 21-microsecond delay at best; a difference of ± 5 percent from the desired time.

Second, program-controlled timing loops generally do not allow a large variation in time delays. By introducing a flexible time delay into a program-controlled timing loop, decision-making instructions are also introduced which tend to disrupt the carefully structured timing of the sequence of instructions.

Third, and this is usually the most serious disadvantage in a system, program-controlled timing performs only one function: a delay. No other processing is being accomplished, and an active interrupt would destroy the integrity of the timing loop. It is possible that a relatively large time delay could be combined with some processing operation, but, practically speaking, this is rarely possible.

Also, a real-time clock is not possible since the processor is totally dedicated to timing loops. Such a clock needs to be external, driving the microprocessor through interrupts.

Hardware Clock. The other method, using a hardware clock, overcomes most of the disadvantages of the program-controlled timing loop.

First, a hardware clock can be built to produce a very accurate time base. The hardware clock can be designed to produce interrupts

at precisely defined time intervals independent of the execution time of the instructions.

Second, the individual clock-tick interrupts can be combined by the software to provide a number of widely varying time delays with a precision equal to the inverse of the clock frequency.

Third, with a hardware clock, the software can perform useful work in between clock ticks. The software is not locked in a loop simply waiting for time to expire. It can be performing a processing function with the assurance that it will be notified whenever the elapsed time has expired. Furthermore, other interrupts can be allowed, and the programmer can designate them to be of either higher or lower priority than the clock interrupt.

The hardware clock, however, does have some disadvantages. First, additional hardware is required to implement the clock and, second, the accuracy of the clock is limited by the interrupt response delays described in the previous section.

There are several methods of building a hardware clock, often called a real-time clock (or RTC). For example, the time base can be established around the commonly available 60-Hz power line. Figure 9-22 shows how an interrupt can be built to produce an interrupt based upon the positive-to-negative transition sine wave. Notice that this method will cause an interrupt to be generated, on the average, every 1/60 of a second or every 16.67 milliseconds. The frequency of the line varies slightly, but the average frequency over a period of time will be very close to 60 Hz. The average frequency is normally sufficient to maintain the accuracy of a time-of-day clock.

A second method, illustrated in Figure 9-23 and 9-24, uses discrete components to generate an RTC of a given frequency. In Figure 9-23, the time base is derived from a variable RC network. In figure 9-24, the time base is established by a crystal for more precision.

A third method is to use with the microprocessor an LSI peripheral component that includes a real-time clock. For example, the TMS 9901 on the TM 990/189 includes an interval timer the frequency of which can be programmed from software. (For details of the TMS 9901 operation, refer to Chapter 7.)

The Real-Time Clock (RTC)

Once an RTC has been added to a system, it can be used for a number of purposes. For example, the occurrence of the individual clock ticks can be recorded so that there is a running measurement of elapsed time. In addition, the clock ticks can be accumulated and used to keep track of the time of day and date.

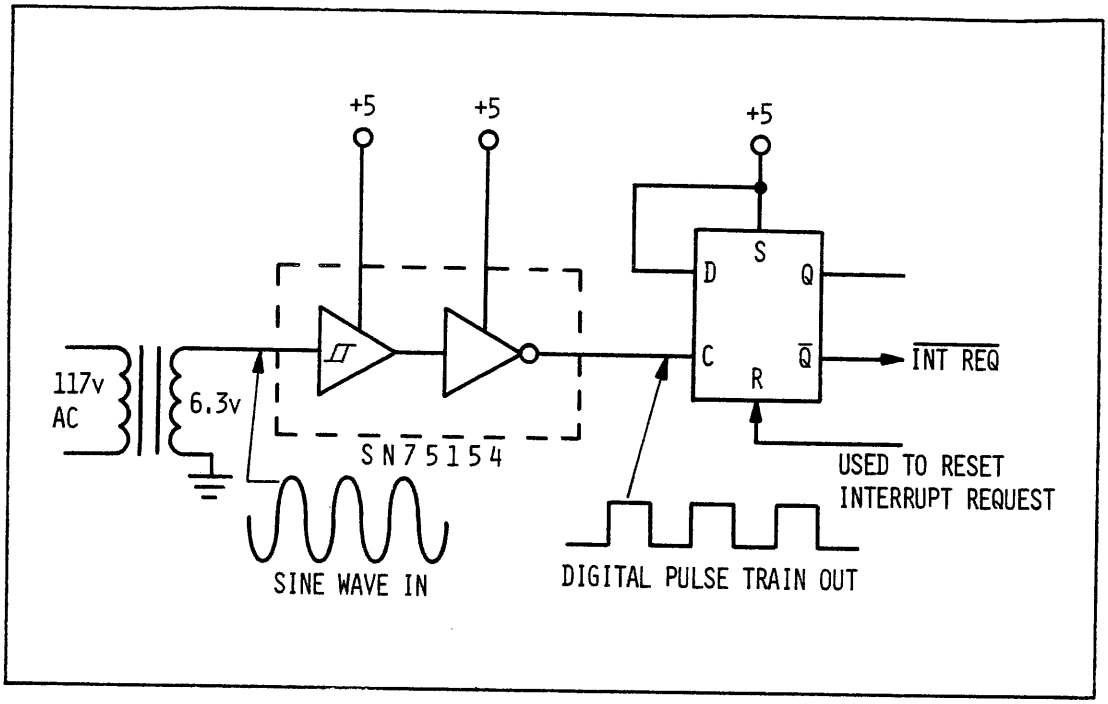


Figure 9-22. An RTC Based Upon the AC Power Line

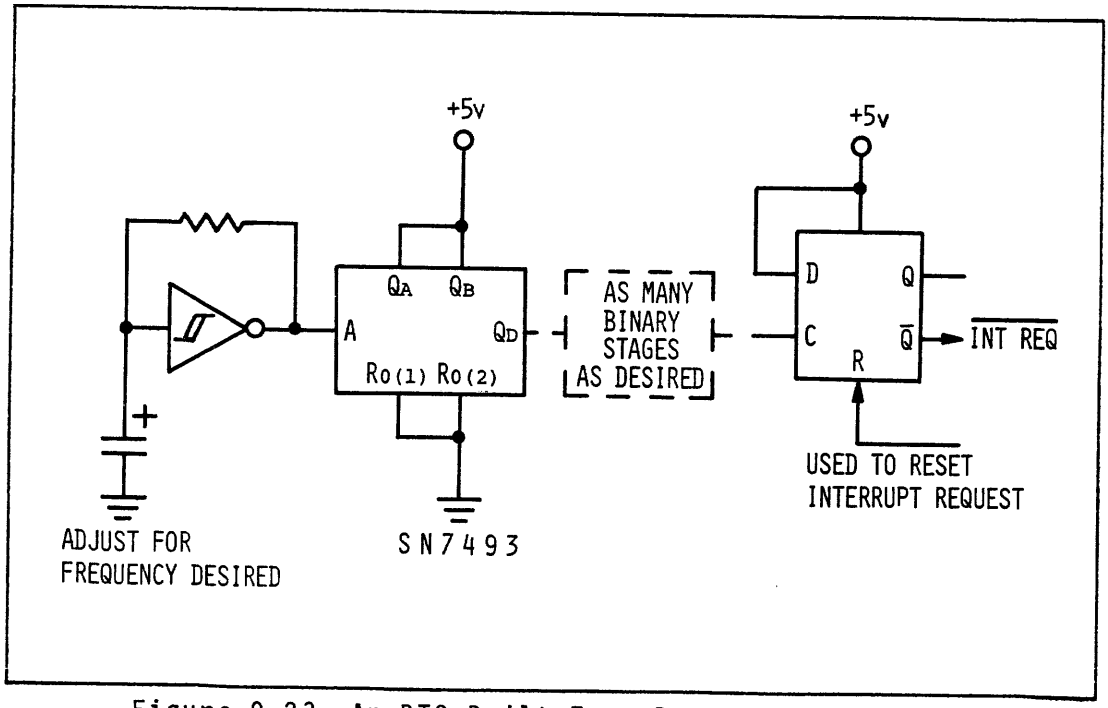


Figure 9-23. An RTC Built From Descrete Components
With a Time Base Set By An RC Network

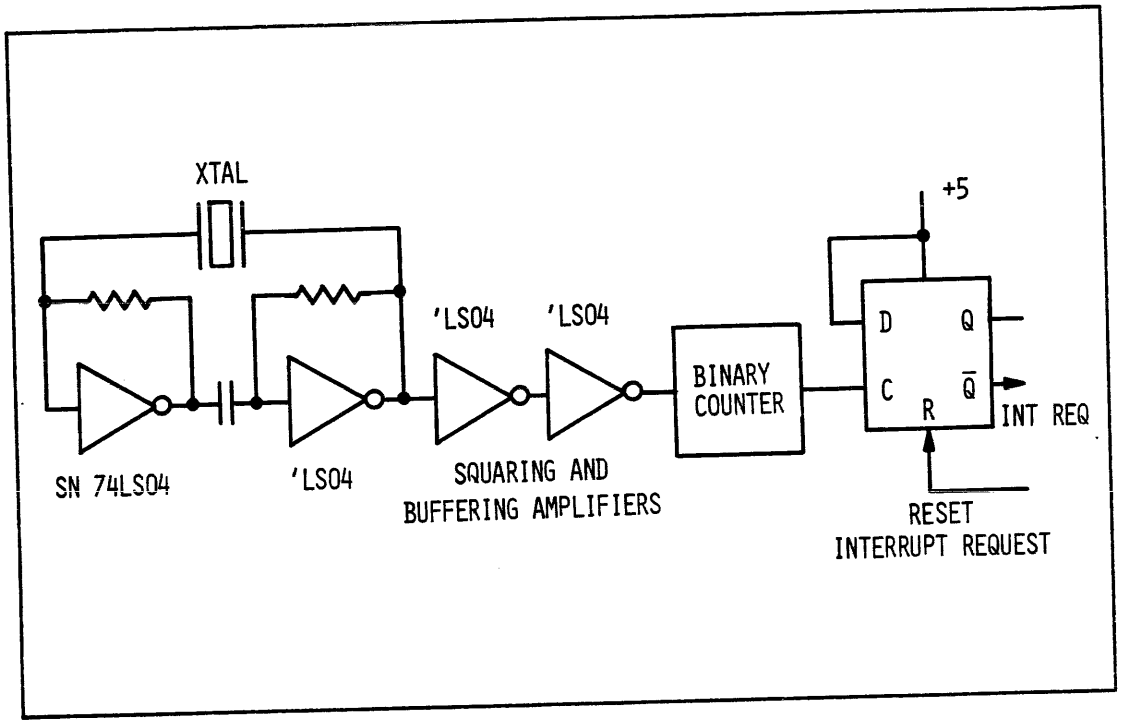


Figure 9-24. An RTC With a Crystal-Controlled Time Base

Some consideration needs to be given to the structure of the interrupt-service routine that processes the RTC and to the choice of frequencies chosen for the RTC.

As with most interrupt-service routines, as little time as practical should be spent in its execution. As an example, if a time-of-day clock is used in a system, the interrupt-service routine may simply need to record the fact that a clock tick occurs. The updating of the clock can be performed once the interrupt has been dismissed.

Consideration must also be given to balancing the accuracy of the RTC and the amount of interrupt overhead introduced. If the RTC frequency chosen is too slow, there may not be the precision needed. If the frequency is too fast, the system throughput may be slowed down because of too many interrupts. It is important to note at this point that the TMS 9901 timer, after making an interrupt request, automatically reloads itself with the established timer value and begins counting down the value to generate the next interrupt. It is not necessary for the interrupt-service routine to reload the TMS 9901 with the timer value once the value is initially established.

Real-Time Operating Systems

The factory system described earlier in this section includes a multitude of functions: control, alarm monitoring, operator interface, report generation, and time-of-day keeping. To manage all of these functions might require a real-time operating system.

Definition and Usages. A real-time operating system is a supervisor program operating in a real-time mode. A supervisor is responsible for coordinating the various functions in a system. The various programs in the application are given control of the processor one at a time by the supervisor. The use of the processor in a real-time application may be on a "time-slice" basis; that is, one program function gets control of the processor for a given length of time and then another program function is given control for another period of time. The time periods are usually measured by a real-time clock. In addition, many real-time operating systems also keep track of the time of day (again, based upon the real-time clock) and may schedule various program functions according to the time of day.

Functions. Three functions usually performed by a real-time operating systems include

- Task management
- I/O management
- Time management.

Task management includes several specific functions. As previously mentioned, there may be several tasks (program functions) in an application which must share use of the processor. (The processor can execute only one program at a time.) There must be a supervisory program to coordinate the execution of these tasks. The real-time operating system does this. All programs can be assigned a fixed length of time with which they have the processor and each one is given control on a "round robin" basis. For example, a system may have five program functions, each of which is given control for 100 milliseconds. Although each program function has the processor for only 20 percent of the time, the processor is so fast (by human standards) that it may appear the processor is performing all five functions simultaneously.

Some degree of priority can be added to this time slicing. Perhaps one program function is deemed to be more important than another. In this case, the program function could be assigned a longer period of time than the others to have control of the processor. Or, perhaps, its time slice could be the same length as the others, but it receives a time slice every other time, as shown in Figure 9-25. Now, the higher priority task has control of the processor for half of the total time and the other half of the time is shared by the other four program functions.

A more complex priority structure could be arranged where relative priorities are established among all five of the program functions. The relative priorities might even change dynamically. For instance, they might change as a result of the time of day or based upon external conditions that occur.

The processor, which is shared by the various tasks, is a system resource. There are usually other system resources. As an example, there may be a single printer in a system, and several of the tasks may be capable of writing information to the printer. If a situation develops in the system where more than one task wishes to use the printer, the operating system must resolve this resource contention and assign the use of the printer to each task in turn. Allocation of system resources is a part of the specific task-management functions of a real-time operating system.

A real-time application is often a dynamic environment where each program function (task) is carrying out its assigned job and passing data to or exchanging information with other tasks regarding the status of work completed, error conditions, or other pertinent information. This exchange of information among the various tasks is usually the task-management responsibility of the real-time operating system.

As part of its I/O management duties, the operating system usually assumes the control of the I/O devices. If a task wishes to receive some data from a keyboard, for example, the request for such input is made to the operating system which can analyze the request based upon contention for the resource (another task may also be requesting input from the keyboard). Based upon relative priorities of this request compared to other functions in the system, the operating system can then schedule the operation and alert the requesting task when the operation is completed.

The individual I/O driver routines that service the various I/O devices in the system are also normally modules which are supervised by the operating system.

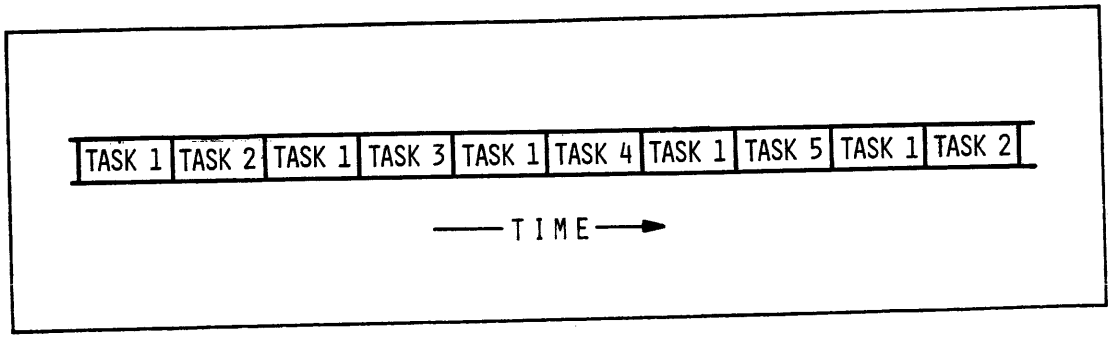


Figure 9-25. A Higher Priority Task Shares Time Slices With Lower Priority Tasks

In an interactive system, that is, a system that includes operator communication via a terminal with the system, the interaction with the operator is handled by a task managed by the operating system.

Finally, time management is a vital part of a real-time operating system. Using the time base provided by the RTC, the operating system is able to provide the time slicing among the individual tasks and can maintain the time of day upon which many functions and task priorities depend.

Because of its many capabilities, a real-time operating system can bring a high degree of flexibility to an application.

Reentrancy

In the factory system discussed earlier, it is possible that the microprocessor could be controlling several conveyor belts, and separate tasks could be responsible for all of the functions associated with each belt (speed determination, motor control, motor temperature monitoring, etc.) As an example, it is likely that there is only one program module in the system which monitors the motor temperature. The same program logic (set of instructions) is used to read the temperature and it is only the data read in from each individual motor that varies.

Concept. In this situation it may be desirable to make the individual program functions reentrant. Reentrancy is the attribute of a program which allows it to be shared by several tasks in an interrupt environment. One task can begin the execution of the reentrant program, be interrupted while control is in the reentrant program, and another task that uses the same program can execute it without the variable data of either program being affected. Figure 9-26 illustrates the concept. The following list of numbered comments refers to the circled numbers on Figure 9-26.

1. Task A is given control by the operating system and begins executing.

2. Task A calls the reentrant subroutine to perform a function.

3. The subroutine executes until an interrupt occurs.

4. The interrupt causes the operating system to transfer control to Task B.

5. Task B begins executing.

6. Task B calls the reentrant subroutine to perform its function.

7. The subroutine is reentered at the top and completes the function.

8. The subroutine returns control to Task B.

9. Task B completes.

10. Control is returned to the reentrant subroutine at the point where the interrupt occurred.

11. The subroutine completes execution.

12. The subroutine returns control to Task A which originally called the subroutine.

13. Task A then completes.

Considerations. There are several key considerations that must be given to the creation of a reentrant program. First, there is no data in the program except those data constants which are used by all tasks. Second, no data and, certainly, no instruction is modified within the program. Third, any data that is not common to all tasks is stored in a data area unique to each task where it can be addressed by a pointer unique to each task. It is usually the duty of the calling task to supply such a pointer to the subroutine.

Reentrancy can be accomplished quite easily in the TMS 9900 family of microprocessors because the common reentrant routine can be called using a different set of registers for each calling task.

Reentrancy Examples. As an example, assume the sequence of instructions in the motor-temperature monitor routine shown in Figure 9-27. The routine is entered at location TM with the CRU software base address of a motor's temperature-sensing device in R12. The routine reads the temperature into memory location LC and then compares the value read with a limit value in memory location LM. If the value read is greater than the limit value, the routine raises an alarm. This program works well if only one program at a time calls the subroutine and if it is allowed to be completed before another program calls it, but it is not reentrant.

Suppose there were two conveyor belts in the system. Task A handles one belt and its associated functions, and Task B handles the other belt and its associated functions. Now consider the following sequence of events.

Task A uses the routine in Figure 9-27 to monitor the temperature of motor A. Task A places the CRU base address of motor A in R12 and calls this subroutine.

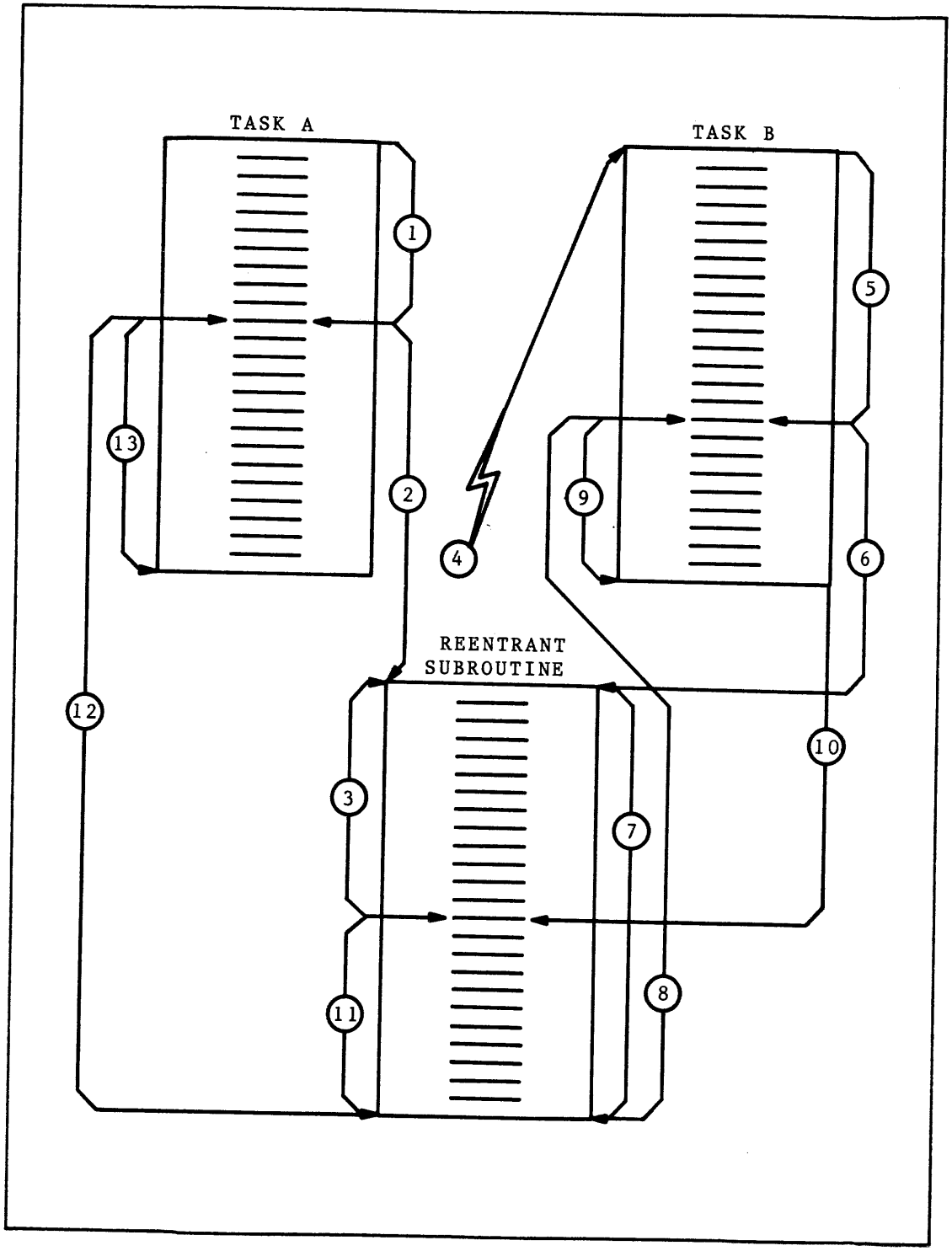
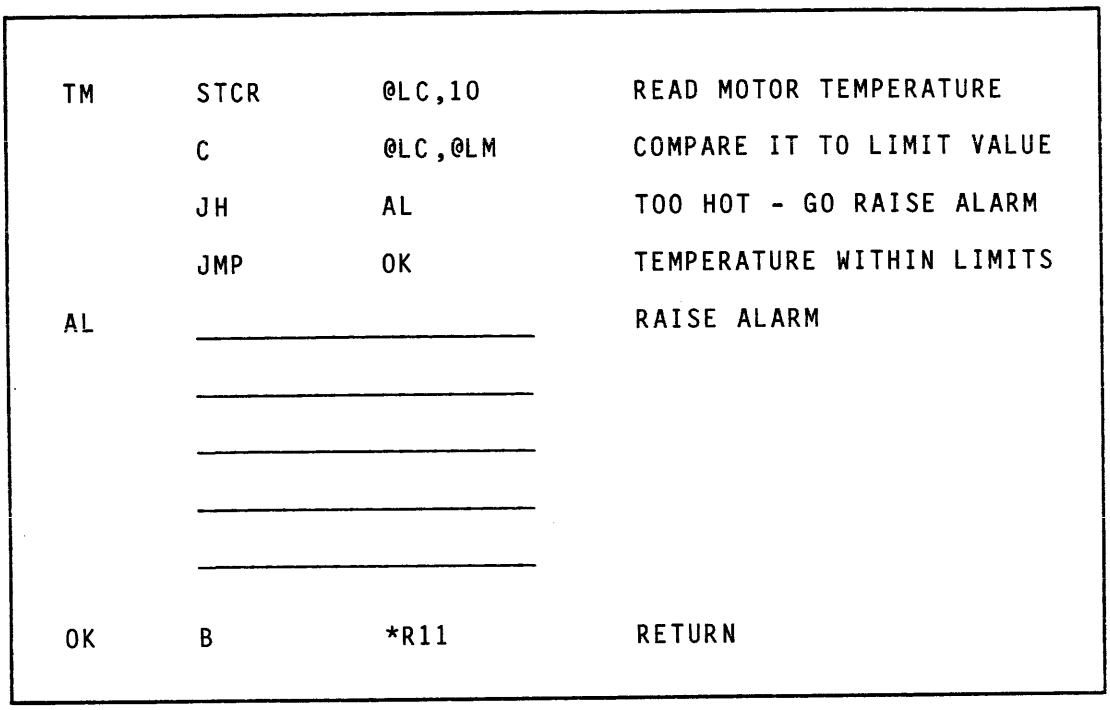


Figure 9-26. The Flow of Control With a Reentrant Subroutine



```
TM      STCR    @LC,10          READ MOTOR TEMPERATURE
        C       @LC,@LM         COMPARE IT TO LIMIT VALUE
        JH      AL              TOO HOT - GO RAISE ALARM
        JMP     OK              TEMPERATURE WITHIN LIMITS
AL      _____________________   RAISE ALARM
        _____________________
        _____________________
        _____________________
        _____________________

OK      B       *R11            RETURN
```

Figure 9-27. Nonreentrant Motor Temperature Monitor Routine

The routine reads motor A's temperature into memory location LC (with the STCR instruction). At this point, an interrupt occurs which causes the operating system to transfer control to Task B.

Task B begins executing, places the CRU base address of motor B in R12, and calls this subroutine to monitor the temperature of motor B.

The routine is reentered at its entry point, and the STCR instruction reads motor's B temperature into memory location LC (which overwrites the temperature read from motor A). Motor B's temperature is compared to the limit value, and a decision is made to raise an alarm or not, and control returns to Task B.

Task B eventually relinquishes control, and control is returned to the subroutine at the point of interruption (just before the C instruction). The subroutine compares the value in memory location LC with the limit value, but the value in location LC is now the temperature of motor B, not motor A. The temperature read from motor A was lost when Task B got control and called the subroutine.

It is possible that the temperature read from motor A was over the limit, but because the subroutine is not reentrant, this alarm condition would not be detected.

This problem can be avoided by making the subroutine reentrant as shown in Figure 9-28.

```
TM      STCR    R0,10       READ MOTOR TEMPERATURE
        C       R0,@LM      COMPARE IT TO LIMIT VALUE
        JH      AL          TOO HOT - GO RAISE ALARM
        JMP     OK          TEMPERATURE WITHIN LIMITS
AL      _____               RAISE ALARM
        _____
        _____
        _____
        _____
OK      B       *R11        RETURN
```

Figure 9-28. The Motor Temperature Monitor Routine Modified to Make It Reentrant

In this modified program, it is assumed that each calling program has its own unique set of register. The temperature is read into the register 0 of the calling program and now, even though interrupts may occur, the temperature read from a motor is preserved in the register.

In this section, several considerations involving real-time applications are discussed, including timing considerations, ways to construct a real-time clock, the functions of a real-time operating system, and the construction of a reentrant subroutine. Many of these concepts and practical considerations are illustrated in the program example which follows.

9.7 PROGRAM EXAMPLE: TIME-OF-DAY CLOCK

Goal of the Program Example

The goal of this program example is to illustrate some of the fundamentals of software engineering discussed in this chapter. These fundamentals include top-down design, structured programming, program modularity, intermodule communication, real-time programming, and the construction of interrupt-service routines. Character conversion techniques are also illustrated.

What the Program Does

The program example maintains the time of day and activates an alarm at a selected time. The time of day is maintained by the program based upon a series of periodic interrupts received from a real-time clock. The program allows an operator to set an alarm time and to initalize the time of day from the on-board terminal of the TM 990/189. Once the alarm time has been selected and the time of day has been initalized by the operator, the program will maintain the time of day and show the current time on the terminal's display. If an alarm time is set by the operator, the speaker on the board will sound for a short period when the alarm time is reached.

Program Definition

The program maintains a 24-hour clock; that is, it records the hours as a series of digits from 0 through 23. The program maintains the time of day and displays only the four digits constituting the current hours and minutes.

The interval timer of the "display" TMS 9901 on the microcomputer board is programmed to generate an interrupt at 200-millisecond intervals. The display 9901 is the device at position U11 on the microcomputer board. Two hundred milliseconds was chosen as the timer value for, primarily, two reasons. First, it is a relatively large interval which reduces the number of interrupts generated by the real-time clock (and, thereby, reduces the interrupt overhead), and, second, 200 milliseconds is a value that produces a nonfractional number for the time command presented to the 9901 for a system running at 2 MHz.

The interrupt from the real-time clock is interpreted by the TMS 9980A microprocessor as a level-four interrupt. (However, this is an internal level 3 at the TMS 9901.) This interrupt level is already predefined on the board for the display 9901 interval timer. The 200-millisecond real-time clock that is used to keep track of the time of day is used in conjunction with another interval timer which is also input as an interrupt to the TMS 9980A. This periodic interrupt (which occurs every millisecond) is necessary to periodically refresh the characters displayed by the on-board terminal.

The interrupt-service routine that services the 200-millisecond real-time clock is designed to spend a minimum amount of time in the routine. The routine simply increments a memory location which records that a clock interrupt has occurred. The updating of the time-of-day clock as well as the comparison of the time of day with the alarm time is performed outside of the interrupt-service routine.

The program uses the XOP utilities available with the TM 990/189 monitor to interface with the operator as the alarm time and time of day are initialized. A user XOP utility is also used to display

the current time of day once the time has been set. The characters that are entered by the operator to select the alarm time and to select the time of day are converted into ASCII characters. This is done to somewhat simplify the displaying of the time-of-day characters but, also, to illustrate character-conversion techniques. When an alarm time is selected by the operator, the speaker on the board sounds the alarm. The cycle generator program example in Chapter 6 is used to drive the speaker.

Program Design

Figure 9-29 is the flowchart of the program, and Figure 9-30 is a listing of the program. Refer to the listing for the following discussion.

The first three system EQU's (labeled IW, W4, and P4) are used for initializing the two-word, level-four interrupt vector. Recall that the 200-millisecond real-time clock will appear as a series of interrupts at level four. To process these interrupts, a level-four vector must be initialized at memory location 10_{16} (WP) and location 12_{16} (PC). The system EQU labeled WE is used to identify to the time-of-day clock program the entry address of the cycle generator program at location 220_{16}. The last two system equates, labeled CT and DT, are the two values that are used for a 200-millisecond timer command and a 1-millisecond timer command, respectively, for the two TMS 9901 devices on the University Board (at U11 and U10).

This program's starting address is 250_{16}, which is a memory location below the last address of the cycle generator program. (See the program example in Chapter 6 for a listing of the cycle generator program.) The LWPI instruction at location 250_{16} initializes the workspace pointer, and the following instructions establish the interrupt vector for the level-four interrupt. The CLR instruction at location 264_{16} initializes memory location TK to zero. This memory cell accumulates a count of the number of clock ticks. The contents of location TK is incremented by one from the level-four interrupt-service routine each time a clock tick occurs.

The memory location which records the operator's selected alarm time is initialized to zero at locations 268_{16} and $26C_{16}$.

The sequence of instructions beginning at label T1 starts the operator interaction. The XOP instruction at location 274_{16} is used to display a prompting message to the operator requesting the desired function. The two possible functions include setting an alarm time and setting the current time. The operator indicates that an alarm time is to be set by entering the character A or that the current time is to be set by entering the character T. The program checks to ensure that the character entered is either and A or T. If the operator enters some character other than these two, the program displays the function-prompt message again and awaits another entry.

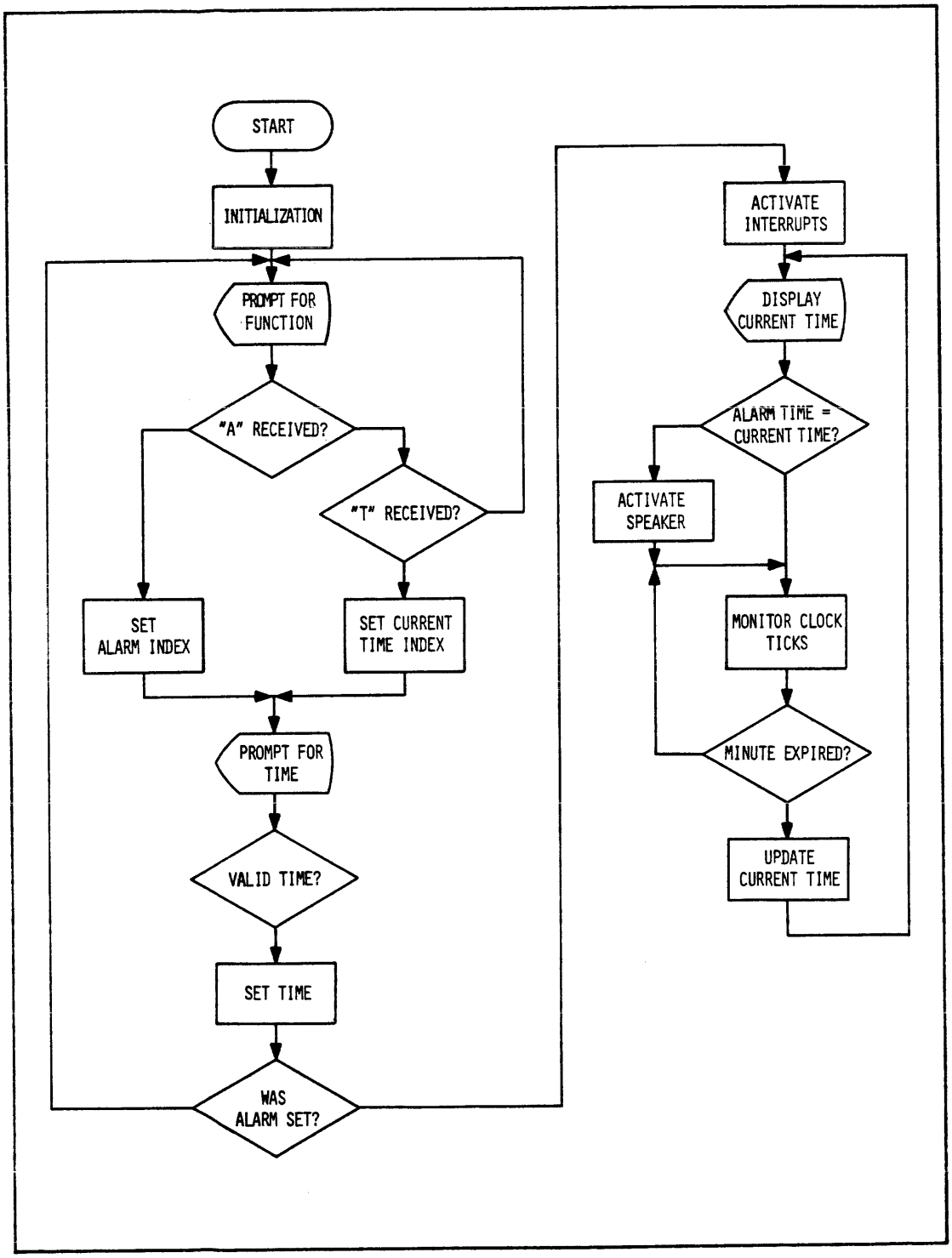


Figure 9-29. Flowchart for Time-of-Day Clock Program

```
0001                    IDT  'TODCLK'
0002              *
0003              *  TIME-OF-DAY CLOCK PROGRAM
0004              *
0005              *  SYSTEM EQUATES
0006              *
0007       01E0   IW    EQU  >1E0          ADRS OF WP AREA FOR INTRPT 4
0008       0010   W4    EQU  >0010         ADRS OF WP VECTOR FOR INTRPT 4
0009       0012   P4    EQU  >0012         ADRS OF PC VECTOR FOR INTRPT 4
0010       0220   WE    EQU  >220          ADRS OF WAVE CYCLE ENTRY POINT
0011       30D5   CT    EQU  >30D5         200 MS. 9901 TIMER COMMAND
0012       003F   DT    EQU  >003F         ONE MS. 9901 TIMER COMMAND
0013              *
0014              *
0015 0250                ORG  >250
0016 0250 02E0   TC      LWPI >200          INIT WP
     0252 0200
0017 0254 0200           LI   R0,IW         SET INTRPT LEVEL 4 VECTOR
     0256 01E0
0018 0258 C800           MOV  R0,@W4
     025A 0010
0019 025C 0200           LI   R0,IS
     025E 03F0
0020 0260 C800           MOV  R0,@P4
     0262 0012
0021 0264 04E0           CLR  @TK           ZERO OUT CLOCK TICK COUNTER
     0266 03E0
0022 0268 04E0           CLR  @AL           INITIALIZE ALARM TIME OFF
     026A 03D2
0023 026C 04E0           CLR  @AL+2
     026E 03D4
0024 0270 0202   T1      LI   R2,AL         SET INDEX TO ALARM TIME
     0272 03D2
0025 0274 2FA0           XOP  @EF,14        PROMPT FOR FUNCTION
     0276 03A6
0026 0278 2EC0           XOP  R0,11         RECEIVE A CHAR
0027 027A 0240           ANDI R0,>7F00      STRIP OFF PARITY BIT
     027C 7F00
0028 027E 9800           CB   R0,@AT        IS CHAR AN "A"?
     0280 03BC
0029 0282 1305           JEQ  T2            YES
0030 0284 9800           CB   R0,@AT+1      IS CHAR A "T"?
     0286 03BD
0031 0288 16F3           JNE  T1            NO
0032 028A 0202           LI   R2,TM         ADJUST INDEX TO CURRENT TIME
     028C 03DA
0033 028E 2FA0   T2      XOP  @ET,14        PROMPT FOR TIME
     0290 03B2
0034 0292 2E40           XOP  R0,9          READ TIME INPUT
0035 0294 0270           DATA T1            NO INPUT-REPROMPT FOR FUNC.
0036 0296 028E           DATA T2            BAD VALUE-REPROMPT FOR TIME
0037              *  VALIDITY CHECK FOR VALID DECIMAL CHARACTERS
0038 0298 C040           MOV  R0,R1         HOURS CHARS OK?
0039 029A 0241           ANDI R1,>0FFF
     029C 0FFF
0040 029E 9801           CB   R1,@C9
     02A0 03CC
0041 02A2 1BF5           JH   T2            NO
0042 02A4 06C1           SWPB R1            CHECK MINS CHARS
0043 02A6 9801           CB   R1,@TB        MINS CHARS OK?
```

Figure 9-30. Listing of Clock Program

```
     02A8 03C4
0044 02AA 1BF1          JH    T2              NO
0045 02AC 0241          ANDI  R1,>0FFF
     02AE 0FFF
0046 02B0 9801          CB    R1,@C9
     02B2 03CC
0047 02B4 1BEC          JH    T2              NO
0048 02B6 9800          CB    R0,@LM          DEC. CHARS IN BOUNDS?
     02B8 03BE
0049 02BA 1BE9          JH    T2              NO
0050 02BC 06C0          SWPB  R0              GET MIN.'S ENTRY IN LEFT BYTE
0051 02BE 9800          CB    R0,@LM+1        VALID MINUTES ENTRY?
     02C0 03BF
0052 02C2 1BE5          JH    T2              NO
0053                 *  CONVERT DIGITS TO ASCII CHARS
0054 02C4 C0C2          MOV   R2,R3           GET TIME INDEX IN R3
0055 02C6 0204          LI    R4,4            INIT COUNTER
     02C8 0004
0056 02CA 0BC0  T3      SRC   R0,12           R. JUSTIFY DIGIT IN LEFT BYTE
0057 02CC DCC0          MOVB  R0,*R3+         MOVE TO MEMORY
0058 02CE 0604          DEC   R4              THRU?
0059 02D0 16FC          JNE   T3              NO
0060 02D2 C0C2          MOV   R2,R3           RESET TIME INDEX
0061 02D4 0204          LI    R4,4            INIT COUNTER
     02D6 0004
0062 02D8 0201          LI    R1,>F030        SET MASK VALUES IN R1
     02DA F030
0063 02DC 5CC1  T4      SZCB  R1,*R3+         ISOLATE DIGITS
0064 02DE 0604          DEC   R4              THRU?
0065 02E0 16FD          JNE   T4              NO
0066 02E2 C0C2          MOV   R2,R3           RESET TIME INDEX
0067 02E4 0204          LI    R4,4            INIT COUNTER
     02E6 0004
0068 02E8 06C1          SWPB  R1              PUT NEW MASK IN R1 LEFT BYTE
0069 02EA FCC1  T5      SOCB  R1,*R3+         ADD >40 TO DIGIT
0070 02EC 0604          DEC   R4              THRU?
0071 02EE 16FD          JNE   T5              NO
0072 02F0 0282          CI    R2,AL           ALARM OR CURRENT TIME SET?
     02F2 03D2
0073 02F4 13BD          JEQ   T1              ALARM TIME WAS SET
0074                 *  INIT U11 9901 TIMER TO 200 MS. FOR TOD CLOCK
0075 02F6 020C          LI    R12,>400        SET CRU BASE ADRS TO 9901
     02F8 0400
0076 02FA 0201          LI    R1,CT           GET TIMER VALUE
     02FC 30D5
0077 02FE 33C1          LDCR  R1,15           START TIMER
0078 0300 1E00          SBZ   0               SWITCH 9901 TO INTRPT MODE
0079 0302 1D03          SBO   3               ENABLE 9901 INTRPT 3
0080                 *  INIT U10 9901 TO 1 MS. FOR DISPLAY REFRESH
0081 0304 020C          LI    R12,>0          SET BASE ADRS TO U10 9901
     0306 0000
0082 0308 0201          LI    R1,DT           GET TIMER VALUE
     030A 003F
0083 030C 33C1          LDCR  R1,15           START TIMER
0084 030E 1E00          SBZ   0               SWITCH 9901 TO INTRPT MODE
0085 0310 1D03          SBO   3               ENABLE U10 9901 INTRPT 3
0086 0312 1D06          SBO   6               ENABLE INTTRPT 6 ON U10 9901
0087 0314 0300          LIMI  4               ENABLE 9980 INTRPT 4
     0316 0004
0088 0318 2FA0  T6      XOP   @DM,14          PRINT 'TOD ' WITH TIME
```

Figure 9-30. Listing of Clock Program (Continued)

```
      031A 03D6
0089  031C 8820        C     @TM,@AL       COMPARE ALARM:CURRENT TIME
      031E 03DA
      0320 03D2
0090  0322 160B        JNE   T7
0091  0324 8820        C     @TM+2,@AL+2
      0326 03DC
      0328 03D4
0092  032A 1607        JNE   T7
0093  032C 0200        LI    R0,>A           SET UP 1KHZ, 5 SEC. TONE
      032E 000A
0094  0330 C040        MOV   R0,R1           FOR WAVE CYCLE SUBROUTINE
0095  0332 0202        LI    R2,>1388
      0334 1388
0096  0336 06A0        BL    @WE
      0338 0220
0097  033A 8820  T7    C     @TK,@SC         60 SEC. ELASPED?
      033C 03E0
      033E 03CA
0098  0340 1AFC        JL    T7              NOT YET
0099  0342 6820        S     @SC,@TK         YES-UPDATE TICK COUNTER
      0344 03CA
      0346 03E0
0100  0348 0201        LI    R1,>0100        PUT A BYTE OF 1 IN R1
      034A 0100
0101  034C B801        AB    R1,@TM+3        ADD ONE MIN.
      034E 03DD
0102  0350 9820        CB    @TM+3,@MV+3     ADD TO TEN MINS?
      0352 03DD
      0354 03C3
0103  0356 1626        JNE   T9              NO
0104  0358 D820        MOVB  @A0,@TM+3       YES-& SET MIN. TO ZERO
      035A 03CE
      035C 03DD
0105  035E B801        AB    R1,@TM+2
      0360 03DC
0106  0362 9820        CB    @TM+2,@MV+2     ADD TO HOURS?
      0364 03DC
      0366 03C2
0107  0368 161D        JNE   T9              NO
0108  036A D820        MOVB  @A0,@TM+2       YES-&SET TEN MINS TO ZERO
      036C 03CE
      036E 03DC
0109  0370 B801        AB    R1,@TM+1
      0372 03DB
0110  0374 9820        CB    @TM,@MV         OVER 2000 HOURS?
      0376 03DA
      0378 03C0
0111  037A 1B0A        JH    T8              YES
0112  037C 9820        CB    @TM+1,@MV+1     HRS. OVER 9?
      037E 03DB
      0380 03C1
0113  0382 1610        JNE   T9              NO
0114  0384 D820        MOVB  @A0,@TM+1       YES-SET HRS TO ZERO
      0386 03CE
      0388 03DB
0115  038A B801        AB    R1,@TM           AND BUMP TEN HOURS
      038C 03DA
0116  038E 100A        JMP   T9
0117  0390 9820  T8    CB    @TM+1,@A4       HRS OVER 3?
```

Figure 9-30. Listing of Clock Program (Continued)

```
     0392 03DB
     0394 03D0
0118 0396 1606          JNE  T9              NO
0119 0398 D820          MOVB @A0,@TM+1       YES-SET HRS TO ZERO
     039A 03CE
     039C 03DB
0120 039E D820          MOVB @A0,@TM          AND TEN HRS TO ZERO
     03A0 03CE
     03A2 03DA
0121 03A4 10B9  T9      JMP  T6              GO DISPLAY IT
0122                    *
0123 03A6 0D0A  EF      DATA >0D0A           CR,LF
0124 03A8   41          TEXT 'A OR T ?'      FUNCTION PROMPT
     03A9   20
     03AA   4F
     03AB   52
     03AC   20
     03AD   54
     03AE   20
     03AF   3F
0125 03B0 0000          DATA 0
0126 03B2 0D0A  ET      DATA >0D0A           CR,LF
0127 03B4   54          TEXT 'TIME?          TIME PROMPT
     03B5   49
     03B6   4D
     03B7   45
     03B8   3F
     03B9   20
0128 03BA 0000          DATA 0
0129 03BC   41  AT      TEXT 'AT'            FUNC. COMPARISON TEST
     03BD   54
0130 03BE 2359  LM      DATA >2359           LIMITS FOR TIME
0131 03C0 313A  MV      DATA >313A           ASCII LIMITS FOR HRS AND MINS
0132 03C2 363A          DATA >363A           CHECK VALUE
0133 03C4 9F00  TB      DATA >9F00           CHECK VALUE
0134 03C6 0000  ZR      DATA 0               BYTE CONSTANT - 0
0135 03C8 0100  C1      DATA >0100           BYTE CONSTANT - 1
0136 03CA 012C  SC      DATA >12C            NO. OF TICKS IN A SEC
0137 03CC 0900  C9      DATA >0900           BYTE CONSTANT - 9
0138 03CE 3000  A0      DATA >3000           ASCII'0'
0139 03D0 3400  A4      DATA >3400           ASCII'4'
0140                    *
0141 03D2 0000  AL      DATA 0               ALARM TIME
0142 03D4 0000          DATA 0
0143 03D6 544F  DM      DATA >544F           ASCII'T' AND '0'
0144 03D8 4420          DATA >4420           ASCII'D' AND SPACE
0145 03DA 0000  TM      DATA 0               CURRENT TIME
0146 03DC 0000          DATA 0
0147 03DE 2000          DATA >2000           DELIMITER FOR CHAR STRING
0148 03E0 0000  TK      DATA 0               CLOCK TICK COUNTER
0149                    *
0150                    *  INTERRUPT SERVICE ROUTINE
0151                    *
0152 03F0               AORG >3F0
0153 03F0 05A0  IS      INC  @TK             BUMP CLOCK TICK COUNTER
     03F2 03E0
0154 03F4 020C          LI   R12,>400        SET CRU ADRS TO 9901
     03F6 0400
0155 03F8 1D03          SBO  3               CLEAR & REENABLE 9901 INTRPT
0156 03FA 0380  I2      RTWP                 RETURN-REVERSE CONTEXT SWITCH
```

Figure 9-30. Listing of Clock Program (Continued)

```
TODCLK          SDSMAC      3.1 *    11:03:41 FRIDAY, DEC 22, 1978.
                                                              PAGE 0006

 0157                *
 0158       0250            END   TC
NO ERRORS
```

Figure 9-30. Listing of Clock Program (Concluded)

Once the operator has selected the appropiate function, the program then displays a prompting message to the operator for the time desired. The operator enters the four digits of either the alarm time or the current time. The program checks for the validity of these characters to ensure that it is a valid time; that is, that the series of digits constitute a decimal value between 0 and 2359_{10}. Once the time has been validated, it is converted to ASCII characters and saved in memory locations AL and AL + 2 (in the case of an alarm time) or memory locations TM and TM + 2 (in the case of the current time).

If an alarm function is selected by the operator, the operator enters the selected alarm time and, once the alarm time has been validated and set in memory, the function prompt is again issued to the operator. At this time, the operator may enter another A character to change the alarm time. The alarm time can be reset as often as necessary. Once the alarm time has been finalized, the operator then enters a T character in response to the function prompt to indicate that the current time is to be set. The program receives the entered characters indicating the current time and validates them as it did for the alarm time. As soon as the current time is validated and set, the program then activates the 200-millisecond real-time clock with the series of instructions beginning at location $2F6_{16}$. Register 12 is loaded with the value 400_{16} to establish the CRU base address to the "display" TMS 9901. The timer command for a 200-millisecond delay (which is $30D5_{16}$) is sent to the TMS 9901 device with the LDCR instruction at location $2FE_{16}$. The SBZ 0 instruction at location 300_{16} sets the TMS 9901 to interrupt mode, and the SBO 3 instruction at location 302_{16} enables the TMS 9901's interrupt level three (which is the level of interrupt for a TMS 9901 interval timer). The series of instructions beginning at location 304_{16} initializes a 1-millisecond interval timer on the "user" TMS 9901 at location U10 on the TM 990/189. This 1-millisecond interval timer is used to ensure a periodic refresh of the characters that are being displayed on the on-board terminal. This 1-millisecond interval timer is presented to the TMS 9980A as a level-one interrupt. The interrupt vector for level one is initialized by the monitor when the board is powered up or reset. Notice that it is necessary to execute an SBO 6 instruction at (location 312_{16}) in order to pass the level-four interrupt from the display TMS 9901 to the user TMS 9901 so that both interrupts are compatible within the system.

Once the interval timers have been initiated, the program is active and will begin displaying the current time of day by displaying the characters "TOD " along with the four characters constituting the hours and minutes. The XOP instruction at location 318_{16} is used to display the characters. The time of day selected by the operator has been set by the program in memory location TM. The program compares the current time of day with the selected alarm time, and, if the two times are equal, the program activates the speaker to produce a 1-kHz, 5-second tone. If the current time is not equal to the alarm time, the program continues monitoring memory location TK. Location TK is the memory cell that is used

by the real-time clock interrupt-service routine to record the individual clock ticks. The program monitors this location to determine when 60 seconds has expired. When this occurs, the program then updates the current time of day and returns to the memory location labeled T6 where the updated time of day is displayed and the current time of day is again compared to the alarm time.

Beginning at label EF, the program's data constants and data variables are defined. Notice that the ASCII characters indicating the current time of day are stored in 4 bytes of memory beginning at the location labeled TM. Immediately preceding these 4 bytes holding the current time are 4 bytes (labeled DM) which hold the ASCII characters "TOD ." The XOP 14 instruction at label T6 will display the string of characters constituted by TOD and the four characters of the current time whenever the instruction is executed.

The four instructions beginning at location $3F0_{16}$ constitute the entire interrupt-service routine. Notice that the interrupt-service routine has been made as small as possible in order to limit the amount of time spent at an "interrupt level." The interrupt-service routine simply records the fact that a clock tick has occurred by incrementing the contents of memory location TK and then addresses the TMS 9901 and reactivates the TMS 9901 level-three interrupt to enable the next interrupt from the device. The RTWP instruction reverses the context switch and causes an exit from the interrupt-service routine.

Program Operation

Once this program (along with the cycle generator program) has been entered into memory, the program counter may be set to the value 250_{16} and the program executed. (Do not forget to change the last instruction in the cycle generator program from a B to a B *R11 since it will be accessed in this program by a BL instruction.)

The program begins by printing the message A or T? on the display. This prompting message asks the operator to choose the alarm or time function. The operator should respond by entering the character A or the character T. The A indicates that the operator wishes to set the alarm time. The T character indicates that the operator wishes to set the current time.

Once the A or T character is entered, the program displays the prompting message TIME?, which is a prompt to the operator asking for a time to be entered. The operator should respond by entering four digits of either an alarm time or current time (the first two digits 00 to 23 for the hour and the second two digits 00 to 59 for the minute).

After setting the alarm time, the function-prompting message is given again, at which time the operator may set a new and different alarm time by entering another A character, or he may now choose to select the current time by entering a T character.

When the T function has been selected, the prompting message TIME? is given, and the operator should respond by entering four digits indicating the current time expressed as a 24-hour clock value as with the A function. As soon as the operator has entered the current time (assuming a valid value), the program immediately initiates the interval timers and begins maintaining the time of day based upon the value entered by the operator.

The program displays the characters TOD (indicating time of day) along with four digits showing the current time of day. This display is updated each minute. If the operator has set an alarm time, the program will sound the speaker for a 5-second period when the alarm time is reached. As long as the program is kept active it will maintain and display the current time of day.

9.8 SUMMARY

This chapter is a study of software engineering. Software engineering begins with the decision of which system functions to implement in software rather than in hardware.

In developing the software, the designer can apply the methodologies of top-down design, structured programming, and program modularity, each of which can potentially increase the speed of development, ease the debugging task, and improve the documentation.

The individual programs composing the system must reside in memory together. Techniques for linking programs together have been discussed, including ROM/RAM partitioning, the use of memory maps, the use of assemblers producing relocatable object code, and the use of relocating loaders.

The introduction of interrupts into a system design adds another level of complexity. Special considerations have been discussed which should be given to the structure of interrupt-service routines in a real-time application.

Many real-time applications include a real-time operating system which can add another degree of flexibility to a system application design.

9.9 EXERCISES

1. What two main advantages does a software approach have over a hardware approach to a system function?

2. What considerations might lead to a hardware approach rather than a software approach?

3. When compared to high-level language, assembler language normally produces a faster _____ time, but a slower _____ time.

4. Consider an application program designed to accept a number consisting of up to four decimal digits from a keyboard, followed by either a plus sign or a minus sign, and a second number consisting of up to four other decimal digits. The second number is either added to the first or subtracted from it depending upon whether a plus sign or a minus sign was entered. The digits entered must be checked to ensure that they are decimal digits and the operation character entered by the operator must be checked to ensure it is either a plus sign or a minus sign. After the program calculates the answer, the result is displayed. Use a top-down design to create a block diagram of the program.

5. Use structured programming techniques to develop a flowchart of the above program.

6. Flowchart the following program segments and identify the type of program structure used.

```
(a)       MOVB  @CT,R0
          JEQ   AA
          AB    R0,@C1
          JMP   AB
     AA   AB    R0,@C2
     AB   MOVB  R0,@KN


(b)       LI    R0,0
          LI    R3,>1D00
     Z    MOV   R0,*R3+
          CI    R3,>1E00
          JL    Z


(c)       MOV   @IM,R0
          ANDI  R1,>7F7F
          SOC   R1,R0
          MOV   R0,@RG


(d)       C     @A,@B
          JGT   DW
          JEQ   OT
          NEG   @A
          JMP   OT
     DW   C     @A,@C
          JGT   OT
          INC   @A
          JMP   DW
     OT   B     *R11
```

7. From the program example, identify the following intermodule communication and linkage techniques.

(a) Why is memory location 0250_{16} used as the starting address of the TODCLK program?

(b) How is the entry point of the cycle generator program identified by this program?

(c) What mechanism is employed to link together this program and the user-accessible utilities in the firmware monitor?

(d) How does the level-four interrupt-service routine communicate to the main program that a clock tick has occurred?

8. In the program example, which memory addresses contain variable data that must be implemented in RAM?

9. From the program example, create a memory map showing the memory locations that could be placed in ROM and the addresses that must be placed in RAM. (Be sure to include the cycle generator program.)

10. Assume that the portion of the two programs that can be placed in ROM will be placed in 256 x 8-bit memory components beginning at address 0200_{16}. Draw a memory map showing the starting address and ending address of each of the two programs. Determine the number of ROM components required.

11. Write the assembly language instructions necessary to initialize a TMS 9901 interval timer and to establish a level-one interrupt-service routine to process the series of interrupts produced by the timer. An interrupt should be produced every 20-milliseconds. Assume a 2-MHz system clock. The CRU hardware base address for the TMS 9901 is 0100_{16}. Included within its processing, the interrupt-service routine must turn on a CRU bit at the beginning of every half-second period and then turn it off after one-tenth of a second. The CRU hardware base address for the CRU bit is 309_{16}.

12. The following program segment is designed to move a series of characters from one place in memory to another place. The number of characters moved is variable, depending upon the count in memory location CT. What changes are necessary in the calling programs and this program to make this program segment reentrant?

```
     LI    R2,A1        PUT "FROM" ADRS IN R2
     LI    R3,A2        PUT "TO" ADRS IN R3
     MOV   @CT,R4       PUT THE CHAR COUNT IN R4
LP   MOVB  *R2+,*R3+    MOVE A CHAR
     DEC   R4           THRU?
     JNE   LP           NO
```

9.10 LAB EXPERIMENTS

1. Modify the program example to maintain the alarm time and current time as BCD digits rather than as ASCII characters.

2. Modify the program example to convert it from a 24-hour clock to a 12-hour clock with AM and PM indication. Display the time of day as the series of characters HH:MM AM or HH:MM PM (where HH indicates the current hour and MM indicates the current minute).

3. Modify the program example to maintain and display the current second as well as the current hour and minute. Additionally, modify the program to produce an audible beep on the speaker every second. A routine resides in the UNIBUG firmware to produce a short, 2.5 kHz beep. This function can be abtained with the following instruction:

```
BLWP  @>3008
```

4. Develop a program that functions as a countdown timer. Allow an operator to input the countdown value as hours, minutes, and seconds. Count down the time and turn on the speaker when the time has expired.

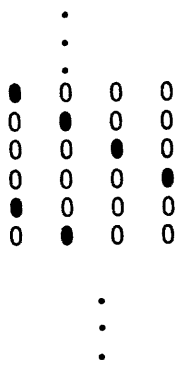
5. Develop a program to function as a stop watch. When the operator presses a key, record the elapsed time (by updating a series of memory locations) until the LOAD switch is activated. At that time, examine the memory locations to read the elapsed time. Record the time to tenths of a second.

6. Develop a program to simulate a real-time operating system. Assume there are four tasks in the system. Each task is assigned a period of time and an order of execution as follows.

Task 0	-	4 seconds
Task 1	-	3 seconds
Task 0	-	4 seconds
Task 2	-	3 seconds
Task 0	-	4 seconds
Task 3	-	2 seconds

Each task has an ID code of 0 to 3. As each task is executing, it displays its ID as a BCD value on the four user-addressable LED's.

7. Develop a program to produce the illusion of a rotating light on the four user-addressable LED's by lighting one light at a time in sequence at a periodic interval. The sequence of lights should be as follows.



8. Modify the above program to allow the operator to vary the speed of "rotation." As an example, the operator might enter the digit 1 for the slowest rotation and the digit 9 for the fastest rotation. The decimal digits between 1 and 9 would represent a relative rotation speed between the slowest speed and the fastest speed.

9. Utilize the TM 990/189 to develop the program in Exercise 4.

CHAPTER 10

PRODUCT DEVELOPMENT

10.1 INTRODUCTION

The goal of this book is to prepare the student for the application of microprocessors to the solution of engineering problems. Previous chapters equip the student with specific tools, primarily assembly language programming, and acquaint him with the University Board in the process of developing skill in the use of these tools. This chapter summarizes the general flow of effort from the conception of an idea to the delivery of a product, for a microprocessor-based system.

10.2 PRODUCT DEVELOPMENT OVERVIEW

Figure 10-1 presents a general flowchart identifying the major steps in developing a microprocessor-based system from both the hardware and software development viewpoints. Even though the steps may be different, it is apparent that there is a remarkable degree of similarity in structure in the two paths. Since the primary thrust of this book is more closely related to the software effort, more attention is given to that area rather than the hardware area in the remainder of this chapter.

The entry point is conception--that instant when the idea for a product is born--the point that someone some time asks the question, "Why don't we build a newer and better gizwidgit?" From that point, the remainder of the effort begins to take some form.

The next step is product definition. It is here that some of the most important work is done, since a poor definition hinders the future effort. A good market analysis is often the key to a good product design, for even the most cleverly designed product may be a money-losing dust-gatherer if there is no one to buy it. On the other side of the picture, a well-written set of specifications often serves as a good basis for an efficient design effort.

At system-design time, a design outline is created with enough detail so that the work can be parceled out to members of the design team. If this step is done well, the individual designers do not have to be concerned with details outside the scope of their assignment. It is here that the functions of the product are allocated to either hardware or software. High-level design constraints are also decided, such as whether to use assembly language or a high-level language.

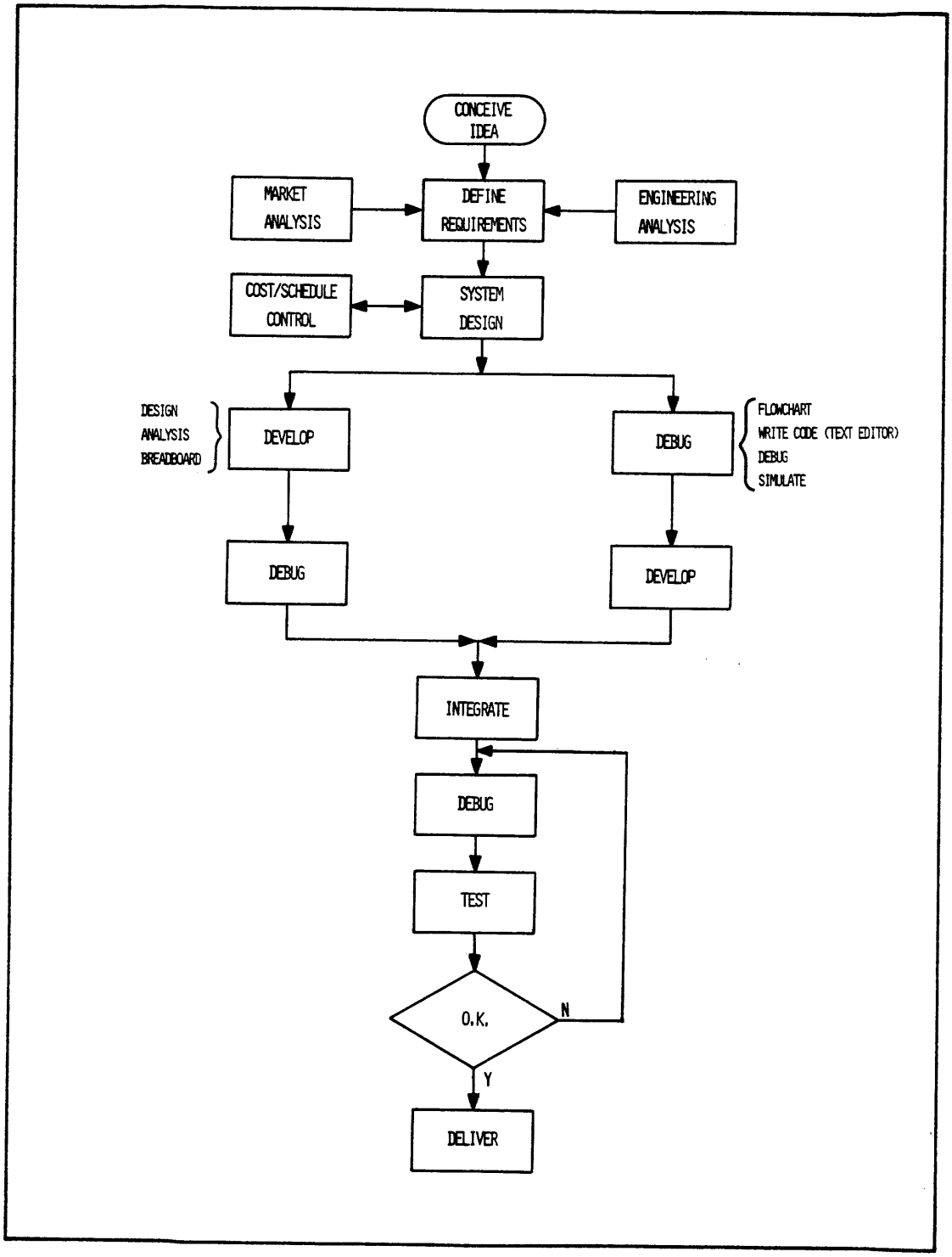


Figure 10-1. Project Development Flowchart

Development is the stage wherein the design objectives are realized. To the hardware designer, this implies circuit design, breadboarding, and debugging. To the software designer, it means writing, executing, and debugging code. Up to this point, the product has literally been "on the drawing boards." As development proceeds, the first signs of life of the product begin to appear, and it is here that engineering has some of its greatest emotional rewards.

The transition from development to debugging is so vague that the two are often executed concurrently, or at least appear to be. The words, "I never designed a circuit that didn't work--it was the ones I built that gave me trouble," distinguish the concepts of development and debugging. Debugging, in both software and hardware, is the most unstructured of all the steps of product development.

The final steps, testing and delivery, appear to be far simpler than they really are. Although not all products require formal testing, the wise manufacturer knows that a good final testing program will save him much effort in customer relations. By the same token, the manner in which a product is delivered, from hand-held calculators to electronic pinball machines, can either enhance or deteriorate the customer's willingness to continue as a customer.

In the remainder of this chapter, these areas are discussed in greater detail, with the aim of enlightening the reader in some aspects of engineering which hitherto might have gone underemphasized.

10.3 PRODUCT DEFINITION

The detailed discussion of Figure 10-1 will be clearer if a specific product is used as an example. Since most microprocessor applications are special-purpose, a controller will be chosen to illustrate the hardware decisions to be made. Suitable for this discussion is an elevator controller for a four-story building. This is the conception that initiates the entire process to be discussed in the next several pages. Having conceived the idea, the next step is product definition.

The controller is configured in this way: The first and fourth floors have one call button each (up on the first, down on the fourth) and floors two and three have two each (up and down). Each floor has a light associated with each call button to indicate unserviced requests. Each floor has two approach switches, and a floor-level switch to control elevator speed. Inside the car are four switches and lights to indicate destination requests and four numbered lights indicating approaching floors. The motor is controlled by two lines, one for direction and one for speed. The speed control is pulse-width modulation. A steady logic ONE results in full speed; a steady logic ZERO results in zero speed. Intermediate speeds are controlled by proportional or variable duty-cycle pulse trains. The last line is the door-control line. Figure 10-2 illustrates the basic interface diagram of this system. The controller is housed

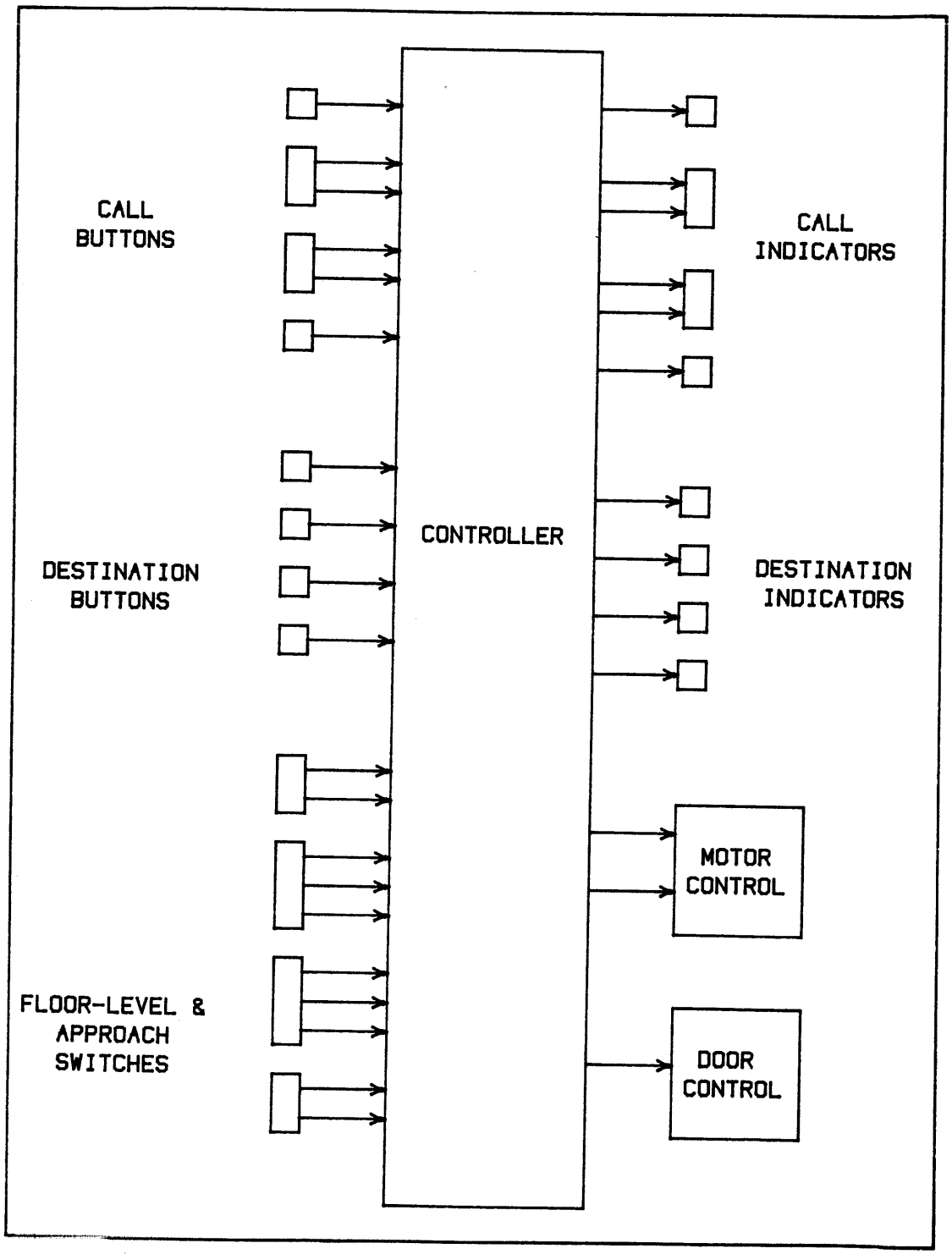


Figure 10-2. Elevator Controller Interface Diagram

in the elevator equipment room for ease in maintenance, and has manual-override capability for the sake of safety.

This is a highly simplified product specification. In practice, the detailed specifications comprise a formal document several pages long, including such things as

- Environmental constraints
- Target price
- Packaging constraints
- Power limitations.

For the moment, the details already provided are sufficient to begin the system design effort. As the need for more detail arises, that detail will be defined. In the world of engineering, it is a rare experience indeed when all the details are known ahead of time.

10.4 SYSTEM DESIGN

Having defined the system, the next step is to consider alternate approaches to the design. The task of evaluation is especially complex in the area of controller design, because there are so many options. A full treatment of this is beyond the scope of this book, but a few words are in order to indicate some of the major considerations.

Random Logic Controller

One approach to controller design is called random logic. With this approach the designer converts the product definition into a logic diagram, which is realized with a collection of small-, medium-, and large-scale integrated circuits. This approach is generally appropriate when the resulting design uses fewer than about 50 integrated circuit devices, or when a high-speed requirement dictates it. For example, magnetic-disk memory controllers are often designed using this technique because of the speed requirement, even though the resulting designs may use well over a hundred small-, medium-, and large-scale IC's.

A major drawback to random logic design is the difficulty of correcting errors. This difficulty increases drastically toward the end of the development effort. There are ways to guard against this, and when a random logic design is appropriate, the designer builds in certain "safety exits," to be used only in the event of a last-minute change.

ROM-Driven Controller

Another approach to consider is a ROM-controlled state generator. There are many variations to this, but Figure 10-3 illustrates a

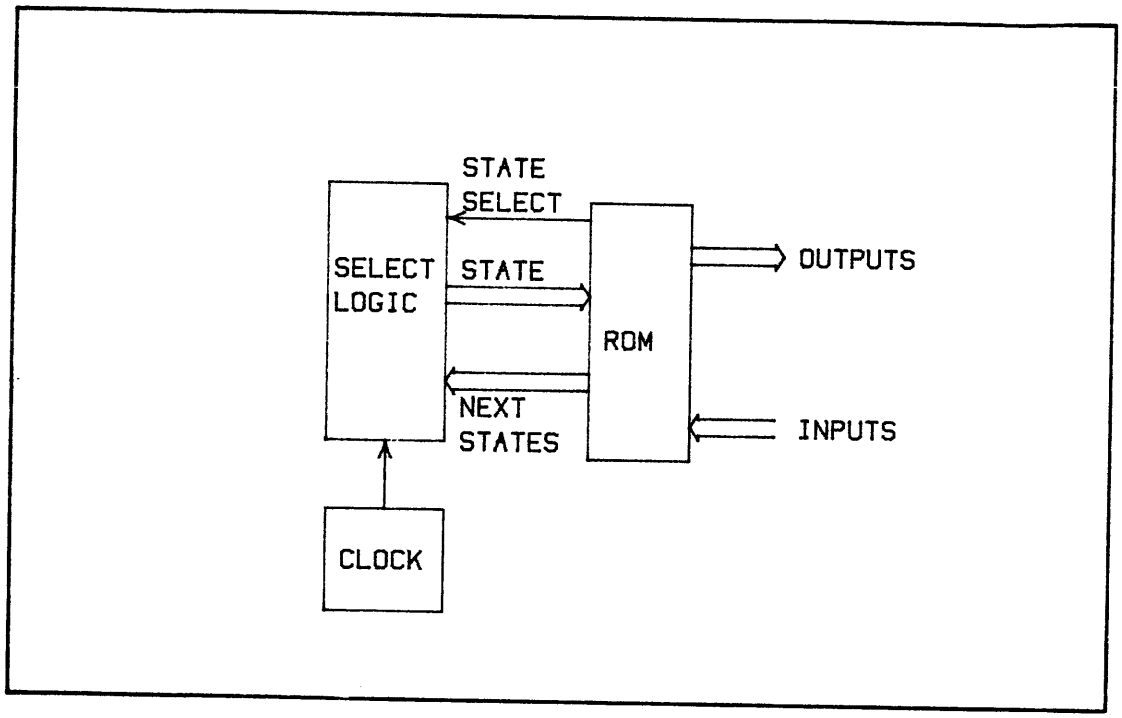


Figure 10-3. ROM-Controlled State Generator

possible block diagram. In this system, the ROM decodes the state number from the state generator, producing an output, an input selector, and a pair of next states. An input multiplexer selects which of several possible inputs are requested and passes the input to the state generator. The state generator uses this to select which state to go to next. With this approach, hardware design can proceed very quickly, since all the control is in the ROM. In this respect, this type of controller is very similar to a microprocessor-based controller.

A major limitation of the ROM-driven controller is the difficulty of debugging and modifying the control logic. Often the state flow diagrams take many pages to draw, and tracing through them to find the source of an error can be quite an undertaking.

Microprocessor-Based Controller

The last approach to be considered here is microprocessor-based design. Many questions need to be answered; these fall into a few broad categories. The first is simply "Should a microprocessor be used?" Within this category are the following considerations.

- Will the manufacturing cost be reduced?
- Will the share-of-market be enhanced?
- Will product capabilities be improved?
- Will there be any impact on future development efforts?

Of these four questions, the share-of-market idea and future development bear some expansion. The percentage of market dollars spent on a particular product is a complex function of many factors, but a key rule for getting a high share of any market is simple: Get a good product into the marketplace as soon as you can! Any design technique which results in a good product in a short time is a good technique. For many products of any complexity at all, microprocessors offer a good design technique.

The question of future development enhancement is often of prime importance, especially to a manufacturer doing his first microprocessor product. The financial investment for the first design can be high in terms of personnel/training, capital-equipment purchases, manufacturing setup, etc. However, once the original investment is made, the benefits propagate throughout successive design efforts.

Hardware/Software Tradeoffs. Having had the first category of questions answered, and a microprocessor-based design decided upon, the next broad category is "How shall the functions be partitioned between hardware and firmware?" (Recall that firmware refers to a program stored in ROM.) The basic guideline to use here is to get the best product possible into the marketplace as soon as possible. Obviously, there are tradeoffs implied in that standard, and it is necessary to evaluate those tradeoffs. These guidelines indicate the major factors to consider:

- Project schedule
  - Can a hardware function requiring a long-lead-time part be done in firmware?
  - Can a subroutine be written and debugged in less time than a circuit can be designed and tested?
- Budget
- Personnel
  - Are the total hardware skills better than the total software skills?
  - Can the necessary expertise be acquired quickly?
- Flexibility: Can changes be made easily?
- Technical risk
- Microprocessor capability
- Off-the-shelf hardware availability
- Off-the-shelf software availability
- Interfacing requirements
- Reliability, both hardware and software.

Design Considerations. In the system design stage, there are usually a number of design considerations in both hardware and software. Depending on the nature of the product, examples of these are

Hardware	Software
Type of construction - PC Board (logic density, or how many layers?) - Wrapped wire Environmental constraints Standard components Documentation	Modularity Standardized subroutine interfacing I/O fully interrupt mode Documentation

Some products may be able to take advantage of existing microcomputer circuit boards, such as the TM 990/189. It is possible that in many applications, the hardware design cost can be reduced substantially by using such a board. Also, the on-board firmware can be used to enhance software development.

10.5 SYSTEM DEVELOPMENT

Once a microprocessor is selected, memory requirements are estimated and the external interface is defined. Hardware development can proceed concurrently with software development. Software development often takes longer than hardware development for several reasons. With modern development tools, it is often the practice to start developing software without adequate preparation in terms of product definition. The potential designer should bear this in mind and carefully document all phases of development, so that backtracking is simplified. As much as possible, definition should precede design and development, but, in any case, the successful software engineer is capable of preparing for and responding to changes and additions in the product definition.

In most design projects, the software development can benefit from a team effort. In its simplest form, a team might be only two persons: one working on the main program, the other on the subroutines. Good teamwork can be the key to the success of the project.

Special Software Aspects

Because software must be developed on a computer for transfer to a special-purpose piece of hardware, it is often necessary for the design to accommodate two different sets of I/O interface routines. One set consists of the development system configuration, whereas the other is the production system software. The next few paragraphs discuss some of the special considerations implicit

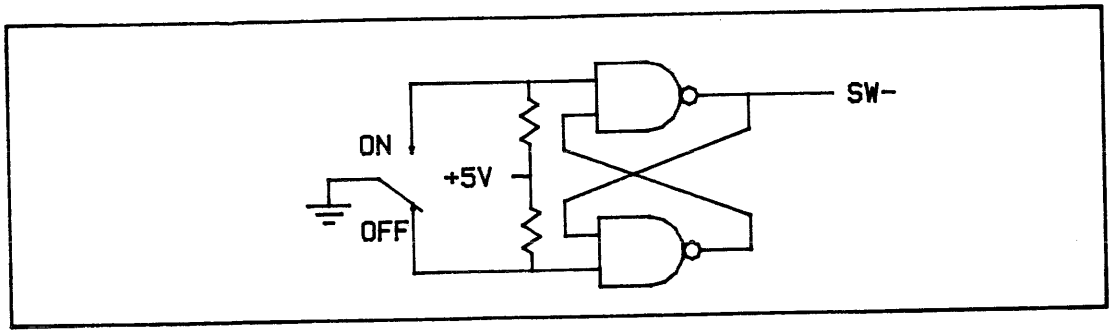


Figure 10-4. Hardware Switch Debouncing

```
WAIT  TB    SW      TEST SWITCH
      JNE   WAIT
      BLWP  DELAY   DELAY FOR BOUNCE
      TB    SW      TEST AGAIN
      JNE   WAIT    REPEAT IF OFF
      ...
```

Figure 10-5. Software Switch Debouncing

in the I/O system software design for the elevator controller example.

Clearly the I/O software must be developed in close conjunction with the I/O hardware. Signal polarities, I/O port numbers, and timing requirements usually involve tradeoffs between software and hardware.

A good example of the tradeoff decisions to be made is switch debouncing. It is a fact well-known to hardware designers that when a switch is opened or closed, there is a transition period, roughly depending on the physical size of the switch, during which the state of the switch alternates rapidly and randomly. This is called switch bounce. In high-speed circuits, it is the same as many openings or closures in a brief period of time. On closure, bounce is caused by the switch armature literally bouncing against the stator. On opening, it is caused by surface irregularities. In many applications, it is necessary to provide a bounce-free input. Figure 10-4 illustrates a common way of doing this in hardware, and Figure 10-5 illustrates a common way of handling it in software.

In some applications hardware debouncing is preferred, whereas in others software debouncing is more appropriate. In a few cases the decision may be clearcut, but careful cost/performance/reliability analysis is usually required.

In general, the I/O software development should be concerned not only with the hardware/software interface but also with the format in which information is passed to the controller logic. In this respect, the checkout software must simulate the controller logic to a certain extent.

For example, in the elevator controller a subroutine is called to determine which call buttons are depressed. In the production system the routine scans the call buttons and returns a floor number and direction. The internal logic of the routine obviously depends on the arrangement of the switches. The development system has a different input configuration, so the corresponding routine reflects this. For example, using only the University Board as a development tool necessitates employing the keyboard as an input device. There is in firmware a routine that scans the keys and returns a number indicating which one is depressed. An intermediate routine is necessary to decode the key number into a floor and direction number.

Hardware Development Technique

Since the main thrust of this book is software-oriented, most of the emphasis has been on software. Nevertheless, there are some microprocessor-related hardware development techniques to be considered. Generally, in hardware development the basic steps are

- Block diagram
- Circuit design
- Breadboard
- Prototype.

Ideally, the design is not released to the manufacturing group until all these things have been done to the satisfaction of the project manager. In practice, however, the design may be released to manufacturing as early as the beginning of the breadboard phase, due to the long setup time often required. Therefore, it is important to perform the first two steps very carefully so as to minimize the redesign required by discoveries made during the breadboarding and prototyping phases.

In general, the breadboards should reflect the anticipated final configuration as much as possible. This may help uncover potential problem areas such as heat dissipation, mechanical clearances, etc. Wrapped-wire is often a good technique for breadboarding digital circuits, because it is neat, is easily modified, and makes durable connections. A wrapped-wire breadboard can be constructed about twice as fast as a solder-type breadboard.

10.6 SOFTWARE DEVELOPMENT

A basic approach to software development is called "top-down" design. This is the method in which a high-level, functional program is written first, with only token subroutine calls to indicate the

overall structure of the program. Then, as development proceeds, more and more refinement becomes appropriate, and the program literally evolves to its final form.

As an example of top-down design with respect to the elevator controller, see Figure 10-6. The following discussion refers to this figure and describes a typical development procedure.

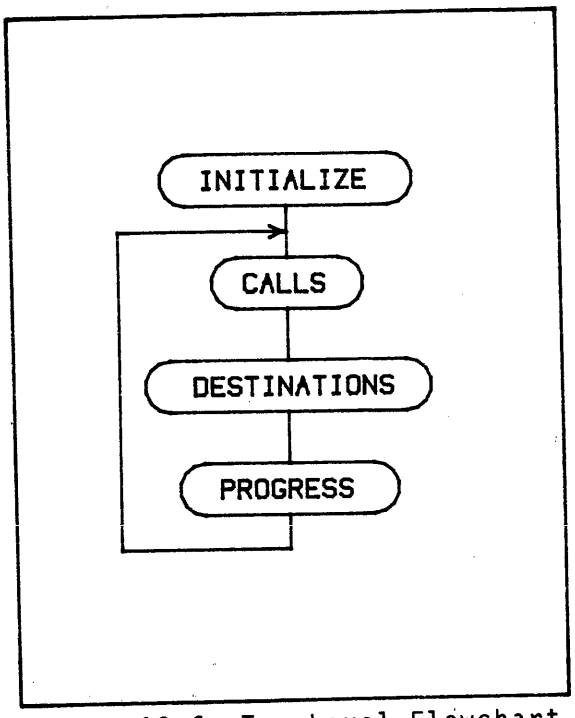


Figure 10-6. Top Level Flowchart, Elevator Controller

The "initialize" block initializes all the necessary variables to represent an elevator waiting at the first floor with no calls to service, no passengers aboard, and the doors open, waiting for either a call or a destination.

The "calls" block scans the call buttons, returning the numbers of the floors and the directions requested. The "destinations" block scans the destination buttons, returning the number of the floor requested. The "progress" block examines the calls, destinations, and current state to set motor speed and direction, and clear appropriate calls and destinations.

At this point, the power of top-down development in controller applications becomes apparent. All the I/O hardware interfacing is relegated to subroutines, which are selected to represent either of several possible hardware configurations. The first executable code block in the development process may well look like the listing in Figure 10-7.

It is important to recognize that the code of Figure 10-7 does absolutely nothing functionally. It does, however, provide the framework for the ultimate controller program. The only thing remaining is to put some functional code into IS (Initial State), CS (Call State), DS (Destination State), and PS (Progress State).

Once the overall program structure is set, there begins a complex combination of analysis, code writing, reformulation, and trading off, that cannot be described adequately because of the individuality of each program. The more experience an individual has, the more able he is to plan his work so as to decrease the total turn-around time.

```
WS   BSS   32
ST   BLWP  @IN
LP   BLWP  @CA
     BLWP  @DE
     BLWP  @PR
     B     @LP
IN   DATA  IW,IS
IW   BSS   32
IS   RTWP
CA   DATA  CW,CS
CW   BSS   32
CS   RTWP
DE   DATA  DW,DS
DW   BSS   32
DS   RTWP
PR   DATA  PW,PS
PW   BSS   32
PS   RTWP
     END   ST
```

Figure 10-7. Top Level Controller Code

An example of the decisions and tradeoffs that must be made is apparent from Figure 10-7. A gross misuse of programmable memory is obvious here, since 160 bytes are required for workspace registers alone. An approach that would be beneficial here is to use BL calls wherever possible. When different workspaces are required, they can often be overlapped, so that a subroutine requiring only, say, four registers, might overlap another workspace by 12 registers.

A prime reason for using great care in software development is the usually limited ROM space available for program storage. If the program requires 2200 words of ROM and only 2048 words are available, then either the programmer has to be more clever, or some of the program requirements have to be eased, or more ROM has to be made available.

10.7 DEBUGGING, TESTING, AND DELIVERY

Debugging is simplified by the top-down approach. As development proceeds, individual modules are added to the total package, one at a time. The best procedure is usually to debug each module "off-line," creating only enough external code as necessary to check out the internal operation of the module. Then, when it is added to the primary package, any resulting errors should be a result of interfacing (software interface). Adding and debugging only one module at a time is a good rule of thumb, but the developer should always remember that most rules have exceptions, and should be ready to make them when necessary.

After all the design, development, and debugging efforts are completed, the final step is testing. At first glance, it may appear that all the testing should have been accomplished at debug time. However, the purpose of testing, in this context, is not so much to determine if the software is correct, but to demonstrate that it is. Because of this distinction, testing is often given

a formal name, such as "performance testing" or "acceptance testing." In most companies this is also a carefully controlled and documented process, usually involving skilled personnel who are unfamiliar with the detailed design of the product. The lack of familiarity assures that the test is unbiased.

Performance tests are more important than they may appear at first. They are important to the customer, because he is assured of having a working product with known performance standards. Performance tests are also important to the designer in that he has a well-defined standard of performance to guide his efforts. If possible, the performance tests should be defined early in the development effort. Often the creation of the performance tests will expose potential weaknesses or omissions in the product definition.

Having designed, developed, debugged, and proved the performance of a product, the next step, perhaps the final step, is delivery. This usually entails much more than just physical delivery of a piece of hardware. Often a certain amount of training is required.

Depending on the product, the training may be for operators, maintenance people, management, or any combination of these. In some cases, the type of training required may affect some aspect of the product design. Operating manuals, maintenance manuals, and other types of documentation are usually required, especially for complex and more sophisticated products.

Maintenance is often a key factor in the successful delivery of a product. Thus, the designer can expect that the customer's maintenance requirements will be reflected in the design. For instance, if a customer has a short mean-time-to-repair (MTTR) requirement, a built-in diagnostic program and modular hardware design might help meet that. On the other hand, the customer may have a long mean-time between-failures (MTBF) requirement, in which case the emphasis would go to high-reliability design, perhaps even to the extent of redundancy and fault-tolerant logic.

The last phase of any system-design project is the support phase. Many projects require a certain level of continuous support beyond delivery to assist in system modification, software enhancements, trouble-shooting, etc.

10.8 DEVELOPMENT TOOLS

Just as the hardware designer needs the right tools for his job, so does the software designer need proper tools. These tools take the form of hardware, firmware, and software carefully integrated into an effective means for software development.

To support the 990/9900 series of microprocessor products, Texas Instruments has developed a variety of tools. The TM 990/100M, for instance, is a self-contained computer using the TMS 9900 micro-

processor, with 16 I/O pins and provision for a small amount of breadboarding on the board. There are similar boards in support of several of the major 8-bit microprocessors. These are usually nearly minimal configurations, with the ability to expand with easily added attachments such as ROM, RAM, communications, and mass-storage peripherals.

A key part of any development system is the "operating system." Basically, the operating system is a set of programs, subroutines, and hardware that interface the development system to its user. Even the simplest microcomputer board has an operating system.

The simpler systems, such as that on the University Board, are usually called "monitors." Monitors provide the basic functions of memory inspection and modification, terminal I/O, and in many cases some form of mass-memory I/O. Other operations may be provided for the benefit of the user, such as simple hexadecimal arithmetic.

Firmware-Based Development System

The TM 990/302, shown in Figure 10-8, is an example of a firmware development system. It is compatible with the TM 990 line and was designed specifically as a software development tool. It consists of one board which includes

- A single or dual audio-cassette interface
- An EPROM programmer
- An EIA port
- 4K bytes of RAM
- 8K bytes of nonvolatile memory containing program-development firmware.

The EPROM programmer has the option of programming several of the most commonly used EPROM's, including the TMS 2708, TMS 2716, TMS 2516, and TMS 2532. The programmer can also program the EPROM version of the TMS 9940 single-chip microcomputer when this product is released.

The 8K bytes of nonvolatile memory on the TM 990/302 includes a text editor, symbolic assembler, relocating loader, the firmware for the EPROM programmer, and a debugger.

The TM 990/302 is designed to be used with a TM 990/100 or TM 990/101 CPU board, a user-supplied terminal, power supply, and audio-cassette player/recorder to form a complete software development system.

As options, a user also has available a chassis and additional memory boards. In addition, a BASIC language interpreter can be employed.

The text editor within the firmware is used to create a new source program or modify an existing source program. The source

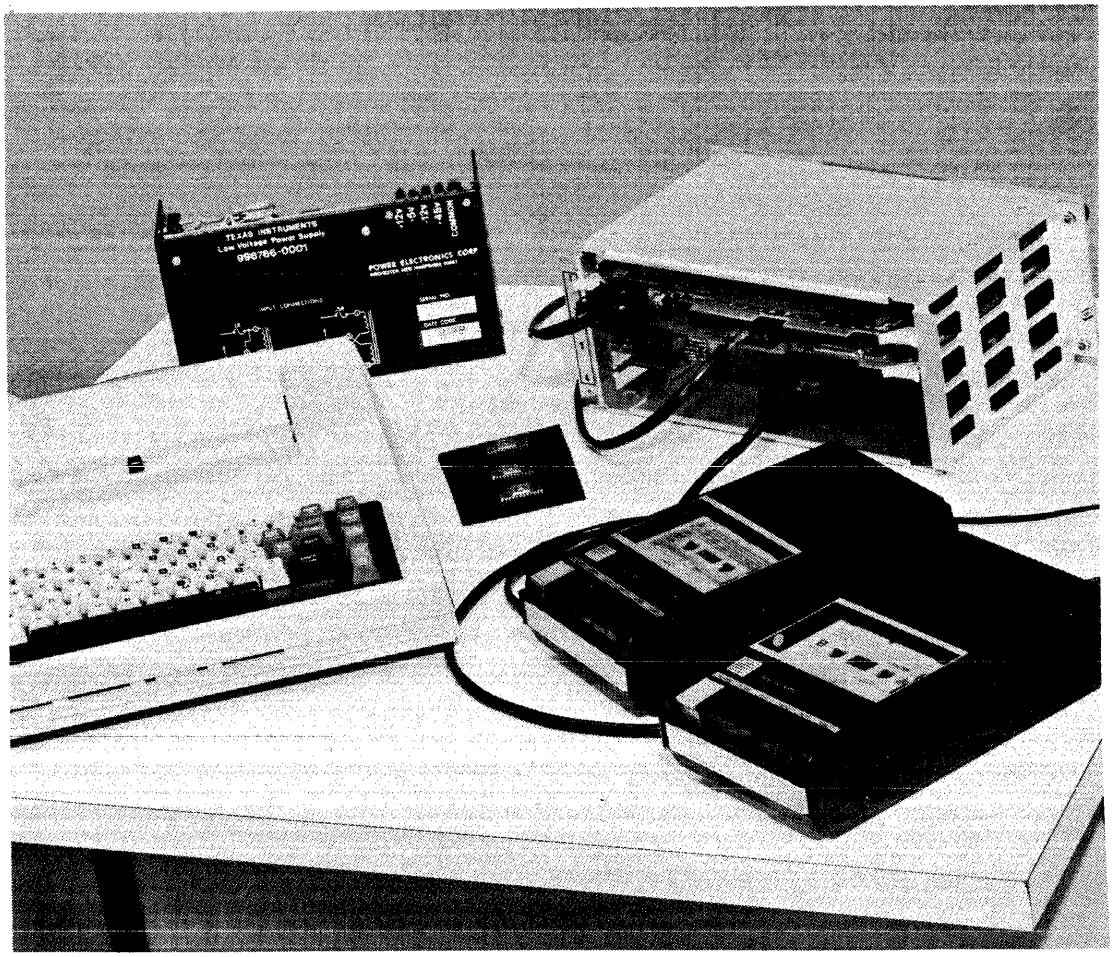


Figure 10-8. TM 990/302 Board

program can be stored on audio cassette and input to the symbolic assembler, which translates the source program into machine code and writes the resultant machine code onto an object cassette, while at the same time writing a program listing to a hard-copy terminal used with the system.

With the symbolic assembler on the TM 990/302, the user may create absolute programs which can be loaded into memory using the firmware-resident loader.

The TM 990/302 software development system provides a user with a complete microprocessor development system for the creation of 9900 family software.

Floppy-Based Development System

The next step in complexity of development tools is the complete software development system. This usually consists of a micro-powered minicomputer with an adequate supply of memory, a keyboard terminal, a printer, and some mass memory, such as floppy disk or high-speed cassette tape. Floppy disks are rapidly superseding cassette tapes because of cost, performance, and reliability. For such a tool, the operating system provides control of the entire system while supplying smooth control of the applications programs.

Operating systems are available for any desired level of cost/performance. Careful planning is required in selecting the first development system or in upgrading. An example of the importance of this derives from the previous discussion of the I/O routine development. For development purposes, programmers generally take the inputs from the development-system keyboard, with necessary logical buffering to interface the I/O routine to the controller logic. When changing operating systems, those I/O routines generally need to be changed to reflect the new operating system.

The more complex operating systems are for more than one user at any given instant. This is called time-sharing. In most cases, this can radically increase the overall throughput of the system as compared to a single-user system. Throughput is a key factor in assessing the cost/performance rating for a development system.

Texas Instruments has two purely software entries in the development tool category to be discussed here: AMPL* and POWER BASIC.*

AMPL is a high-level language designed to operate in a floppy disk system (FS990) for the purpose of prototyping microprocessor systems (see Figure 10-9). Some of its key features are

- Block-structured concepts and simple, concise syntax
- Straightforward interactive evaluation and display of expressions
- User- and system-defined procedures and functions
- Interactive display and modification of memory
- Instruction assembly and disassembly
- Symbolic debug.

These features, and others, allow the user to develop support tasks for emulation, debugging, and evaluation to ease the microprocessor development effort.

AMPL-based microprocessor prototyping labs are available, complete with all the necessary hardware and software support to make the most of this language.

POWER BASIC is a family of software products providing a wide range of capabilities, for a wide range of computer configurations,

* AMPL and POWER BASIC are trademarks of Texas Instruments.

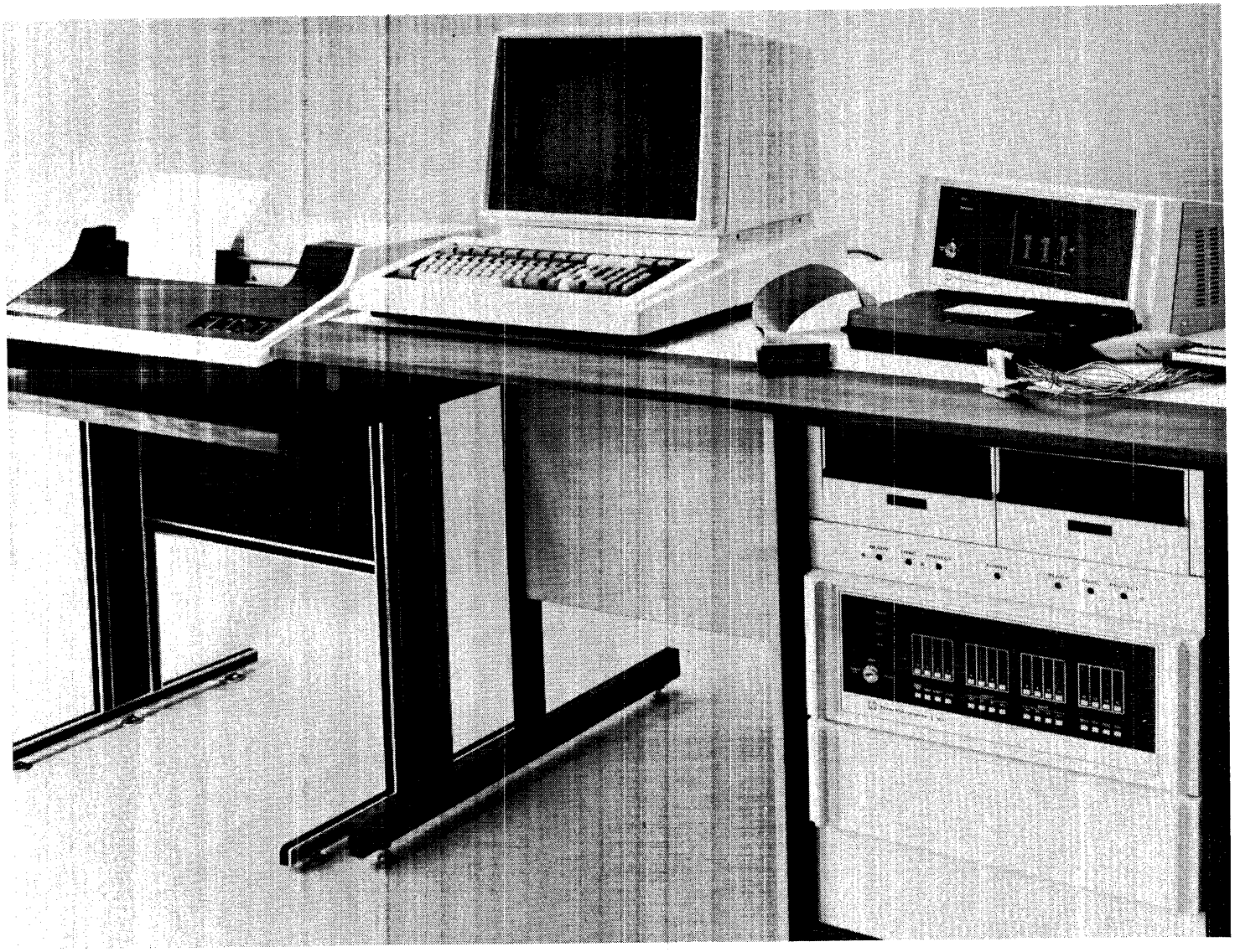


Figure 10-9. AMPL System

from TM 990 boards to floppy disk systems. This family of products adds to standard BASIC some features designed to support real-time control applications. Two members of this family are specifically intended for the TM 990/100M and /101M boards, with some memory expansion. Features such as two-user capability, audio-cassette storage/retrieval, and the interactive format of BASIC make this a very useful entry in the field of development tools.

High-Level Language

For a large-scale development effort, the use of a high-level language is generally advisable, and often mandatory. Whereas assembly language reflects the way the computer operates, high-level languages reflect the way the programmer thinks. In most large applications, the result of using high-level languages is a shorter development time. The price to pay for this improvement is additional investment in a few key areas:

- Greater initial cost in development system software.
- Programs that run slower and take up more memory space.
- Usually greater effort in I/O software interface development.

The decision to use high-level language as opposed to assembly language is sometimes hard to make, and is often very intuitive, influenced heavily on the experience of the participants.

To describe all of the high-level languages applicable to microprocessor systems would be difficult in a short space since they are being developed at such a rapid pace. However, some of the key languages that have come to the forefront and will apparently stay for a while are BASIC, Pascal, and PLM. Of these, the one with probably the greatest potential is Pascal, because of its structure, power, and self-documenting characteristics. The interested reader can see references for a full treatment of the language. The others mentioned above, although well entrenched in microprocessor use, suffer from lack of speed, power, or intelligibility.

This chapter is an overview of the development process, to prepare the student for the real world of software engineering. However, there is no teacher and no course like that of experience. Thus, the reader is encouraged to actively seek out and develop microprocessor applications. The knowledge and skills gained in experimentation are invaluable assets for future work in microprocessor system development.

10.9 PROGRAM EXAMPLE CONTINUED: MOTOR CONTROL RAMP GENERATOR

In the previous discussion, an elevator controller was used as an example to illustrate some of the problems in software development. Since the unexpanded University Board will not provide memory

requirements for such a controller, only a portion of it will be used as an example. Recall that the motor driver is described as a variable-duty-cycle pulse train. This example generates such a pulse train, and applies it to one of the onboard LED's. The brightness of the LED is approximately proportional to the speed of the hypothetical motor. The reader is encouraged to experiment with a variable-duty-cycle pulse train to verify that above about 50 percent the duty cycle has no effect on the apparent brightness. This fact is used in the program design.

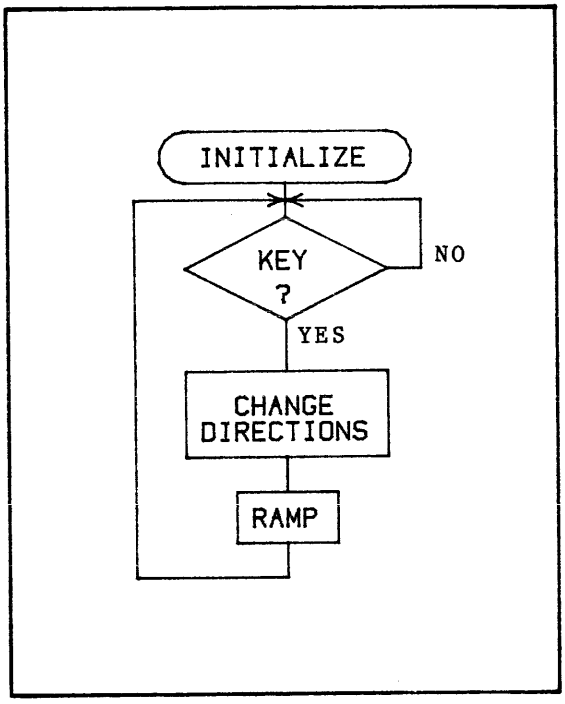


Figure 10-10. Top-Level Flowchart, Ramp Generator

The specifications are as follows.

- A basic period of 10 milliseconds
- Ramp up and ramp down in three seconds
- Initially, the LED will be off. When a keyboard key is depressed, it will ramp up to full brightness (50 percent duty cycle). Thereafter, every key depression results in a ramped change in state, during which the keyboard is ignored until the steady state has been reached.

A top-level flowchart that indicates this operation is given in Figure 10-10.

The keyboard is tested by using XOP 13 to simplify the effort. Any key besides Ret will result in a ramp. Ret causes a return to UNIBUG.

The ramp is generated by using the TMS 9901 timer. A counter keeps track of progress in the ramp interval. This counter is used to compute the value required to set the timer. It is necessary to establish a procedure for loading the timer. Consider the following.

The maximum interval is 0.01 second. Referring to Chapter 7, the reader can verify that the timer has to be set to 312 to yield a 0.01-second interval. In three seconds there will be a total of 600 timer timeouts. To meet these requirements, an intricate

scheme of multiplies and shifts is necessary. At this point, an alternative is proposed. A coding efficiency can be achieved by loading the timer with either 1/2 the count or 312 minus 1/2 the count, if the count is allowed to go to 624, rather than 600. This means a ramp interval of 3.12 seconds, or a 4 percent increase. In this application, that is quite acceptable, and the alternative is accepted.

A detailed flowchart is presented in Figure 10-11. Notice that to load the timer with a count of N, the value sent to the timer via LDCR must be 2*N+1.

The code that excutes this routine is given in Figure 10-12. The reader is encouraged to experiment with variations of it, such as the ramp time, period, and keyboard logic as suggested in the lab experiments. For example, one might desire to ignore all keys except A and D for accelerate and decelerate.

10.10 SUMMARY

In this chapter the overall product-development effort is described with particular emphasis on the tradeoffs and decisions involved in most development efforts. The reader has seen some of these principles applied to a product example, an elevator controller, and should now have a better understanding of the level of detail required at various stages of development.

The program example has been given to illustrate a small development effort, including a decision to modify the original product description. Engineering projects are extensions of the principles illustrated in the example, and the reader is encouraged to involve himself as much as possible in the formulation and execution of engineering projects so as to better prepare himself for a productive engineering experience.

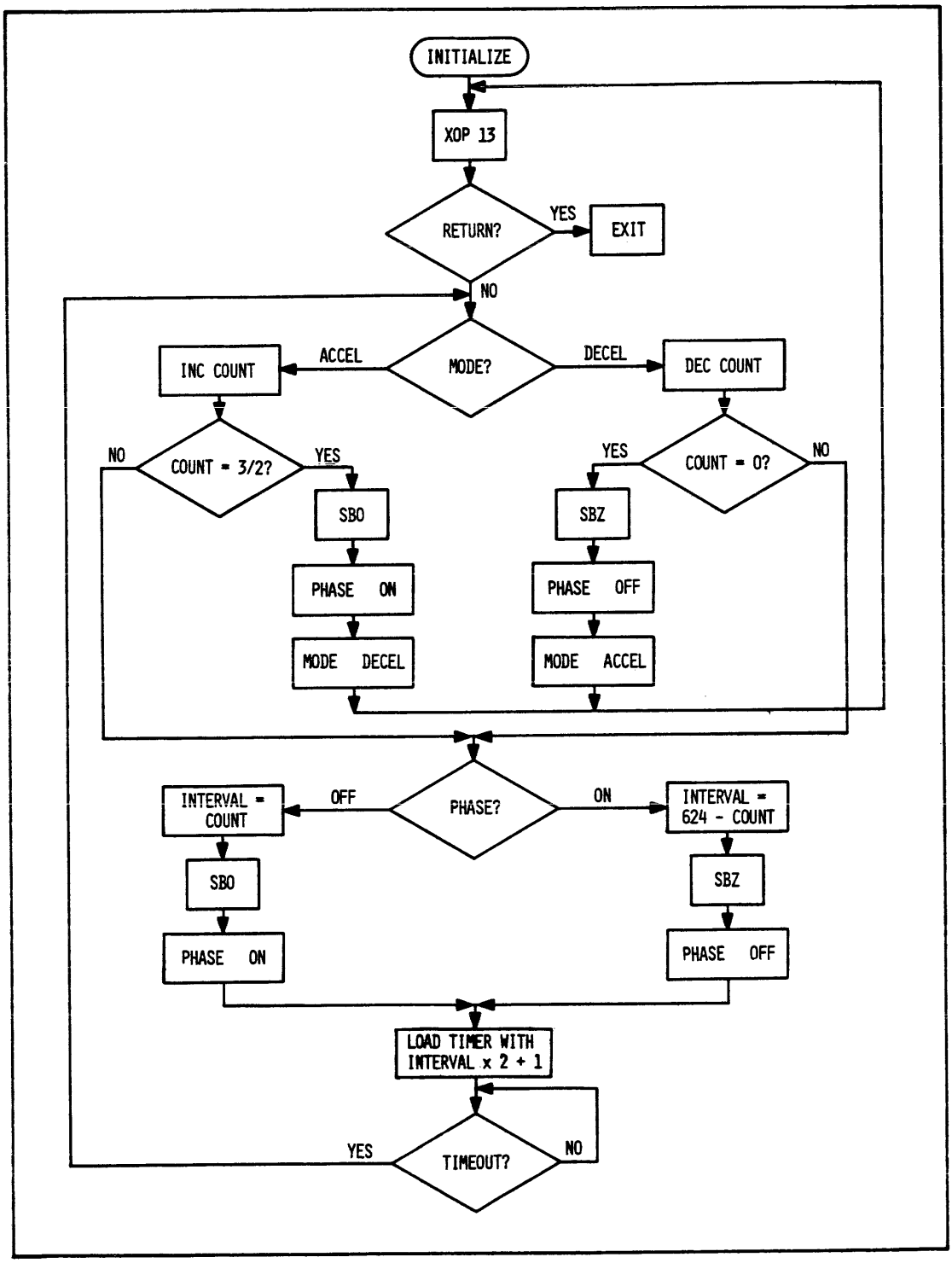


Figure 10-11. Detailed Flowchart, Ramp Generator

```
0001 0200                AORG >200
0002       000C  R12     EQU  12
0003       0004  VW      EQU  4                    INTERRUPT VECTOR
0004       0006  VP      EQU  6
0005       0000  TM      EQU  0                    TIMER REGISTER
0006       0001  TP      EQU  1                    TEMPORARY REGISTER
0007       0002  WP      EQU  2                    POINTER REGISTER
0008       0003  MD      EQU  3                    MODE REGISTER
0009       0006  M2      EQU  6                    R3 OFFSET
0010       0004  PH      EQU  4                    PHASE REGISTER
0011       0008  P2      EQU  8                    R4 OFFSET
0012       0005  CT      EQU  5                    COUNT REGISTER
0013       000A  C2      EQU  10                   R5 OFFSET
0014       0006  DN      EQU  6                    DONE FLAG REGISTER
0015       000C  D2      EQU  12                   R6 OFFSET
0016       0000  KR      EQU  0                    KEY REGISTER
0017       0138  MX      EQU  312                  MAX COUNT
0018       0010  LD      EQU  16                   LED OFFSET
0019       0003  CK      EQU  3                    CLOCK MASK
0020       0000  U9      EQU  0                    USER TMS9901 ADDRESS
0021 0200        MW      BSS  32                   MAIN WORKSPACE
0022 0220        IW      BSS  32                   INTERRUPT WORKSPACE
0023 0240 02E0   ME      LWPI MW                   MAIN ENTRY
     0242 0200
0024 0244 020C           LI   R12,U9
     0246 0000
0025 0248 0200           LI   TM,1                 STOP TIMER
     024A 0001
0026 024C 33C0           LDCR TM,15
0027 024E 0202           LI   WP,IW                CLEAR INTERRUPT WORKSPACE
     0250 0220
0028 0252 04E2           CLR  @M2(WP)
     0254 0006
0029 0256 04E2           CLR  @P2(WP)
     0258 0008
0030 025A 04E2           CLR  @D2(WP)
     025C 000C
0031 025E 04E2           CLR  @C2(WP)
     0260 000A
0032 0262 2F40   LP      XOP  KR,13                GET KEY
0033 0264 0280           CI   KR,>0D00             RETURN?
     0266 0D00
0034 0268 1316           JEQ  EX                    YES, EXIT
0035 026A 0201           LI   TP,IW                SET UP VECTOR
     026C 0220
0036 026E C801           MOV  TP,@VW
     0270 0004
0037 0272 0201           LI   TP,IE
     0274 029A
0038 0276 C801           MOV  TP,@VP
     0278 0006
0039 027A 0300           LIMI 1
     027C 0001
0040 027E 04E2           CLR  @D2(WP)              CLEAR DONE FLAG
     0280 000C
0041 0282 0200           LI   TM,MX+MX+1           SET TIMER
     0284 0271
0042 0286 33C0           LDCR TM,15
0043 0288 0200           LI   TM,>0800             ENABLE INTERRUPT
     028A 0800
```

Figure 10-12. Program Listing, Ramp Generator

```
0044 028C 3100          LDCR TM,4
0045 028E C062  WT      MOV  @D2(WP),TP       WAIT FOR DONE FLAG
     0290 000C
0046 0292 13FD          JEQ  WT
0047 0294 10E6          JMP  LP
0048 0296 0460  EX      B    @>3000
     0298 3000
0049 029A 020C  IE      LI   R12,U9           INTERRUPT ENTRY
     029C 0000
0050 029E 04C0          CLR  TM               DISABLE INTERRUPTS
0051 02A0 3100          LDCR TM,4
0052 02A2 C0C3          MOV  MO,MO            MODE?
0053 02A4 1307          JEQ  AC
0054 02A6 0605          DEC  CT                DECELERATE
0055 02A8 1610          JNE  CP               NOT DONE, SKIP
0056 02AA 0586          INC  DN               SET DONE FLAG
0057 02AC 1E10          SBZ  LD               TURN OFF LED
0058 02AE 04C4          CLR  PH               SET PHASE TO OFF
0059 02B0 04C3          CLR  MO               SET MODE TO ACCELERATE
0060 02B2 101D          JMP  RN
0061 02B4 0585  AC      INC  CT                ACCELERATE
0062 02B6 0285          CI   CT,MX
     02B8 0138
0063 02BA 1607          JNE  CP               NOT DONE, SKIP
0064 02BC 0586          INC  DN               SET DONE FLAG
0065 02BE 1D10          SBO  LD               TURN ON LED
0066 02C0 0201          LI   TP,1
     02C2 0001
0067 02C4 C101          MOV  TP,PH            SET PHASE TO ON
0068 02C6 C0C1          MOV  TP,MO            SET MODE TO DECELERATE
0069 02C8 1012          JMP  RN
0070 02CA C104  CP      MOV  PH,PH            PHASE?
0071 02CC 1306          JEQ  OF                OFF, SKIP
0072 02CE 0200          LI   TM,MX+MX
     02D0 0270
0073 02D2 6005          S    CT,TM            INTERVAL = 2*MAX-COUNT
0074 02D4 1E10          SBZ  LD               TURN OFF LED
0075 02D6 04C4          CLR  PH               SET PHASE TO OFF
0076 02D8 1003          JMP  LT
0077 02DA C005  OF      MOV  CT,TM
0078 02DC 1D10          SBO  LD               TURN ON LED
0079 02DE 0584          INC  PH               SET PHASE TO ON
0080 02E0 0A10  LT      SLA  TM,1             GET INTERVAL INTO POSITION
0081 02E2 0260          ORI  TM,1             SET TIMER MODE BIT
     02E4 0001
0082 02E6 33C0          LDCR TM,15
0083 02E8 0200          LI   TM,>0800         RETURN TO INTERRUPT MODE
     02EA 0800
0084 02EC 3100          LDCR TM,4
0085 02EE 0380  RN      RTWP
0086      0240          END  ME
NO ERRORS
```

Figure 10-12. Program Listing, Ramp Generator (Concluded)

B I B L I O G R A P H Y

The following are several lists of selected publications that can assist the reader in finding additional information and references on the various topics covered in the text.

ARTICLES

Altman, Laurence. "Single-Chip Microprocessors Open Up a New World of Applications," Electronics, April 18, 1974, pp.81-100.

Backler, Jordan. "Disk Storage Devices," Digital Design Handbook, December, 1974, pp. 57-75. A detailed development of disk memory systems including electrical and mechanical specifications.

Baldridge, Ronald L. "Interrupts Add Power, Complexity to uC-System Design," EDN, August 5, 1977, 67-73.

Benson, Terry. "Microcomputers: Single-Chip or Single-Board?," Electronic Design, June 21, 1978, pp. 86-91.

"16-Bit Microprocessors," Electronic Products Magazine, July, 1978, pp. 24-31. Report of a forum on 16-bit microprocessors with participants from semiconductor firms and designers (potential buyers).

Burton, D. Phillip. "Know a Microcomputer's Bus Structure," Electronic Design, June 21, 1978, pp. 78-84.

"The Computer Society: The Age of Miracle Chips," Time, February 20, 1978, pp. 44-45. One of a number of cover stories on microprocessors/microcomputers and their impact on the home, business, science, and society in general.

Cushman, Robert H. "EDN's Fourth Annual Microprocessor Directory," EDN, November 20, 1977, pp. 44-83.

Davis, Sidney. "Selection and Application of Semiconductor Memories," Computer Design, January, 1974, pp. 65-77.

Electronics, April 15, 1976. Special issue which includes a roundup of available components and other articles on evaluating software, etc.

Frankenberg, Robert J. "Designer's Guide to Semiconductor Memories," EDN, August 5, 1975, pp. 22-35. The first of a comprehensive ten-part series which goes from memory component theory through memory system design, development and maintenance.

Franson, Paul. "Special Report: Semiconductor Memories," EDN, June 20, 1977, pp. 46-58. An excellent survey of present and future availability of semiconductor memory including pin-out and function compatibility.

Gellender, Edward. "Learn Microprocessor Fundamentals," Electronic Design, October 11, 1977, pp. 74-79.

Gladstone, Bruce. "Software Development: Some Tools Do Big Jobs Automatically," EDN, pp. 45-54.

Hackmeister, Dick. "Focus on Semiconductor RAMs," Electronic Design, August 16, 1977, pp. 56-62. An excellent survey of all types of semiconductor memory available at the time of writing.

Lane, Art. "Microprocessor-System Design," Digital Design, August, 1975, pp. 62-70.

Leventhal, Lance A. "Why Structured Programming?" Kilobaud, February, 1978, pp. 84-88.

_____. "The Top-Down Approach," Kilobaud, May, 1978. These two articles are a clear and concise introduction to the topics of structured programming and top-down design.

"Magnetic Bubble Memories: Past°Present°Future," Digital Design, May, 1977, pp. 50-66. A Benwill/Technocast Report.

Martinez, Ralph. "A Look at Trends in Microprocessor/Microcomputer Software Systems," Computer Design, June, 1975, pp. 51-57.

Metzer, Jerry. "EAROMs," Electronic Products Magazine, September, 1977, pp. 57-60. An explanation of electrically alterable ROM mechanisms and capability, and a survey of devices available at the time of writing.

"Microprocessor Data Manual," Electronic Design, October 11, 1977. pp. 54-190. Multi-section publication including primer article, microprocessor specifications, etc.

Posa, John G. "Pascal Becomes Software Superstar," Electronics, October, 1978, pp. 81-84. Discusses recent progress by both Pascal users and micro- and minicomputer manufacturers who have included this new language in their software repertoire.

"Primer on Microprocessors," Electronic Products Magazine, January 20, 1975, pp. 25-32. A good primer on computer operation basics.

Riley, Thomas P. "A Morse Code to Alphanumeric Converter and Display," QST, October, 1975. Part I of a 3-part series of articles. Includes a discussion of the rules for Morse code communications.

Ulrickson, Robert W. "Real-Time Systems Often Use Interrupts," Electronic Design, May 10, 1977, pp. 80-84.

Vachon, Bradstreet. "Interfacing: A Balancing Act of Hardware and Software," Electronic Design, May 27, 1971, pp. 58-63. Part 4 of a series on "The Minicomputer and the Engineer."

Vacroux, Andre G. "Explore Microcomputer I/O Capabilities," Electronic Design, May 10, 1975, pp. 114-119. A good place to start on microcomputer I/O.

Winfield, Jerry. "Dynamic Memories Offer Advantages," Electronic Design, July 5, 1977, pp. 66-70. A detailed explanation of the design and use of dynamic read-write memory components.

BOOKS

Aron, J. D. The Program Development Process, Addison-Wesley Publishing Company, Reading, Mass., 1974. Addresses fundamental questions regarding the steps in the development of programs.

Bartee, Thomas C. Digital Computer Fundamentals, McGraw-Hill Book Company, New York, 1977. A comprehensive text on computer hardware structures.

Blakeslee, Thomas R. Digital Design with Standard MSI and LSI, John Wiley & Sons, New York, 1975. Includes a discussion of the rationalization for the use of digital circuits for control of analog signals and certain "nasty realities" in hardware development.

Chandor, Anthony, John Graham and Robin Williamson. A Dictionary of Computers, Penguin Books Inc., Baltimore, 1970. An expensive pocket dictionary well worth getting.

Finkel, Jules. Computer-Aided Experimentation: Interfacing to Minicomputers, John Wiley and Sons, New York, 1975. Addresses fundamental interfacing considerations.

Hughes, John K. and Jay I. Michtom. A Structured Approach to Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977. Includes a discussion of both top-down design and structured programming.

Jensen, K. and N. Wirth. Pascal User Manual and Report, Springer-Verlag, 1976. Considered by many to be the best general source on Pascal.

Nicholls, John E. The Structure and Design of Programming Languages, Addison-Wesley Publishing Company, Reading, Mass., 1975. Includes a chapter on the theoretical foundations of structured programming languages.

Ogdin, Carol Anne. Microcomputer Design, Prentice-Hall, Inc., Englewood, Cliffs, N.J., 1978. Contains many good practical examples of microcomputer design.

Osborne, Adam. An Introduction to Microcomputers, Volume 1 Basic Concepts, Adam Osborne and Associates, Incorporated, Berkeley, Ca., 1976. This is one of a number of introductory books on microprocessors and microcomputers.

Peatman, John B. Microcomputer-Based Design, McGraw-Hill, Inc., 1977. Includes, among other things, a comprehensive discussion of practical I/O interfacing considerations.

Sippl, Charles J. and Charles P. Sippl. Computer Dictionary and Handbook, Howard W. Sams & Co., Inc., Indianapolis, 1972 (Second Edition). Contains not only an extensive dictionary but also a number of appendices on topics such as computer system principles, personnel, management science, number systems, flowcharting, languages, and future of computers.

Struble, George W. Assembler Language Programming: The IBM System/360 and 370, Addison-Wesley Publishing Company, Reading, Mass., 1975. Includes many good examples and exercises on assembler language programming in relation to the IBM 360 and 370 computers.

Wester, John G. and William D. Simpson. Software Design for Microprocessors, Texas Instruments Inc., 1976. Contains an example of a 9900-based application.

Zaks, Rodnay. Microprocessors - From Chips to Systems, Sybex Inc., Berkeley, 1977 (Second Edition). A general treatment of microprocessors with emphasis on 8-bit devices.

MANUFACTURER'S MANUALS, PUBLICATIONS

The BiPolar Microcomputer Components Data Book for Design Engineers, Texas Instruments Incorporated, Semiconductor Components Group, publication LCC4270A, December, 1977. An excellent reference on bipolar memory components.

990 Computer Family Systems Handbook, Manual No. 945250-9701, 3rd Edition, May, 1976, Texas Instruments Incorporated.

"Magnetic Bubble Memories and System Interface Circuits from Texas Instruments," February, 1977, Texas Instruments Incorporated. A detailed treatment on design of memory systems using TI bubble memory technology.

Model 990 Computer TI Pascal User's Manual, Texas Instruments, Incorporated, 1978.

Model 990 Computer TMS 9900 Emulator and Buffer Modules--Installation and Operations Manual, Manual No. 946245-9701, 15 June, 1977, Texas Instruments Incorporated, Digital Systems Division. Provides a general description of the AMPL Microprocessor Prototyping Lab and its installation, programming, and operation.

Model 990 Computer TMS 9900 Microprocessor: Assembly Language Programmer's Guide, Manual No. 94341-9701, Texas Instruments Incorporated. The basic reference on 9900 assembly language.

MOS Memory Data Book for Design Engineers, Texas Instruments Incorporated, Semiconductor Group Publication X018C, 1978. A reference source for all types of TI MOS memory components.

Simpson, William D., Gerald Luecke, et al, 9900 Family Systems Design and Data Book, Texas Instruments Incorporated, Houston, 1978. A primary reference for design and programming activities for the 9900 family of devices.

Sitrick, David H. "Memory System Design Utilizing TMS 4050/4051 4K Dynamic RAMs," A Texas Instruments Application Report, Bulletin MOSA 1, 1976. A detailed report covering all aspects of dynamic memory system design and cost analysis, including typical system schematic diagrams.

TM 990/189 Microcomputer User's Guide, Texas Instruments Incorporated, Semiconductor Group, January, 1979. A must for user's of the TM 990/189 board.

"TMS 9900 System Development Manual," A Texas Instruments Application Report, Bulletin MOSA1, September, 1976.

"TMS 9901 Programmable Systems Interface Data Manual," Texas Instruments Incorporated, Semiconductor Group, July, 1977.

"TMS 9902 Asynchronous Communications Controller Data Manual," Texas Instruments Incorporated, July, 1978.

"TMS 9980A/TMS 9981 Microprocessor Data Manual," November, 1977, Texas Instruments Incorporated, Semiconductor Group.

The TTL Data Book for Design Engineers, Second Edition, Texas Instruments Incorporated, Semiconductor Group, 1976.

SPECIAL PUBLICATIONS

Microprocessor Design Series, EDN. A collection of articles from EDN magazine, first issue published in 1974.

Altman, Laurence, editor. Microprocessors, McGraw-Hill Publications Co., Electronics Book Series, 1975. A collection of articles from Electronics magazine.

_____. Applying Microprocessors: New Hardware, Software, and Applications, McGraw-Hill Publications Co., Electronics Book Series, 1977. Another collection of articles from Electronics magazine.

Bursky, Dave, editor. Microprocessor Data Manual, Hayden Book Company, Inc., Rochelle Park, 1978. Contains a number of data sheets on microprocessors together with a small selection of articles from Electronic Design magazine.

EDN, Special Issue, November 20, 1978. An issue published annually containing a microprocessor directory, systems directory, support chip directory, etc.

Torreno, Edward A., editor, Microprocessors: New Directions for Designers, Hayden Book Company, Inc., Rochelle Park, 1975. A selection of articles from Electronic Design magazine.

APPENDIX A

GLOSSARY

access time
The time interval between the request for information and the instant this information is available.

accounting machine
1. A keyboard actuated machine that prepares accounting records.
2. A machine that reads data from external storage media, such as cards or tapes, and automatically produces accounting records or tabulations, usually on continuous forms.

accumulator
A device which stores a number and which, on receipt of another number, adds the two and stores the sum.

accuracy
The degree of freedom from error, that is, the degree of conformity to truth or to a rule. Accuracy is contrasted with precision. For example, four-place numerals are less precise than six-place numerals, nevertheless a properly computed four-place numeral might be more accurate than an improperly computed six-place numeral.

ACK
Acknowledge message sent upon a communication link to indicate the reception of correct data. Used with error detectors in block and tree codes.

adder
Switching circuit that combines binary bits to generate the Sum and Carry of these bits.

address
An expression, usually numerical, which designates a specific location in a storage or memory device.

address format
1. The arrangement of the address parts of an instruction. The expression "plus-one" is frequently used to indicate that one of the addresses specifies the location of the next instruction to be executed, such as one-plus-one, two-plus-one, three-plus-one, four-plus-one.
2. The arrangement of the parts of a single address, such as those required for identifying channel, module, track, etc., in a disc system.

address register

A register in which an address is stored.

algorithm

A term used by mathematicians to describe a set of procedures by which a given result is obtained.

alphanumeric code

A code whose code set consists of letters, digits, and associated special characters.

ALU

Arithmetic Logic Unit, a computational subsystem which performs the mathematical operations of a digital system.

analog representation

A representation that does not have discrete values but is continuously variable.

arithmetic shift

1. A shift that does not affect the sign position.
2. A shift that is equivalent to the multiplication of a number by a positive or negative integral power of the radix.

array logic

A logic network whose configuration is a rectangular array of intersections of its input-output leads, with elements connected at some of these intersections. The network usually functions as an encoder or decoder.

ASCII

American National Standard Code for Information Interchange. The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. Synonymous with USASCII.

assemble

To prepare a machine language program from a symbolic language program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assembler

A computer program that assembles.

asynchronous device

A device in which the speed of operation is not related to any frequency in the system to which it is connected.

baud

A unit of signaling speed equal to the number of discrete conditions or signal events per second. For example, one baud equals one-half dot cycle per second in Morse code, one bit per second in a train of binary signals, and one 3-bit value per second in a train of signals each of which can assume one of eight different states.

Baudot code

Information code used in data transmission.

benchmark problem

A problem used to evaluate the performance of hardware or software software or both.

binary coded decimal (BCD)

A binary numbering system for coding decimal numbers in groups of 4 bits. The binary value of these 4-bit groups ranges from 0000 to 1001, and codes the decimal digits "0" through "9". To count to 9 takes 4 bits; to count to 99 takes two groups of 4 bits; to count to 999 takes three groups of 4 bits, etc.

block

1. A set of things, such as words, characters, or digits handled as a unit.
2. A collection of contiguous records recorded as a unit. Blocks are separated by block gaps and each block may contain one or more records.
3. A group of bits, or n-ary digits, transmitted as a unit. An encoding procedure is generally applied to the group of bits or n-ary digits for error-control purposes.
4. A group of contiguous characters recorded as a unit.

block diagram

A diagram of a system, instrument, or computer in which the principal parts are represented by suitable associated geometrical figures to show both the basic functions and the functional relationships among the parts.

bootstrap

A technique or device designed to bring itself into a desired state by means of its own action, e.g., a machine routine whose first few instructions are sufficient to bring the rest of itself into the computer from an input device.

borrow

An arithmetically negative carry.

branching

A method of selecting, on the basis of results, the next operation to execute while the program is in progress.

buffer

An isolating circuit used to avoid reaction of a driven circuit on the corresponding driver circuit. Also, a storage device used to compensate for a difference in the rate of flow of information or the time of occurrence of events when transmitting information from one device to another.

bus

One or more conductors used for transmitting signals or power.

byte

A sequence of adjacent binary digits operated upon as a unit and usually shorter than a computer word.

calculator

1. A data processor especially suitable for performing arithmetical operations which requires frequent intervention by a human operator.
2. Generally and historically, a device for carrying out logic and arithmetic digital operations of any kind.

call

1. To transfer control to a specified closed subroutine.
2. In communications, the action performed by the calling party, or the operations necessary in making a call, or the effective use made of a connection between two stations.

carry

1. One or more digits, produced in connection with an arithmetic operation on one digit place of two or more numerals in positional notation, that are forwarded to another digit place for processing there.
2. The number represented by the digit or digits in definition 1 above.
3. Most.commonly, a digit as defined in definition 1 above that arises when the sum or product of two or more digits equals or exceeds the radix of the number representation system.
4. Less commonly, a borrow.
5. To forward a carry.
6. The command directing that a carry be forwarded.

carry look-ahead

A type of adder in which the inputs to several stages are examined and the proper carries are produced simultaneously.

cascade connection

Two or more similar component devices arranged in tandem, with the output of one connected to the input of the next.

central processor unit (CPU)

Part of a computer system which contains the main storage, arithmetic unit, and special register groups. It performs arithmetic operations, controls instruction processing, and provides timing signals and other housekeeping operations.

channel

1. A path along which signals can be sent, e.g., data channel, output channel.
2. The portion of a storage medium that is accessible to a given reading or writing station, e.g., track, band.
3. In communications, a means of one-way transmission. Several channels may share common equipment. For example, in frequency multiplexing carrier systems, each channel uses a particular frequency band that is reserved for it.

character

A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes.

check bit

A binary check digit, e.g., a parity bit.

circuit

In communications, a means of two-way communication between two points, comprising associated "go" and "return" channels.

code

1. A set of unambiguous rules specifying the way in which data may be represented, e.g., the set of correspondences in the standard code for information interchange. Synonymous with coding scheme.
2. In telecommunications, a system of rules and conventions according to which the signals representing data can be formed, transmitted, received, and processed.
3. In data processing, to represent data or a computer program in a symbolic form that can be accepted by a data processor.

combinatorial logic system

Digital system *not* utilizing memory elements.

communication control character

A control character intended to control or facilitate transmission of data over communication networks.

communication link

The physical means of connecting one location to another for the purpose of transmitting and receiving data.

compile

To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one machine instruction for each symbolic statement, or both, as well as performing the function of an assembler.

compiler

A program that compiles.

complement notation

A system of notation where positive binary numbers are identical to positive numbers in sign and magnitude notation, but where negative numbers are the exact complement of the magnitude of the corresponding positive value.

conditional jump

A jump that occurs if specified criteria are met.

control character

A character whose occurrence in a particular context initiates, modifies, or stops a control operation, e.g., a character that controls carriage return, a character that controls transmission of data over communication networks. A control character may be recorded for use in a subsequent action. It may in some circumstances have a graphic representation.

control hierarchy

Design development used in complex systems to ensure an order of priority to several controls coming from more than one source.

controller

Digital subsystem responsible for implementing "how" a system is to function. Not to be confused with "timing" as timing tells the system "when" to perform its function.

counter

A circuit which counts input pulses and will give an output pulse after receiving a predetermined number of input pulses.

CRC

The Cyclic Redundancy Check character.

critical race

Timing situation related to asynchronous operation. A "race" can occur when two variables are asked to change states simultaneously. "Critical" refers to the outcome that will determine the state of the machine.

crosstalk

Interference which appears in a given channel but has its origin in another channel.

cycle

1. An interval of space or time in which one set of events or phenomena is completed.
2. Any set of operations that is repeated regularly in the same sequence. The operations may be subject to variations on each repetition.

data

1. A representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or automatic means.
2. Any representations such as characters or analog quantities to which meaning is or might be assigned.

data bus

One method of input-output for a system where data are moved into or out of the digital system by way of a common bus connected to several subsystems.

data processing

The execution of a systematic sequence of operations performed upon data. Synonymous with information processing.

data processor

A device capable of performing data processing, including desk calculators, punched card machines, and computers. Synonymous with processor.

debug

To detect, locate, and remove mistakes from a routine or malfunctions from a computer. Synonymous with troubleshoot.

decimal

1. Pertaining to a characteristic or property involving a selection, choice, or condition in which there are ten possibilities.
2. Pertaining to the number representation system with a radix of ten.

decimal digit

In decimal notation, one of the characters 0 through 9.

decoder

A conversion circuit that accepts digital input information – in the memory case, binary address information – that appears as a small number of lines and selects and activates one line of a large number of output lines.

device control character

A control character intended for the control of ancillary devices associated with a data processing or telecommunication system, usually for switching devices "on", or "off".

diagnostic

Pertaining to the detection and isolation of a malfunction or mistake.

digit

A symbol that represents one of the non-negative integers smaller than the radix. For example, in decimal notation, a digit is one of the characters from 0 to 9. Synonymous with numeric character.

digitize

To use numeric characters to express or represent data, e.g., to obtain from an analog representation of a physical quantity, a digital representation of the quantity.

direct access

1. Pertaining to the process of obtaining data from, or placing data into, storage where the time required for such access is independent of the location of the data most recently obtained or placed in storage.
2. Pertaining to a storage device in which the access time is effectively independent of the location of the data.
3. Synonymous with random access.

direct addressing

Method of programming that has the address pointing to the location of data or the instruction that is to be used.

direct memory access channel (DMA)

A method of input-output for a system that uses a small processor whose sole task is that of controlling input-output. With DMA, data are moved into or out of the system without program intervention.

dot matrix

A matrix of dots that is used to identify alphanumeric characters.

double precision

Pertaining to the use of two computer words to represent a number.

dump

1. To copy the contents of all or part of a storage, usually from an internal storage into an external storage.
2. A process as in definition 1 above.
3. The data resulting from the process as in definition 1 above.

duplex
The method of operation of a communication circuit in which each end can simultaneously transmit and receive.

dynamic storage elements
Storage elements which contain storage cells that must be refreshed at appropriate time intervals to prevent the loss of information content.

EBCDIC
Extended Binary Coded Decimal Interchange Code. An 8-bit, 256-character code used in transmission of binary data.

ECL circuits
Bipolar emitter-coupled logic circuits, also called current-mode logic circuits.

edge triggering
Activation of a circuit at the edge of the pulse as it begins its change. Circuits then trigger at the edge of the input pulse rather than sensing a level change.

edit
To modify the form or format of data, e.g., to insert or delete characters such as page numbers or decimal points.

electrostatic storage
A storage device that stores data as electrostatically charged areas on a dielectric surface.

emulate
To imitate one system with another such that the imitating system accepts the same data, executes the same programs, and achieves the same results as the imitated system.

encode
To apply a set of unambiguous rules specifying the way in which data may be represented such that a subsequent decoding is possible. Synonymous with code.

end-around carry
A carry generated in the most significant digit place and sent directly to the least significant place.

entry point
In a routine, any place to which control can be passed.

erase
To obliterate information from a storage medium, e.g., to clear, to overwrite.

error

Any discrepancy between a computed, observed, or measured quantity and the true, specified, or theoretically correct value or condition.

exclusive-OR function

A modified form of the OR function which has a logic equation equal to the sum output of the half-adder.

execute

That portion of a computer cycle during which a selected control word or instruction is accomplished.

exponent

In a floating point representation, the numeral, of a pair of numerals representing a number, that indicates the power to which the base is raised.

fan-out

The number of loads connected to the output of a logic stage. (A load normally consists of the input impedance of a logic circuit.)

feedback loop

The components and processes involved in correcting or controlling a system by using part of the output as input.

feedback system

See **information feedback system.**

fetch

That portion of a computer cycle during which the next instruction is retrieved from memory.

field

In a record, a specified area used for a particular category of data, e.g., a group of card columns used to represent a wage rate, a set of bit locations in a computer word used to express the address of the operand.

fixed-point binary number

A binary number represented by a sign bit and one or more number bits, with a binary point fixed somewhere between two neighboring bits.

flag

1. Any of various types of indicators used for identification, e.g., a wordmark.
2. A character that signals the occurrence of some condition, such as the end of a word.
3. Synonymous with mark, sentinel, tag.

flip-flop (storage element)

A circuit having two stable states and the capability of changing from one state to another with the application of a control signal and remaining in that state after removal of signals.

floating-point binary number

A binary number expressed in exponential notation. That is, a part of the binary word represents the mantissa and a part the exponent.

flow chart

A graphical representation for the definition, analysis, or solution of a problem, in which symbols are used to represent operations, data, flow, equipment, etc.

format

The arrangement of data.

FORTRAN

(FORmula TRANslating system) A language primarily used to express computer programs by arithmetic formulas.

full-adder

A logic circuit like the half-adder, but with a provision for a carry-in from a preceding addition.

function

1. A specific purpose of an entity, or its characteristic action.
2. In communications, a machine action such as a carriage return or line feed.

general-purpose computer

A computer that is designed to handle a wide variety of problems.

half-adder

A logic circuit capable of adding two binary numbers with no provision for a carry-in from a preceding addition.

hamming code

An error correction code system used in data transmission. This code uses a parity check matrix in its operation.

hardware

Physical equipment, as opposed to the computer program or method of use, e.g., mechanical, magnetic, electrical, or electronic devices.

hazard

Transient output of a circuit that allows an undesired output value to appear during transition from one state to another.

immediate address

Pertaining to an instruction in which an address part contains the value of an operand rather than its address. Synonymous with zero-level address.

indexed address

An address that is modified by the content of an index register prior to or during the execution of a computer instruction.

indexing

In computers, a method of address modification that is implemented by means of index registers.

index register

A register whose content may be added to or subtracted from the operand address prior to or during the execution of a computer instruction. Synonymous with b box.

indirect addressing

Programming method that has the initial address being the storage location of a word that contains another address. This indirect address is then used to obtain the data to be operated upon.

information feedback system

In telecommunications, an information transmission system that uses an echo check to verify the accuracy of the transmission.

input/output devices (I/O)

Computer hardware by which data is entered into a digital system or by which data are recorded for immediate or future use.

instruction

A statement that specifies an operation and the values or locations of its operands.

instruction counter

A counter that indicates the location of the next computer instruction to be interpreted.

instruction register

A register that stores an instruction for execution.

interface
A shared boundary. An interface might be a hardware component to link two devices or it might be a portion of storage or registers accessed by two or more computer programs.

interleave (or interlace)
To assign successive storage location numbers to physically separated memory storage locations. This serves to reduce access time.

interrupt
To stop a process in such a way that it can be resumed.

jump
A departure from the normal sequence of executing instructions in a computer.

jump conditions
Conditions defined in a transition table that determine the changes of flip-flops from one state to another state.

label
One or more characters used to identify a statement or an item of data in a computer program.

language
A set of representations, conventions, and rules used to convey information.

large-scale integration (LSI)
The simultaneous realization of large-area chips and optimum component packing density, resulting in cost reduction by maximizing the number of system connections done at the chip level. Circuit complexity above 100 gates.

level
The degree of subordination in a hierarchy.

light pen
A small photocell on photomultiplier in a pen-shaped housing which is held against a CRT screen so as to detect the instant the electron beam goes through that particular location during its scanning sweep.

logic shift
A shift that affects all positions.

loop
A sequence of instructions that is executed repeatedly until a terminal condition prevails.

machine code

An operation code that a machine is designed to recognize.

machine language

A language that is used directly by a machine.

macroinstruction

An instruction in a source language that is equivalent to a specified sequence of machine instructions.

macroprogramming

Programming with macroinstructions.

main frame

Same as **central processing unit**.

mask

1. A pattern of characters that is used to control the retention or elimination of portions of another pattern of characters.
2. A filter.

matrix

1. In mathematics, a two-dimensional rectangular array of quantities. Matrices are manipulated in accordance with the rules of matrix algebra.
2. In computers, a logic network in the form of an array of input leads and output leads with logic elements connected at some of their intersections.
3. By extension, an array of any number of dimensions.

microprogramming

Control technique used to implement the stored program control function. Typically the technique is to use a preprogrammed read-only memory chip to contain several control sequences which normally occur together.

mnemonic symbol

A symbol chosen to assist the human memory, e.g., an abbreviation such as "mpy" for "multiply".

MODEM

(MOdulator-DEModulator) A device that modulates and demodulates signals transmitted over communication facilities.

MOS transistor (metal-oxide-semiconductor transistor)

An active semiconductor device in which a conducting channel is induced in the region between two electrodes by a voltage applied to an insulated electrode on the surface of the region.

multiplex
To interleave or simultaneously transmit two or more messages on a single channel.

NAK
Message sent over data communications links to indicate that received data has been checked, is incorrect, and should be retransmitted.

negative logic
Logic in which the more-negative voltage represents the "1" state; the less-negative voltage represents the "0" state.

noise
A term referring to spurious or undesirable electrical signals.

noise immunity
A measure of the insensitivity of a logic circuit to triggering or reaction to spurious or undesirable electrical signals or noise, largely determined by the signal swing of the logic. Noise can occur in either of two directions, positive or negative.

nondestructive read out
A memory designed so that read-out does not affect the content stored. It is not necessary to perform a write after every read operation.

numerical control
Automatic control of a process performed by a device that makes use of all or part of numerical data generally introduced as the operation is in process.

object code
Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

object language
The language to which a statement is translated.

OCR
Optical character recognition.

operand
That which is operated upon. An operand is usually identified by an address part of an instruction.

operating system
Software which controls the execution of computer programs and which may provide scheduling, debugging, input/output control, accounting, compilation, storage assignment, data management, and related services.

operation

1. A defined action, namely, the act of obtaining a result from one or more operands in accordance with a rule that completely specifies the result for any permissible combination of operands.
2. The set of such acts specified by such a rule, or the rule itself.
3. The act specified by a single computer instruction.
4. A program step undertaken or executed by a computer, e.g., addition, multiplication, extraction, comparison, shift, transfer. The operation is usually specified by the operator part of an instruction.
5. The even or specific action performed by a logic element.

operation code

A code that represents specific operations. Synonymous with instruction code.

optical character recognition

The machine identification of printed characters through use of light-sensitive devices. Abbreviated OCR.

pack

To compress data in a storage medium by taking advantage of known characteristics of the data, in such a way that the original data can be recovered, e.g., to compress data in a storage medium by making use of bit or byte locations that would otherwise go unused.

parallel operation

The organization of data manipulating within circuitry wherein all the digits of a word are transmitted simultaneously on separate lines in order to speed up operation.

parameter

A variable that is given a constant value for a specific purpose or process.

parity bit

A check bit appended to an array of binary digits to make the sum of all the binary digits, including the check bit, always odd or always even.

parity check

The technique of adding one bit to a digital word to make the total number of binary ones or zeros either always even or always odd. This type of checking will indicate an error in data but will not indicate the location of the error.

peripheral equipment

Units which work in conjunction with a computer but are not part of it.

PLA (programmable logic array)

An integrated circuit that employs ROM matrices to combine sum and product terms of logic networks.

positive logic

Logic in which the more positive voltage represents the "1" state; the less positive voltage represents the "0" state.

priority interrupt

Designation given to method of providing some commands to have precedence over others thus giving one condition of operation priority over another.

problem oriented language

A programming language designed for the convenient expression of a given class of problems.

processor

1. In hardware, a data processor.
2. In software, a computer program that includes the compiling, assembling, translating, and related functions for a specific programming language, COBOL processor, or FORTRAN processor.

program

1. A series of actions proposed in order to achieve a certain result.
2. Loosely, a routine.
3. To design, write, and test a program as in definition 1 above.
4. Loosely, to write a routine.

programmable read only memory (PROM)

A fixed program, read only, semiconductor memory storage element that can be programmed after packaging.

PROM

See **programmable read only memory**.

propagation delay

The time required for a change in logic level to be transmitted through an element or a chain of elements.

pushdown list

A list that is constructed and maintained so that the item to be retrieved is the most recently stored item in the list, i.e., last in, first out.

pushdown stack

A register which implements a pushdown list.

pushup list

A list that is constructed and maintained so that the next item to be retrieved and removed is the oldest item still in the list, i.e., first in, first out.

RAM

See **random access memory**.

random access memory (RAM)

A memory from which all information can be obtained at the output with approximately the same time delay by choosing an address randomly and without first searching through a vast amount of irrelevant data.

read only memory (ROM)

A fixed program semiconductor storage element that has been preprogrammed at the factory with a permanent program.

real time

1. Pertaining to the actual time during which a physical process transpires.
2. Pertaining to the performance of a computation during the actual time that the related physical process transpires, in order that results of the computation can be used in guiding the physical process.

redundancy

The technique of using more than one circuit of the same type to implement a given function.

refresh

Method which restores charge on capacitance which deteriorates because of leakage.

register

Temporary storage for digital data.

relative address

The number that specifies the difference between the absolute address and the base address.

ROM

See **read only memory**.

sample-and-hold circuit

A circuit that performs the operation of looking at a voltage level during a short time period and accurately storing that voltage level for a much longer time period.

scratch-pad memory

A small local memory utilized to facilitate local data handling on a temporary basis.

sequencing
Control method used to cause a set of steps to occur in a particular order.

sequential logic systems
Digital system utilizing memory elements.

serial accumulator
A register which receives data bits in serial or sequence and temporarily holds the data for future use.

serial operation
The organization of data manipulation within circuitry wherein the digits of a word are transmitted one at a time along a single line. The serial mode of operation is slower than parallel operation, but utilizes less complex circuitry.

set-up time
The minimum amount of time that data must be present at an input to ensure data acceptance when the device is clocked.

shift
A movement of data to the right or left.

shift register
A register in which the stored data can be moved to the right or left.

sign and magnitude notation
A system of notation where binary numbers are represented by a sign bit and one or more number bits.

silo memory
Reads out stored data in a first-in/first-out mode. Also known as FIFO.

simulate
1. To represent certain features of the behavior of a physical or abstract system by the behavior of another system.
2. To represent the functioning of a device, system, or computer program by another, e.g., to represent the functioning of one computer by another, to represent the behavior of a physical system by the execution of a computer program, to represent a biological system by a mathematical model.

simulator
A device, system, or computer program that represents certain features of the behavior of a physical or abstract system.

skip
To ignore one or more instructions in a sequence of instructions.

software
A set of computer programs, procedures, and possibly associated documentation concerned with the operation of a data processing system, e.g., compilers, library routines, manuals, circuit diagrams.

source language
The language from which a statement is translated.

source program
A computer program written in a source language.

state
The condition of an input or output of a circuit as to whether it is a logic "1" or a logic "0". The state of a circuit (gate or flip-flop) refers to its output. A flip-flop is said to be in the "1" state when its Q output is "1". A gate is in the "1" state when its output is "1".

static storage elements
Storage elements which contain storage cells that retain their information as long as power is applied unless the information is altered by external excitation.

stored program
A set of instructions in memory specifying the operation to be performed.

subroutine
A routine that can be part of another routine.

synchronous circuit
A circuit in which all ordinary operations are controlled by equally spaced signals from a master clock.

system
1. An assembly of methods, procedures, or techniques united by regulated interaction to form an organized whole.
2. An organized collection of men, machines, and methods required to accomplish a set of specific functions.

table look-up
A procedure for obtaining the function value corresponding to an argument from a table of function values.

telecommunications
Pertaining to the transmission of signals over long distances, such as by telegraph, radio, or television.

temporary storage
In programming, storage locations reserved for intermediate results. Synonymous with working storage.

terminal
A point in a system or communication network at which data can either enter or leave.

transfer
Same as jump.

translate
To transform statements from one language to another without significantly changing the meaning.

truth table
A chart that tabulates and summarizes all the combinations of possible states of the inputs and outputs of a circuit. It tabulates what will happen at the output for a given input combination.

TTL
Bipolar semiconductor transistor-transistor coupled logic circuits.

2's complement notation
A system of notation where positive binary numbers are identical to positive numbers in sign and magnitude notation, but where 1 must be added to 1's complement notation to obtain negative numbers.

USASCII
United States of America Standard Code for Information Interchange. The standard code used by the United States for transmission of data. Sometimes simply referred to as the "as'ki" code.

variable
A quantity that can assume any of a given set of values.

volatile storage
A storage device in which stored data are lost when the applied power is removed.

word
A character string or a bit string considered as an entity.

write enable

Also called read/write or R/W. The control signal to a storage element or a memory that activates the write mode or operation. Conversely when not in the write mode, the read mode is active.

write time

The time that the appropriate level must be maintained on the write-enable line and that data must be present to guarantee successful writing of data in the memory.

A P P E N D I X B

ANSWERS TO ODD EXERCISES

Chapter 2

1. (a) $1357_{10} = [1x(10^3)]+[3x(10^2)]+[5x(10^1)]+[7x(10^0)]$

   (b) $409_{10} = [4x(10^2)]+[0x(10^1)]+[9x(10^0)]$

   (c) $5_{10} = [5x(10^0)]$

3. (a) $1011_2 = 11_{10}$

   (b) $11111111_2 = 255_{10}$

   (c) $10000000_2 = 128_{10}$

5. (a) $8_{10} = 10_8$

   (b) $29_{10} = 35_8$

   (c) $321_{10} = 470_8$

7. (a) $010\ 110\ 010_2 = 262_8$

   (b) $011\ 001\ 011_2 = 313_8$

   (c) $101_2 = 5_8$

9. (a) $32_{10} = 20_{16}$

   (b) $128_{10} = 80_{16}$

   (c) $300_{10} = 12C_{16}$

11. (a) $1000\ 0001_2 = 81_{16}$

    (b) $1011\ 1111_2 = 5F_{16}$

    (c) $1011_2 = B_{16}$

13. (a) $3.1428_{10} = 11.001001001001\ldots_2$

    (b) $10.125_{10} = 12.1_8$

(c) $1.768_{10} = 1.C49BA_{16}$

15. (a) 17_{10} =

0	--	8	--	0
0	--	4	--	●
0	--	2	--	●
●	--	1	--	●

(b) 80_{10} =

●	--	8	--	0
0	--	4	--	0
0	--	2	--	0
0	--	1	--	0

(c) 61_{10} =

0	--	8	--	0
●	--	4	--	0
●	--	2	--	0
0	--	1	--	●

17. (a) $73_{BCD} = 0111\ 0011_2$

(b) $1.23_{BCD} = 1.0010\ 0011_2$

(c) $14.0_{BCD} = 1110.000_2$

19. (a) 48_{16} = H ; even

(b) 55_{16} = U ; even

(c) 33_{16} = 3 ; even

(d) $7F_{16}$ = DEL; odd

Chapter 3

1. Result: (>300) = >863A
   L> = 1, A> = 0, EQ = 0, CY = 0, OV = 0
   Machine Code: A002

3. Result: (>408) = 0
   L> = 0, A> = 0, EQ = 1
   Machine Code: C554

5. Result: (>40C) = 0
   L> = 0, A> = 0, EQ = 1
   Machine Code: C990
   0100

7. Result: (>30A) = >1234
   L> = 1, A> = 1, EQ = 0
   Machine Code: 0205
   1234

9. Result: (>300) = >0408
   L> = 1, A> = 1, EQ = 0
   Machine Code: 05E0
   0300

11. MOV R2,@0(R3)

13. Yes.
    (R7) = 1
    (R8) = -1
    (R9) = 15
    JMP LP executed 4 times.

Chapter 4

1. LOC. CONTENTS

   0206 1102
   020A 10FC

3. 4 bytes: 0240: >4C41
   0242: >4400
   The resulting location counter will be : 0244
   This result could also be accomplished with:

```
     DATA >4C41,>4400
```

5. Result: (R0) = >0301
   L> = 1, A> = 1, EQ = 0, CY = 0, OV = 0
   Machine Code: B002

7. Result: (>0224) = >0125
   L> = 1, A> = 1, EQ = 0, CY = 0, OV = 0
   Machine Code: B4D2

9. Result: (R1) = >100F
   L> = 1, A> = 1, EQ = 0, CY = 0, OV = 0
   Machine Code: B064
   0224

11. Result: (>0222) = >2368
L> = 1, A> = 1, EQ = 0, CY = 1, OV = 0
Machine Code: 7493

13. Result: (>022A) = >02CD
L> = 1, A> = 1, EQ = 0
Machine Code: D883
0008

15. Result: (>0222) = >0068
L> = 0, A> = 0, EQ = 1, CY = 1, OV = 0
Machine Code: 7492

17. Result: no content change
L> = 0, A> = 0, EQ = 1
Machine Code: 88C0
FFFF

19. Result: no content change
L> = 0, A> = 0, EQ = 0, OP = 1
Machine Code: 90C2

21. Block Move Program:

```
    AORG >200
F1  EQU >300        POINTER TO SOURCE FILE
F2  EQU >380        POINTER TO DEST. FILE
ST  LWPI WS         SET WORKSPACE POINTER
    LI R1,F1        SET POINTER TO SOURCE FILE
    LI R2,F2        SET POINTER TO DEST. FILE
LP  MOV *R1+,*R2+   COPY WORD FROM FILE TO FILE
    CI R1,F1+>80    WAS IT THE LAST WORD?
*
* JLT WORKS ONLY IF SOURCE FILE DOES NOT CONTAIN
* ADDRESS >8000
*
    JLT LP          IF NOT, THEN CONTINUE TRANSFER
    B @>3000        BRANCH TO UNIBUG
WS  BSS 32          SAVE SPACE FOR WORKSPACE
    END ST
```

Chapter 5

1. (a) 32k

   (b) $63E_{16}$

3. 80 ns

5. (a) Any of the following:

>0067, >4867, >4067, >0867

(b) Four ways are given below (using >0067):

(1)
```
LI R4,>0067
SOC R4,R5
```

(2) MS
```
EQU >0067
LI R4,MS
SOC R4,R5
```

(3)
```
ORI R5,>0067
```

(4) MS
```
EQU >0067
ORI R5,MS
```

7. (a) >0000

(b) >F800

(c) >580F

Chapter 6

1. The following are among the reasons why the I/O section is often the most complex part of a processor system.

- The I/O section is usually application-dependent. Whereas the processor-to-memory interface is usually the same regardless of the application, the processor-to-I/O interface is subject to the requirements of the application. Because of this variety, the system designer must normally put more thought into the design of the I/O section.

- The I/O section may be interfacing to non-electronic devices. Whereas the processor-to-memory interface involves electronic components (usually with agreed upon signal levels and timing), the processor-to-I/O interface may involve devices or conditions that are non-electronic (such as thermometers, fluid levels, motor shafts, scales, drill positions, etc.). The signals received from or sent to the non-electronic devices by the processor must be converted to or from electronic form in order for the communication or control to be made. This requires extra effort on the part of the designer.

- Even when signals are converted to electrical form (for example, the temperature of a furnace is converted to an electrically measureable quantity by a thermistor), this signal often must be conditioned to a form suitable for the processor. For example, the output of the thermistor is an analog signal that must be digitized before the processor can utilize the information.

- And even though the signals are converted to digital form, further conditioning is often necessary. For example, the digitized output of a thermistor may have to be scaled and the timing worked out to allow for response times, settling, etc.

3. (a) Interrupt-driven I/O would most likely be used for the sensor since the sensor is used to indicate a situation requiring immediate attention.

   (b) DMA I/O would most likely be used for the high-speed disk since this device is capable of very high speed data transfers.

   (c) Program-controlled I/O would most likely be used to measure the temperature since the temperature changes slowly. The temperature change does not require immediate attention nor is high-speed data transfer required.

5. (a) $(PC) = 2AC_{16}$
   The ONE bit at CRU hardware adress 10_{16} is tested, which sets the EQ status bit to a ONE and results in the JEQ instruction transferring control to location AB.

   (b) $(PC) = 2AC_{16}$
   The same bit is tested as in Exercise 5(a).

   (c) $(PC) = 2AC_{16}$
   The ONE bit at CRU hardware address 28_{16} is tested, which sets the EQ status bit to a ONE and results in the JEQ instruction transferring control to location AB.

   (d) $(PC) = 288_{16}$
   The ONE bit at CRU hardware address 21_{16} is tested, which sets the EQ status bit to a ONE and causes the JNE instruction to <u>not</u> transfer control. Therefore, the PC contains the address of the word following the JNE instruction.

(e) (PC) = 288_{16}
The ONE bit at CRU hardware address 24_{16} (21_{16} + 3 = 24_{16}) is tested, which sets the EQ status bit to a ONE and causes the JNE instruction to not transfer control. Therefore, the PC contains the address of the word following the JNE instruction.

(f) (PC) = 288_{16}
The ZERO bit at CRU hardware address $1E_{16}$ (21_{16} - 3 = $1E_{16}$) is tested, which sets the equal status bit to ZERO and causes the JEQ instruction to not transfer control. Therefore, the PC contains the address of the word following the JEQ instruction.

(g) (PC) = 288_{16}
The ONE bit at CRU hardware address 26_{16} (10_{16} + 16_{16} = 26_{16}) is tested, which sets the EQ status bit to a ONE and causes the JNE instruction to not transfer control. Therefore, the PC contains the address of the word following the JNE instruction.

(h) (PC) = 288_{16}
The ZERO bit at CRU hardware address 11_{16} (24_{16} - 19_{10} = 24_{16} - 13_{16} = 11_{16}) is tested, which resets the EQ status bit to ZERO and causes the JEQ instruction to not transfer control. Therefore, the PC contains the address of the word following the JEQ instruction.

7. (a) (R6) = 0000 0001 0100 1101_2 = $014D_{16}$

(b) (R6) = 0000 0001 0100 1101_2 = $014D_{16}$

(c) (R6) = 1001 1101 0100 1101_2 = $9D4D_{16}$

(d) (R6) = 0000 1001 1101 0100_2 = $09D4_{16}$

(e) (R6) = 0001 0100 1111 0111_2 = $14F7_{16}$

(f) (R6) = 0000 0010 1101 0011_2 = $02D3_{16}$

9.
```
LI      R12,>280        SET CRU S/W BASE ADRS
LI      R2,FD           PUT ADRS OF BITS IN R2
LDCR    *R2+,0          OUTPUT FIRST 16 BITS
AI      R12,32          ADJUST BASE ADRS
LDCR    *R2+,0          OUTPUT SECOND 16 BITS
AI      R12,32          ADJUST BASE ADRS
LDCR    *R2,8           OUTPUT LAST 8 BITS
```

11. The primary advantage of parallel data transfer is that several bits of information may be transmitted at the same time. For example, parallel communication is usually used to transmit data from memory to the processor in a microprocessor system.

Chapter 7

1. (a)
```
MOV   @>F004,R1
ANDI  R1,>0200
```

(b)
```
ORI   R1,>1C00
MOV   R1,@>F004
```

(c)
```
LI    R1,1
SOC   R1,@>F004
SZC   R1,@>F004
```

3. 0.36 seconds

5. (a) >000F, error = 4.2%

(b) >009C, error = 0.2%

(c) >0001, error = 56%

7.
```
SBO   31
LI    R1,>C000   or  >C800
LI    R2,>62B    or  >5A1
LDCR  R1,8
SBZ   13
LDCR  R2,12
```

Chapter 8

1. The concept of program modularity includes the organization of the individual functions of an application into separate program modules. In many cases, a specific function is required at many points in the overall application. In such cases, it is usually advantageous to make the function a subroutine.

3. A subroutine should be restricted to a specific function primarily to reduce the debugging effort. By restructuring a subroutine to one function, the subroutine may be tested and verified more easily.

5. Both the BLWP and the XOP instruction cause a context switch. The BLWP instruction can define a two-word vector anywhere in the memory space, whereas the XOP instruction has a defined area of memory for its vectors (locations 40_{16} through $7F_{16}$). Also, the XOP instruction automatically passes a data item to the called program (in the new workspace register 11) whereas the BLWP instruction does not do this.

7. (WP) = $21EC_{16}$
   (PC) = 3870_{16}
   (WP) = 1002_{16}

9. The value 101_2 (5) must appear on input lines IC0, IC1, and IC2 to request a level-3 interrupt.

11.
```
SR    MOV    *R14+,R2    PUT ADRS OF FIRST VALUE IN R2
      MOV    *R14+,R3    PUT ADRS OF SECOND VALUE IN R3
      MOV    *R2,R0      PUT FIRST VALUE IN R0
      A      *R3,R0      ADD SECOND VALUE TO FIRST
      JNO    DN          JMP IF NO OVERFLOW
      JMP    ON          OVERFLOW-ALREADY SET UP FOR ERROR
                         RETURN
DN    INCT   R14         NO OVERFLOW-SET UP GOOD RETURN
ON    MOV    R0,*R13     PUT ANSWER IN CALLING PROGRAM'S R0
      RTWP               RETURN
```

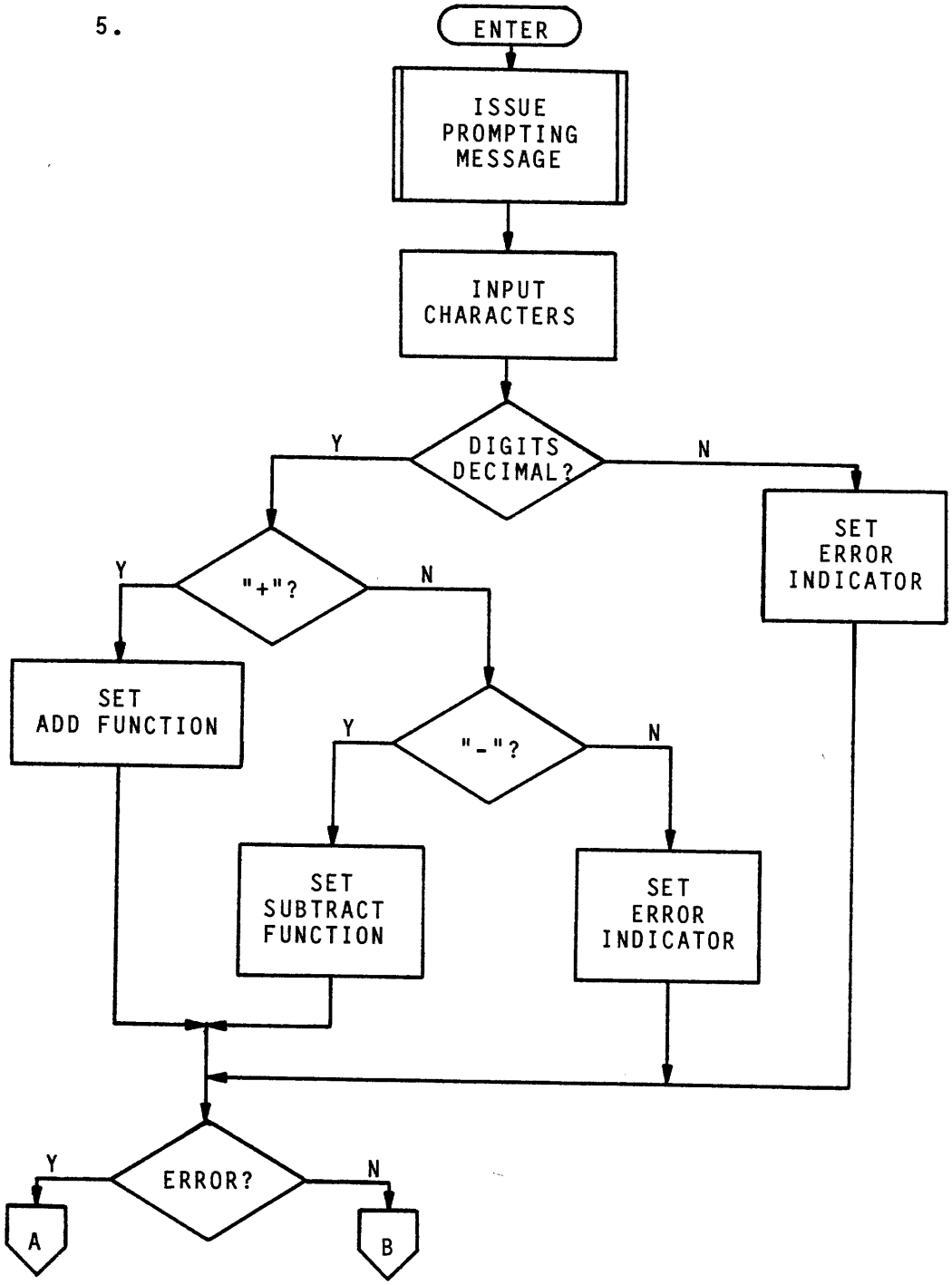
13. The low-order four bits of the status register would still contain the value D_{16}. The XOP instruction does not affect the interrupt mask (the low-order four bits of the status register).

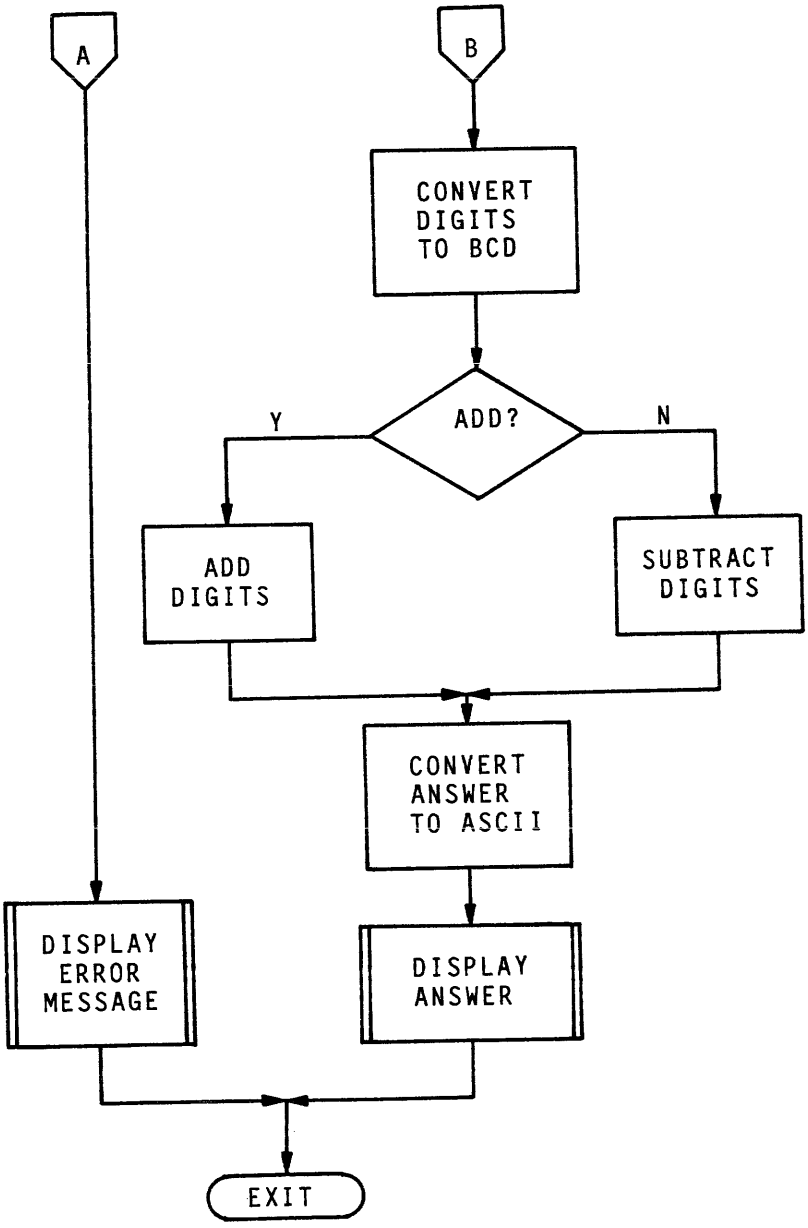
Chapter 9

1. Usually, a software approach to a system function reduces the production cost and produces the most design flexibility when compared to a hardware approach.

3. When compared to high-level language, assembler language normally produces a faster execution (run-) time, but a slower development time.

5.


ENTER
ISSUE PROMPTING MESSAGE
INPUT CHARACTERS
DIGITS DECIMAL?
Y
N
SET ERROR INDICATOR
"+"?
Y
N
SET ADD FUNCTION
"-"?
Y
N
SET SUBTRACT FUNCTION
SET ERROR INDICATOR
ERROR?
Y
N
A
B


A
B
CONVERT DIGITS TO BCD
ADD?
Y
N
ADD DIGITS
SUBTRACT DIGITS
CONVERT ANSWER TO ASCII
DISPLAY ERROR MESSAGE
DISPLAY ANSWER
EXIT

7. (a) The TODCLK program utilizes the cycle generator program from Chapter 6 as a subroutine. The cycle generator program occupies memory location 200_{16} though 247_{16} (including a workspace register area and after changing the last instructor to B *R11). The next available location for the TODCLK program is location 248_{16}, but the designer chose location 250_{16} (rounded up to the next 16-byte boundary) to allow a small amount of expansion area between the two programs.

(b) The entry point of the cycle generator program is identified to the TODCLK program by the

```
WE    EQU   >220
```

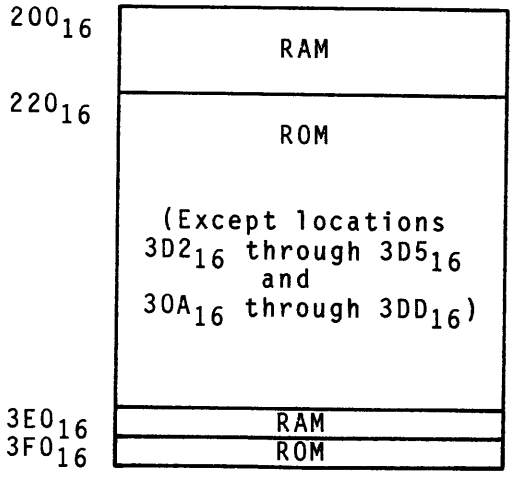
statement.

(c) XOP instructions are used to link this program with the user-accessible utility functions in UNIBUG. The initialization of the contents of the XOP vectors is a part of the monitor's overall initialization functions.

(d) The level-four interrupt service routine increments memory location TK to communicate to the main program that a clock tick has occurred.

9.

M E M O R Y M A P

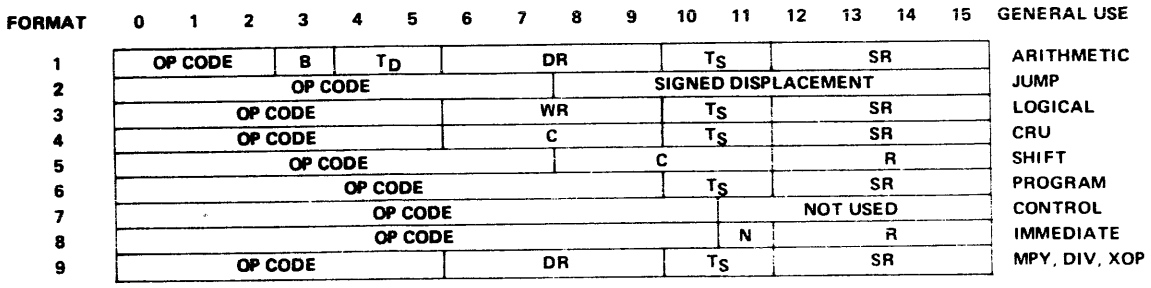


11.

```
        AORG    4
        DATA    W1              ESTABLISH VECTOR VALUES
        DATA    P1                FOR LEVEL-1 INTERRUPT
        AORG    <location>
          .
          .

        CLR     W1              CLEAR R0 OF ISR
        LI      R12,>200        SET CRU S/W BASE ADRS FOR 9901
        LI      R0,>4E3         GET 20 MS. TIMER VALUE
        LDCR    R0,15           START TIMER
        SBZ     0               SWITCH 9901 TO INTERRUPT MODE
        SBO     3               ENABLE 9901 INTERRUPT 3
        LIMI    1               ENABLE 9980 INTERRUPT 1
          .
          .
          .

W1      BSS     32              WORKSPACE AREA FOR ISR
P1      INC     R0              RECORD ONE TIME INTERVAL
        CI      R0,25           HALF SECOND EXPIRED?
        JLT     RT              NOT YET
        JNE     DN              HALF SECOND EXACTLY?
        LI      R12,>612        YES - SET UP CRU BIT ADRESS
        SBO     0               TURN ON THE BIT
        JMP     RT              EXIT
DN      CI      R0,30           TIME TO TURN OFF BIT?
        JNE     RT              NOT YET
        LI      R12,>612        YES - SET UP CRU BIT ADDRESS
        SBZ     0               TURN OFF THE BIT
        LI      R0,5            INITIALIZE S/W TIMER COUNT
RT      LI      R12,>200        SET CRU S/W BASE ADRS FOR 9901
        SBO     0               CLEAR AND REENABLE 9901 INTRPT
        RTWP                    RETURN - REVERSE CONTEXT SWITCH
```

APPENDIX C

TMS 9900/9980 INSTRUCTION FORMATS**



OP CODE	OPERATION CODE
B	BYTE INDICATOR (1=BYTE)
T_D	DESTINATION ADDRESS TYPE*
DR	DESTINATION REGISTER
T_S	SOURCE ADDRESS TYPE*
SR	SOURCE REGISTER
C	CRU TRANSFER COUNT OR SHIFT COUNT
R	REGISTER
N	NOT USED

*T_D OR T_S	ADDRESS MODE TYPE
00	DIRECT REGISTER
01	INDIRECT REGISTER
10	PROGRAM COUNTER RELATIVE, NOT INDEXED (SR OR DR = 0) PROGRAM COUNTER RELATIVE + INDEX REGISTER (SR OR DR>0)
11	INDIRECT REGISTER, AUTOINCREMENT REGISTER

**TM 990/100M Microcomputer User's Guide, Texas Instruments Incorporated, 1978, page 4-7.

A P P E N D I X D

ASSIGNMENT OF TMS 9902 INPUT AND OUTPUT BITS

D.1 OUTPUT

Control Register

7	6	5	4	3	2	1	0
SBS1	SBS2	PENB	PODD	CLK4M	——	RCL1	RCL0

When Bit 14 is set, data written to bits 0 to 7 are loaded into the control register.

Bits 6 and 7--stop bit selection SBS1,SBS2:

SBS1 7	SBS2 6	No. of XMIT Stop Bits
0	0	1 1/2
0	1	2
1	0	1
1	1	1

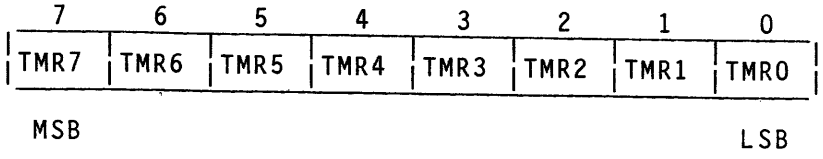
Bits 5,4--Parity Selection PENB,PODD:

PENB 5	PODD 4	Parity
0	X	None
1	0	Even
1	1	Odd

Bits 1,0--Character-length select RCL1,RCL0:

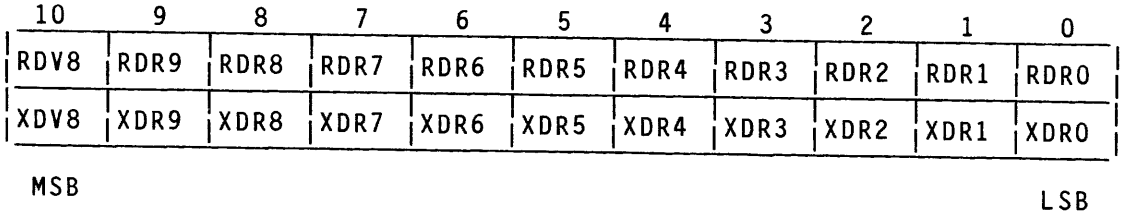
RCL1 1	RCL0 0	Character Length
0	0	5
0	1	6
1	0	7
1	1	8

Interval Register



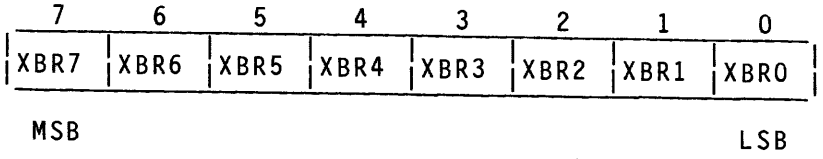
The value in the interval register is the number of units of 64 internal clock cycles between timer interrupts. For example, if f0 = 3MHz, and CLK4M in the control register is 0, then the internal clock period is 1 μsec, so an interval register containing $16_{10} = 10_{16}$ would result in an interval of 1 μsec x 64 x 16 = 1.024 msec.

Receive Data-Rate Register, Transmit Data-Rate Register



The values in the data-rate registers determine the length of one-half of the data period. RDV8 and XDV8 determine if the internal clock is divided by 8 (RDV8, XDV8 = 1, divide by 8; RDV8, XDV8 = 0, divide by 1). The remaining 10 bits represent an integer number of clock periods (or 8 clock periods) comprising one half period. For example, assuming an internal clock period of 1 μsec, and a register value of 10011010000, the resulting data rate would be 1 MHz ÷ 8 ÷ 208 ÷ 2 = 300.48 bps.

Transmit Buffer Register



This register contains the next character to be transmitted, right-justified and zero-filled. That is, even if less than 8 bits are to be transmitted, all 8 bits should be loaded.

Register Load-Control Flags

These bits identify the destination of the next data written to bits 10 to 0, according to the following truth table.

	LDCTRL	LDIR	LRDR	LXDR	DESTINATION of bits 10-0
	1	X	X	X	Control Register
	0	1	X	X	Internal Register
*	0	0	1	X	Receive Data-Rate Register
*	0	0	X	1	Transmit Data-Rate Register
	0	0	0	0	Transmit Buffer Register

*These two lines simplify the setting of equal transmit and receive data rates.

Output bits

Bit 31 RESET: Disables all interrupts, initializes transmitter and receiver, clears RTS, sets load-control flags, clears BREAK flag. Requires 11 clock cycles.

Bit 21 DSCENB: Data Set Change Interrupt Enable, enables the interrupt resulting from change in status.

Bit 20 TIMENB: Timer Interrupt Enable, enables interrupts from interval timer.

Bit 19 XBIENB: Transmit Buffer Interrupt Enable, enables empty transmit buffer interrupt.

Bit 18 REINB: Receiver Interrput Enable, enables full receiver buffer interrupt.

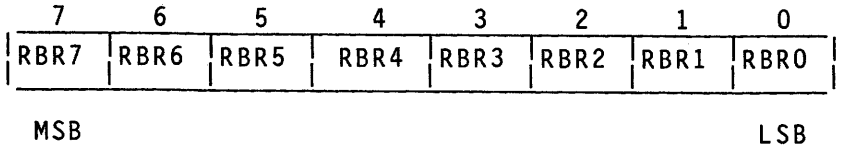
Bit 17 BRKON: Break On, sets serial output to logic 0.

Bit 16 RTSON: Request-to-Send ON, turns on RTS. Writing a ZERO to RTSON clears RTS after transmission is complete.

Bit 15 TSTMD: Test Mode, sets up high-speed local test loop.

D.2 INPUT

Receive Buffer Register



This register contains the last character to be received, right-justified and zero-filled. That is, even if less than 8 bits were received, all 8 bits are loaded.

Input Bits

Bit 31	INT = DSCINT + TIMINT + XBINT + RBINT	
Bit 20	PSCINT:	Data Set Change Interrupt, when enabled by output bit 21, indicates a change of state of either PSR or CTS.
Bit 19	TIMINT:	Timer Interrupt, when enabled by output bit 20, indicates an interval timeout has occurred.
Bit 17	XBINT:	Transmit Buffer Empty Interrupt, when enabled by output bit 19, indicates the transmit buffer has become empty.
Bit 16	RBINT:	Receive Buffer Full Interrupt, when enabled by output bit 18, indicates the receiver buffer has been filled.
Bit 30	FLAG = LDCTRL + LDIR + LRDR + LXDR + BRKON	
Bit 29	DSCH:	Data Set Status Change, indicates either DSR or CTS has changed state.
Bit 28	CTS:	Clear to Send
Bit 27	DSR:	Data Set Ready
Bit 26	RTS:	Request to Send
Bit 25	TIMGP:	Timer Elapsed, indicates interval timer timeout.
Bit 24	TIMERR:	Timer Error, set when a timeout occurs with TIMELP = 1, indicating an unprocessed timer interrput.
Bit 23	XSRE:	Transmit Shift Register Empty, indicates transmission not in progress.
Bit 22	XBRE:	Transmit Buffer Register Empty, indicates the contents of the transmit buffer is not the next character to be sent.
Bit 21	RBRL:	Receive Buffer Register Loaded, indicates a new character is in the receive buffer.

Bit 15	RIN:	Receive Input
Bit 14	RSBD:	Receive Start Bit Detect, indicates receipt of a valid start bit, valid for one character time.
Bit 13	RFBD:	Receive Full Bit Detect, indicates the sample time for the first data bit.
Bit 9	REVERR:	Receive Error = RFER + ROVER + RPER
Bit 12	RFER:	Receive Framing Error, set when the receiver detects a 0 when looking for a stop bit, which should be a 1.
Bit 11	ROVER:	Receive Overrun Error, set when a new character is available before the previous character has been read.
Bit 10	RPER:	Receive Parity Error, set when a received character does not match the parity defined by bits 4 and 5, PENB, PODD of the control register.
Bits 7-0	Receive	Buffer Register, right justified with leading zeros.

APPENDIX E

TMS 9980A PIN DESCRIPTION*

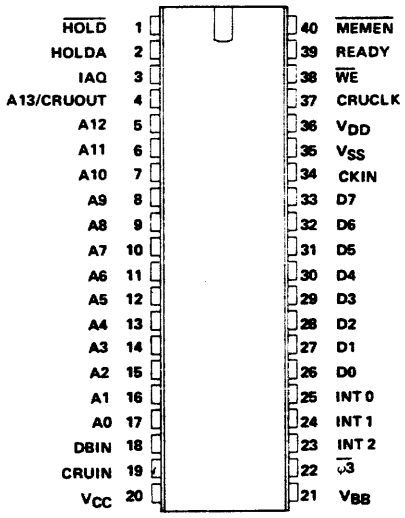
2.8 TMS 9980A PIN DESCRIPTION

Table 2 defines the TMS 9980A pin assignments and describes the function of each pin.

TABLE 2
TMS 9980A PIN ASSIGNMENTS AND FUNCTIONS

SIGNATURE	PIN	I/O	DESCRIPTION
			ADDRESS BUS
A0 (MSB)	17	OUT	A0 through A13 comprise the address bus. This 3-state bus provides the memory-address vector to the external-memory system when MEMEN is active and I/O-bit addresses and external-instruction addresses to the I/O system when $\overline{\text{MEMEN}}$ is inactive. The address bus assumes the high-impedance state when HOLDA is active.
A1	16	OUT	
A2	15	OUT	
A3	14	OUT	
A4	13	OUT	
A5	12	OUT	
A6	11	OUT	
A7	10	OUT	
A8	9	OUT	
A9	8	OUT	
A10	7	OUT	
A11	6	OUT	
A12	5	OUT	
A13/CRUOUT	4	OUT	**CRUOUT** Serial I/O data appears on A13 when an LDCR, SBZ and SBO instruction is executed. This data should be sampled by the I/O interface logic when CRUCLK goes active (high). One bit of the external instruction code appears on A13 during external instruction execution.
			DATA BUS
D0 (MSB)	26	I/O	D0 through D7 comprise the bidirectional 3-state data bus. This bus transfers memory data to (when writing) and from (when reading) the external-memory system when $\overline{\text{MEMEN}}$ is active. The data bus assumes the high-impedance state when HOLDA is active.
D1	27	I/O	
D2	28	I/O	
D3	29	I/O	
D4	30	I/O	
D5	31	I/O	
D6	32	I/O	
D7 (LSB)	33	I/O	
			POWER SUPPLIES
V_{BB}	21		Supply voltage (−5V NOM)
V_{CC}	20		Supply voltage (5 V NOM)
V_{DD}	36		Supply voltage (12 V NOM)
V_{SS}	35		Ground reference
			CLOCKS
CKIN	34	IN	Clock In. A TTL compatible input used to generate the internal 4-phase clock. CKIN frequency is 4 times the desired system frequency.
$\overline{\phi 3}$	22	OUT	Clock phase 3 ($\phi 3$) inverted; used as a timing reference.
			BUS CONTROL
DBIN	18	OUT	Data bus in. When active (high), DBIN indicates that the TMS 9980A has disabled its output buffers to allow the memory to place memory-read data on the data bus during MEMEN. DBIN remains low in all other cases except when HOLDA is active at which time it is in the high-impedance state.

TMS 9980A PIN ASSIGNMENTS



*TM 9980A/TMS 9981 Microprocessor Data Manual, Texas Instruments Incorporated, 1977, pp. 13-14.

TABLE 2 (CONTINUED)

SIGNATURE	PIN	I/O	DESCRIPTION
$\overline{\text{MEMEN}}$	40	OUT	Memory enable. When active (low), $\overline{\text{MEMEN}}$ indicates that the address bus contains a memory address. When HOLDA is active, $\overline{\text{MEMEN}}$ is in the high impedance state.
$\overline{\text{WE}}$	38	OUT	Write enable. When active (low), $\overline{\text{WE}}$ indicates that memory-write data is available from the TMS 9980 to be written into memory. When HOLDA is active, $\overline{\text{WE}}$ is in the high-impedance state.
CRUCLK	37	OUT	CRU clock. When active (high), CRUCLK indicates that external interface logic should sample the output data on CRUOUT or should decode external instructions on A0, A1, A13.
CRUIN	19	IN	CRU data in. CRUIN, normally driven by 3-state or open-collector devices, receives input data from external interface logic. When the processor executes a STCR or TB instruction, it samples CRUIN for the level of the CRU input bit specified by the address bus (A2 through A12).
INT2	23	IN	Interrupt code. Refer to Section 2.2 for detailed description.
INT1	24	IN	
INT0	25	IN	
			MEMORY CONTROL
$\overline{\text{HOLD}}$	1	IN	Hold. When active (low), $\overline{\text{HOLD}}$ indicates to the processor that an external controller (e.g., DMA device) desires to utilize the address and data buses to transfer data to or from memory. The TMS 9980A enters the hold state following a hold signal when it has completed its present memory cycle.* The processor then places the address and data buses in the high-impedance state (along with $\overline{\text{WE}}$, $\overline{\text{MEMEN}}$, and DBIN) and responds with a hold-acknowledge signal (HOLDA). When $\overline{\text{HOLD}}$ is removed, the processor returns to normal operation.
HOLDA	2	OUT	Hold acknowledge. When active (high), HOLDA indicates that the processor is in the hold state and the address and data buses and memory control outputs ($\overline{\text{WE}}$, $\overline{\text{MEMEN}}$, and DBIN) are in the high-impedance state.
READY	39	IN	Ready. When active (high), READY indicates that memory will be ready to read or write during the next clock cycle. When not-ready is indicated during a memory operation, the TMS 9980A enters a wait state and suspends internal operation until the memory systems indicated ready.
			TIMING AND CONTROL
IAQ	3	OUT	Instruction acquisition. IAQ is active (high) during any memory cycle when the TMS 9980A is acquiring an instruction. IAQ can be used to detect illegal op codes. It may also be used to synchronize LOAD stimulus.

*If the cycle following the present memory cycle is also a memory cycle it, too, is completed before TMS 9980 enters hold state.

A P P E N D I X F

INSTRUCTION SUMMARIES

ALPHABETIZED LIST

MNEMONIC CODE	INSTRUCTION SUMMARY NUMBER	PAGE NUMBER	MNEMONIC CODE	INSTRUCTION SUMMARY NUMBER	PAGE NUMBER
A	3-2	120	JOP	7-6	342
AB	4-3	181	LDCR	6-5	301
ABS	3-4	122	LI	3-7	145
AI	3-8	146	LIMI	8-8	391
ANDI	5-9	247	LREX	6-8	304
B	4-12	193	LWPI	4-1	178
BL	8-1	384	MOV	3-1	113
BLWP	8-3	386	MOVB	4-2	180
C	4-5	183	MPY	8-5	388
CB	4-6	186	NEG	7-9	345
CI	4-7	187	ORI	5-10	248
CKOF	6-10	306	RSET	6-7	303
CKON	6-9	305	RTWP	8-4	387
CLR	7-11	347	S	3-3	121
COC	5-3	241	SB	4-4	182
CZC	5-4	242	SBO	6-2	298
DEC	3-9	150	SBZ	6-3	299
DECT	3-11	152	SETO	7-10	346
DIV	8-6	389	SLA	4-11	192
IDLE	6-6	302	SOC	5-5	243
INC	3-10	151	SOCB	5-6	244
INCT	3-12	153	SRA	5-12	250
INV	7-8	344	SRC	5-14	252
JEQ	4-8	189	SRL	5-13	251
JGT	3-6	128	STCR	6-4	300
JH	7-2	338	STST	8-10	393
JHE	7-4	340	STWP	8-9	392
JL	7-1	337	SWPB	7-7	343
JLE	7-3	339	SZC	5-7	245
JLT	4-10	191	SZCB	5-8	246
JMP	3-5	127	TB	6-1	297
JNC	5-2	240	X	8-7	390
JNE	4-9	190	XOP	8-2	385
JNO	7-5	341	XOR	5-11	249
JOC	5-1	239			

INTRODUCTION TO MICROPROCESSORS

USER RESPONSE SHEET

1. Is the general introduction to microprocessors (Chapter 1) covered adequately? Y__ N__ Comments:

_____

_____

2. Is the instruction set covered adequately? Y__ N__ Comments:

_____

_____

3. Is enough information provided on I/O and the CRU? Y__ N__ Comments:

_____

_____

4. Is enough information provided on applications? Y__ N__ Comments:

_____

_____

5. Does this manual provide you with enough information so that ypu can implement microprocessors at the system level of design? Y__ N__ Comments:

_____

_____

6. On what subjects in this textbook would you like to have more information or further clarity? _____

_____

7. General Comments: _____

_____

_____

Name _____ School _____

Street _____ City _____ St. _____ Zip _____

REV. B

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

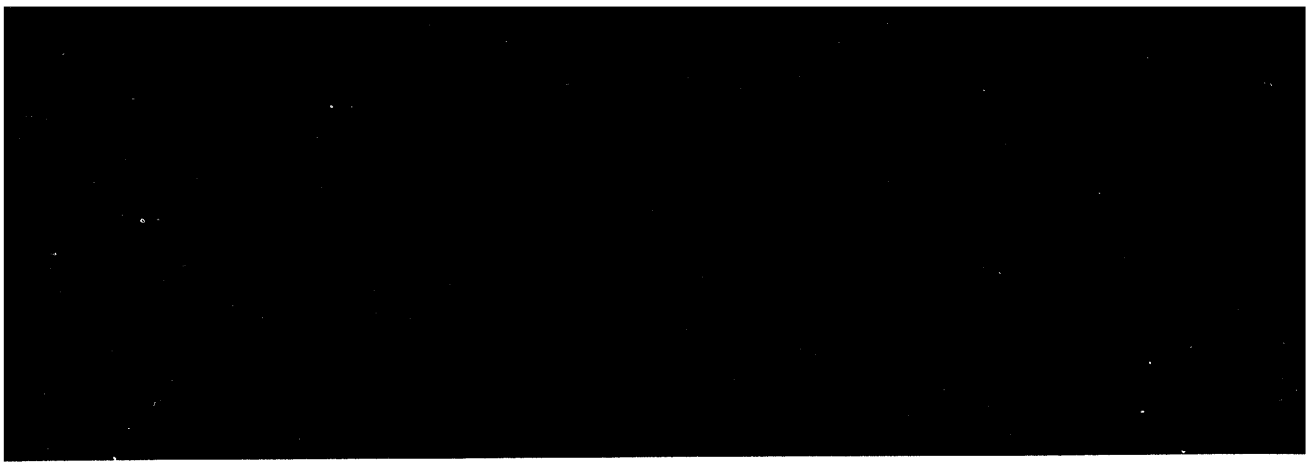
FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
MOS MICROCOMPUTER PRODUCTS
BOX 1443
HOUSTON, TEXAS 77001

ATTENTION: MICROCOMPUTER PRODUCTS
M/S 6750

FOLD



TEXAS INSTRUMENTS
INCORPORATED

MPB30 Rev. B
1602008-9701

Post Office Box 1443 / Houston, Texas 77001
Semiconductor Group

Printed in U.S.A.