THE 99000 MICROPROCESSOR

Focusing on the 99000 family of 16-bit microprocessors from Texas Instruments, this book is for anyone involved in designing electronic systems with microprocessors and microcomputers. It stresses the three primary areas in which a thorough background is needed: microcomputer architecture, microcomputer software, and hardware interface techniques.

The advent of 16-bit microprocessor technology has led to the introduction of newer system concepts, such as pipelining, instruction prefetch, microcoding, illegal instruction detection, macroinstruction emulation, user and supervisor modes of operation, bus demultiplexing, memory paging, memory segmentation, memory mapping, error detection and correction, and cache memory—all of which are discussed in this book.

This book also includes topics common to 8-bit microprocessor technology, such as microprocessor internal architecture, microcomputer system architecture, instruction execution, addressing modes, instruction set, programming techniques, bus cycles, program and data storage memory subsystems, input/output circuitry, and interrupt processing.

THE 99000 MICROPROCESSOR

ISBN 0-13-622846-1

THE 99000 MICROPROCESSOR

CONTENTS

PREFACE

One of the most highly visible types of products in the semiconductor marketplace today is the 16-bit microprocessor. Five major 16-bit processor families are currently available: the 99000 from Texas Instruments, the 8086 from Intel, the 68000 from Motorola, the Z8000 from Zilog, and the 16000 from National Semiconductor. These manufacturers are presently in competition to establish market share in the 16-bit marketplace. This book focuses on just one of the newest of these products, the 99000 family of 16-bit microprocessors from Texas Instruments.

Everyone involved in the design of electronic systems involving microprocessors and microcomputers must have a thorough knowledge of three primary areas: microcomputer architecture, microcomputer software, and hardware interface techniques. However, most of the books presently available on the subject stress the architecture of microprocessors, their instruction sets, and programming techniques. Typically, very little information is provided on hardware design and interface techniques. This leaves a gap in one's understanding of how a microprocessor interacts and interfaces with its memory and I/O subsystems. This information is the key to successful application of microcomputer systems. This book closes the gap between the study of microprocessors and microcomputer systems by putting equal emphasis on the subjects of microprocessor architecture, assembly language programming, and hardware interface techniques.

The material presented in the book includes topics common to 8-bit microprocessor technology, such as microprocessor internal architecture,

microcomputer system architecture, instruction execution, addressing modes, instruction set, programming techniques, bus cycles, program and data storage memory subsystems, input/output circuitry, and interrupt and exception processing. However, the advent of 16-bit microprocessor technology has led to the introduction of newer system concepts, such as pipelining, instruction prefetch, microcoding, illegal instruction detection, macroinstruction emulation, user and supervisor modes of operation, bus demultiplexing, memory paging, memory segmentation, memory mapping, error detection and correction, and cache memory. Detailed coverage of these more modern topics is also provided in this book.

Use of the book does require some prior knowledge of basic digital electronics. This background is at a level consistent with but not necessarily as extensive as the material covered in earlier Prentice-Hall books: *Integrated Digital Electronics*, Walter A. Triebel, 1979, and *Handbook of Semiconductor and Bubble Memories*, Walter A. Triebel and Alfred E. Chu, 1982.

We would like to express special appreciation to Dr. Jerry Van Aken for his worthwhile input on the 99000 microprocessor.

Every effort has been made to provide up-to-date information on the devices covered in the book. However, it is recommended that readers check with the manufacturer for the most recent data.

AVTAR SINGH
WALTER A. TRIEBEL

1

INTRODUCTION TO MICROPROCESSORS AND MICROCOMPUTERS

1.1 INTRODUCTION

The most recent advances in computer system technology have been closely related to the development of high-performance 16-bit microprocessors and their microcomputer systems. During the last three years, the 16-bit microprocessor market has matured significantly. Today, several complete 16-bit microprocessor families are available. They include support products such as large-scale-integrated peripheral devices, development systems, emulators, and high-level software languages. Over the same period of time, these higher-performance microprocessors have become more widely used in the design of new electronic equipment and computers.

This book represents a detailed study of one of the newest 16-bit microprocessors, the 99000 from Texas Instruments. In this chapter we begin our study of microprocessors and microcomputers. The following topics are discussed:

1. The digital computer
2. Mainframe computers, minicomputers, and microcomputers
3. Hardware elements of the digital computer system
4. General architecture of a microcomputer system
5. Types of microprocessors and single-chip microcomputers

1.2 THE DIGITAL COMPUTER

As a starting point, let us consider what a *computer* is, what it can do, and how it does it. A computer is a digital electronic data processing system. Data are input to the computer in one form, processed within the computer, and the information that results is either output or stored for later use. Figure 1.1 shows a modern computer system.

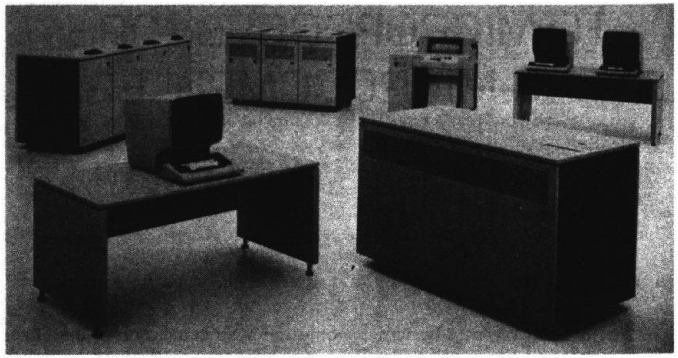


Figure 1.1 Modern large-scale computer. (Courtesy of *International Business Machine Corp.*)

Computers cannot think about how to process data that were input. Instead, the user must tell the computer exactly what to do. The procedure by which a computer is told how to work is called *programming* and the person who writes programs for a computer is known as a *programmer*. The result of the programmer's work is a set of instructions for the computer to follow. This is the computer's *program*. When the computer is operating, the instructions of the program guide it step by step through the task that is to be performed.

For example, a large department store can use a computer to take care of bookkeeping for its customer charge accounts. In this application, data about items purchased by the customers, such as price and department, are entered into the computer by an operator. These data are stored in the computer under the customer's account number. On the next billing date, the data are processed and a tabular record of each customer's account is output by the computer. These statements are mailed to the customers in the form of bills.

In a computer, the program controls the operation of a large amount of electronic circuitry. It is this circuitry that actually does the processing of data. Electronic computers first became available in the 1940s. These early computers were built with vacuum-tube electronic circuits. In the 1950s, a second generation of computers was built. During this period, transistor electronic circuitry, instead of tubes, was used to produce more compact and more reliable computer systems. When the *integrated circuit* (IC) came into the electronic market during the 1960s, a third generation of computers appeared. With ICs, industry could manufacture more complex, higher-speed, and very reliable computers.

Today, the computer industry is continuing to be revolutionized by the advances made in integrated-circuit technology. It is now possible to manufacture *large-scale integrated circuits* (LSI) that can form a computer with just a small group of ICs. In fact, in some cases, a single IC can be used. These new technologies are rapidly advancing the low-performance, low-cost part of the computer marketplace by permitting simpler and more cost effective designs.

1.3 MAINFRAME COMPUTERS, MINICOMPUTERS, AND MICROCOMPUTERS

For many years the computer manufacturers' aim was to develop larger and more powerful computer systems. These are what are known as large scale or *mainframe computers*. Mainframes are always *general-purpose computers*. That is, they are designed with the ability to run a large number of different types of programs. For this reason, they can solve a wide variety of problems.

For instance, one user can apply the computer in an assortment of scientific applications where the primary function of the computer is to solve complex mathematical problems. A second user can apply the same basic computer system to perform business tasks such as accounting and inventory control. The only difference between the computer systems used in these two applications could be their programs. In fact, today many companies use a single general-purpose computer to resolve both their scientific and business needs.

Figure 1.1 is an example of a mainframe computer manufactured by International Business Machines Corporation (IBM). Because of their high cost, mainframes find use only in central computing facilities of large businesses and institutions.

The many advances that have taken place in the field of electronics over the past two decades have led to rapid advances in computer system technology. For instance, the introduction of *small-scale integrated circuits*

(SSIs), followed by *medium-scale integrated circuits* (MSIs) and *large-scale integrated circuits* (LSIs), has led the way in expanding the capacity and performance of the large mainframe computers. But at the same time, these advances have also permitted the introduction of smaller, lower-performance, lower-cost computer systems.

As computer use grew, it was recognized that the powerful computing capability of a mainframe was not needed by many customers. Instead, easier access to a machine with smaller capacity was required. It was for this reason that the *minicomputer* was developed. Minicomputers, such as that shown in Fig. 1.2, are also digital computers and are capable of performing the same basic operations as the earlier, larger systems. However, they are designed to provide a smaller functional capability. The processor section of this type computer is typically manufactured using SSI and MSI electronic circuitry.

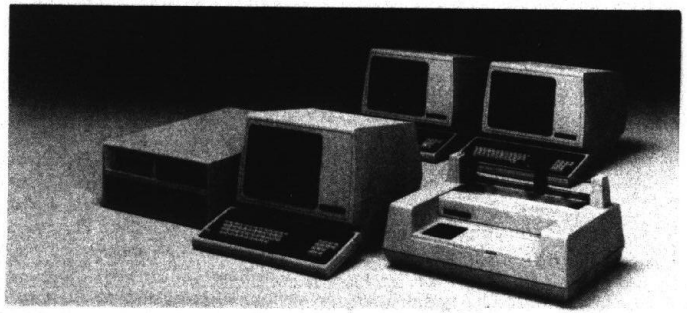


Figure 1.2 Minicomputer system. (Courtesy of *Texas Instruments, Incorporated.*)

Minicomputers have found wide use as general-purpose computers, but their lower cost also allows their use in dedicated applications. A computer used in a dedicated application represents what is known as a *special-purpose computer.* By "special-purpose computer" we mean a system that has been tailored to meet the needs of a specific application. Examples are process control computers for industrial facilities, data processing systems for retail stores, and medical analysis systems for patient care. Figure 1.3 shows a minicomputer-based retail store data processing system.

The newest development in the computer industry is the *microcomputer.* The microcomputer represents the next step in the evolution of the

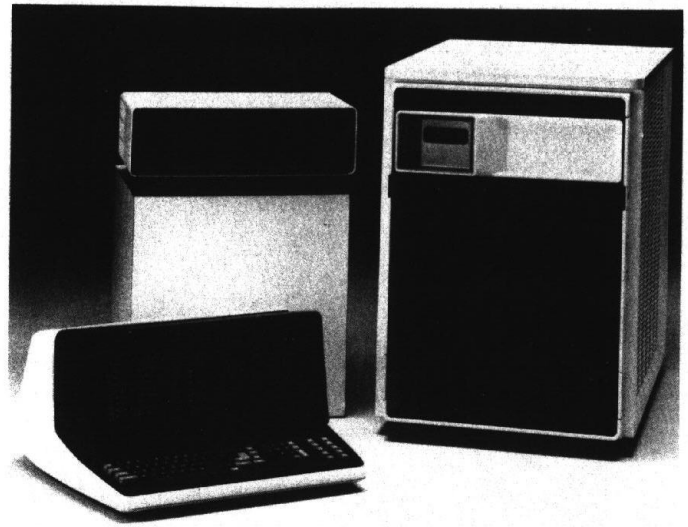


Figure 1.3 Point-of-sale system. (Courtesy of *Sweda International, Inc.*)

computer. It is designed to be smaller and to provide less capability than a minicomputer, at a much lower cost.

The heart of the microcomputer system is the *microprocessor.* A microprocessor is a general-purpose processor built into a single IC. It is an example of an LSI device. Together with the use of LSI circuitry in the microcomputer have come the benefits of smaller size, lighter weight, lower cost, reduced power requirements, and higher reliability.

The low cost of microprocessors, which can be as low as $1, has made possible the use of computer electronics in a much broader range of products. Figures 1.4 and 1.5 show some typical systems in which a microcomputer is used as a special-purpose computer.

Microcomputers are also finding wide use as general-purpose computers. Figures 1.6 and 1.7 show examples of a home computer and a personal computer. In fact, microcomputer systems designed for the high-performance end of the microcomputer market are rivaling the performance of the lower-performance minicomputers and at much lower cost to the user.

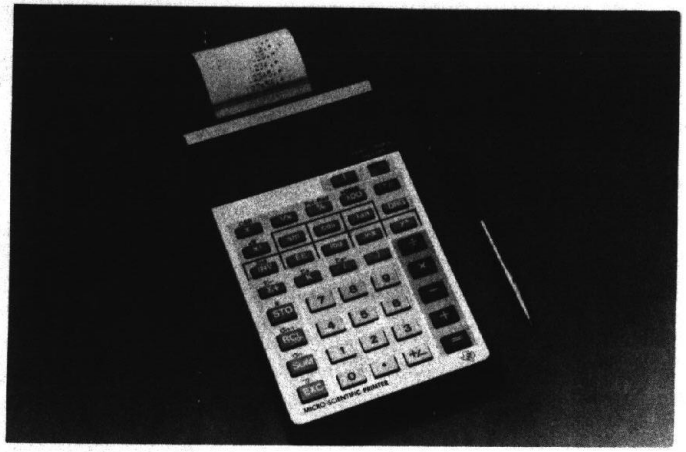


Figure 1.4 Calculator. (Courtesy of *Texas Instruments, Incorporated.*)

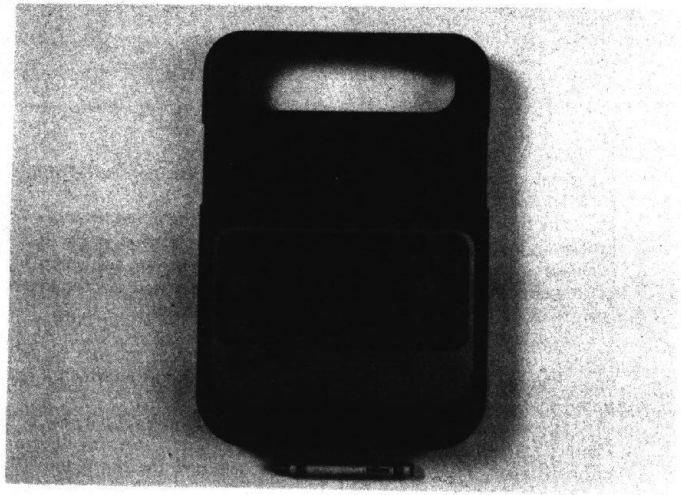


Figure 1.5 Electronic toy. (Courtesy of *Texas Instruments, Incorporated.*)

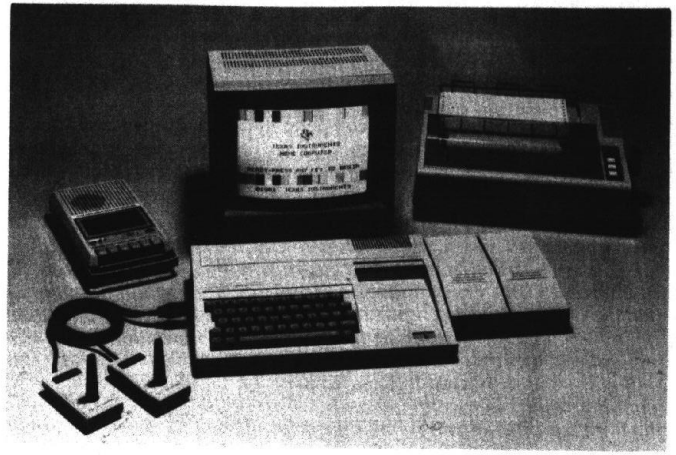


Figure 1.6 Home computer. (Courtesy of *Texas Instruments, Incorporated.*)

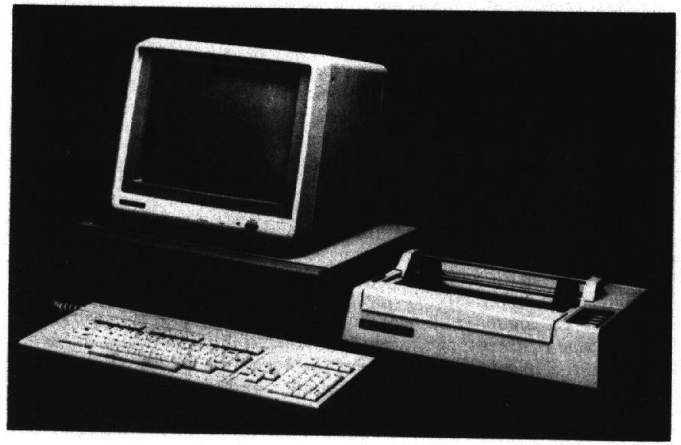


Figure 1.7 Personal computer. (Courtesy of *Texas Instruments, Incorporated.*)

1.4 HARDWARE ELEMENTS OF THE DIGITAL COMPUTER SYSTEM

The hardware of a digital computer system is divided into four functional sections. The block diagram of Fig. 1.8 shows the four basic units of a simplified computer: the *input unit, central processing unit, memory unit,* and *output unit.* Each section has a special function in terms of overall computer operation.

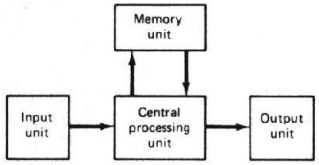


Figure 1.8 Block diagram of a digital computer. (From Walter A. Triebel, *Integrated Digital Electronics,* © 1979. Adapted by permission of Prentice-Hall, Inc., Englewood Cliffs, N.J.)

The *central processing unit* (CPU) is the heart of the computer system. It is responsible for performing all arithmetic operations and logic decisions initiated by the program. In addition to arithmetic and logic functions, the CPU controls overall system operation.

On the other hand, the input and output units are the means by which the CPU communicates with the outside world. The *input unit* is used to input information and commands to the CPU for processing. For instance, a Teletype terminal can be used by the programmer to input a new program.

After processing, the information that results must be output. This output of data from the system is performed under control of the *output unit.* Examples of ways of outputting information are as printed pages produced by a high-speed printer or to display it on the screen of a video display terminal.

The *memory unit* of the computer is used to store information such as numbers, names, and addresses. By "store" we mean that memory has the ability to hold this information for processing or for output at a later time. The programs that define how the computer is to process data also reside in memory.

In computer systems, memory is divided into two sections, known as *primary storage* and *secondary storage.* They are also sometimes called *internal memory* and *external memory,* respectively. *External memory* is used for long-term storage of information that is not in use. For instance, it holds programs, files of data, and files of information. In most computers, this part of memory employs storage on magnetic media such as magnetic tapes, magnetic disks, and magnetic drums. This is because they have the ability to store large amounts of data.

On the other hand, *internal memory* is a smaller segment of memory used for temporary storage of programs, data, and information. For instance, when a program is to be executed, its instructions are first brought from external memory into internal memory together with the files of data and information that it will affect. After this, the program is executed and its files updated while they are held in internal memory. When the processing defined by the program is complete, the updated files are returned to external memory. Here the program and files are retained for use at a later time.

The internal memory of a computer system uses electronic memory devices instead of storage on a magnetic-medium memory. In most modern computer systems, semiconductor read-only memory (ROM) and random access read/write memory (RAM) are in use. These devices make internal memory operate much faster than external memory.

Neither semiconductor memory nor magnetic-medium memory alone can satisfy the requirements of most general-purpose computer systems. Because of this fact, both types are normally present in the system. For instance, in a personal computer system, working storage is typically provided with RAM while long-term storage is provided with floppy disk memory. On the other hand, in special-purpose computer systems, such as video games, only semiconductor memory is used. That is, the program that determines how the game is played is stored in ROM, and data storage, such as for graphic patterns, are in RAM.

1.5 GENERAL ARCHITECTURE OF A MICROCOMPUTER SYSTEM

Now that we have introduced the *general architecture* of a digital computer, let us look at how a microcomputer fits this model. Looking at Fig. 1.9, we find that the architecture of the microcomputer is essentially the same as that of the digital computer shown in Fig. 1.8. It has the same functional elements: input unit, output unit, memory unit, and in place of the CPU, an MPU (*microprocessor unit*). Moreover, each element serves the same basic functions relative to overall system operation.

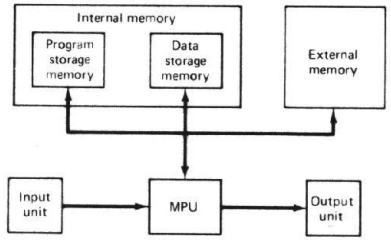


Figure 1.9 General microcomputer system architecture.

The difference between minicomputers, mainframe computers, and microcomputers does not lie in the fundamental blocks used to build the computer; instead, it is in the capacity and performance of the electronics used to implement their blocks and the resulting overall system capacity and performance. As indicated earlier, microcomputers are designed with smaller capacity and lower performance than either minicomputers or mainframes.

Unlike mainframe and minicomputers, a microcomputer can be implemented with a small group of components. Again the heart of the computer system is the MPU (CPU) and it performs all arithmetic, logic, and control operations. However, in a microcomputer the MPU is implemented with a single microprocessor chip instead of a large assortment of SSI and MSI logic functions such as in minis and mainframes. Note that correct use of the term "microprocessor" restricts its use to the central processing unit in a microcomputer system.

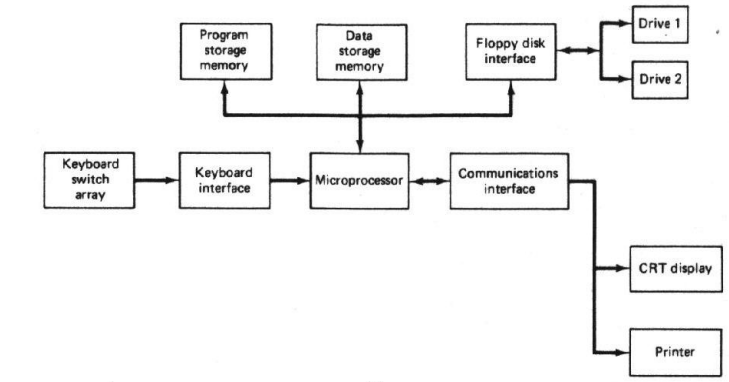
Note that we have partitioned the memory unit into an internal memory section for storage of active data and instructions and an external memory section for long-term storage. As in minicomputers, the long-term storage medium in a microcomputer is frequently a floppy disk. However, Winchester rigid disk drives are also becoming popular when storage requirements are higher than those provided by floppy disks. In industrial applications, where the environment for the equipment is rugged, bubble memories are also being employed as long-term storage devices.

Internal memory of the microcomputer is further subdivided into *program storage memory* and *data storage memory*. Typically, it is implemented with both ROM and RAM ICs. Data, whether they are to be interpreted as numbers, characters, or instructions, can be stored in either ROM or RAM. But in most microcomputer systems, instructions for the program and data, such as look-up tables, are stored in ROM. This is because this type of information does not normally change. By using ROM, its storage is made *nonvolatile*. That is, if power is lost, the information is retained.

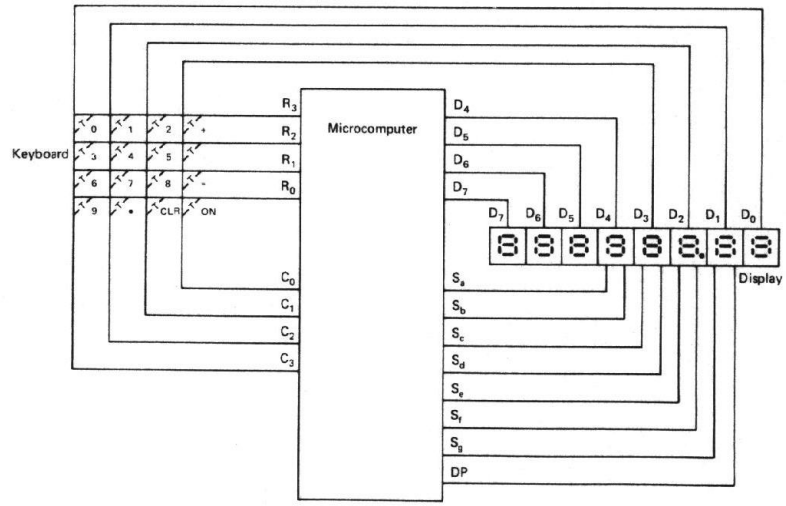
On the other hand, the numerical and character data that are to be processed by the microprocessor change frequently. They must be stored in a type of memory from which they can be read by the microprocessor, modified through processing, and written back for storage. For this reason, data are stored in RAM instead of ROM.

Depending on the application, the input and output sections can be implemented with something as simple as a few switches for inputs and a few light-emitting diodes (LEDs) for outputs. In other applications, for example a personal computer, the input/output (I/O) devices can be more sophisticated devices, such as video display terminals and printers, just like those employed in minicomputer systems.

Up to this point, we have been discussing what is known as a *multichip microcomputer system*: that is, a system implemented with a microprocessor and an assortment of support circuits such as ROMs, RAMs, and I/O peri-



(a)



(b)

Figure 1.10 (a) Block diagram of a personal computer; (b) block diagram of a calculator.

pherals. This architecture makes for a very flexible system design. The system's ROM, RAM, and I/O capacity can easily be expanded by simply adding more devices. This is the circuit configuration used in most larger microcomputer systems. An example is the personal computer system shown in Fig. 1.10(a).

Devices are now being made that include all the functional blocks of a microcomputer in a single IC. This is called a *single-chip microcomputer*. Unlike the multichip microcomputer, single-chip microcomputers are limited in capacity and are not as easy to expand. For example, a microcomputer device can have 4K (4096; K = 1024) bytes of ROM, 128 bytes of RAM, and 32 lines for use as inputs or outputs. Because of this limited capability, single-chip microcomputers find wide use in special-purpose computer applications. A block diagram of a calculator implemented with a single-chip microcomputer is shown in Fig. 1.10(b).

1.6 TYPES OF MICROPROCESSORS AND SINGLE-CHIP MICROCOMPUTERS

The primary way in which microprocessors and microcomputers are categorized is in terms of the number of binary bits in the data they process: that is, their word length. Figure 1.11 shows that the three standard organizations used in the design of microprocessors and microcomputers are 4-bit, 8-bit, and 16-bit data words.

The first microprocessors and microcomputers, introduced in the early 1970s, were all designed to process data that were arranged 4 bits wide. This organization is frequently referred to as a *nibble* of data. Many of the early 4-bit devices, such as the PPS-4 microprocessor made by Rockwell Interna-

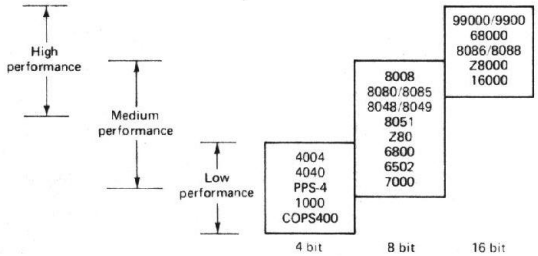


Figure 1.11 Microprocessor and single-chip microcomputer categories and relative performance.

tional and the TMS1000 single-chip microcomputer made by Texas Instruments, are still in wide use today.

The low performance and limited system capabilities of these 4-bit microcomputers limit their use to simpler, special-purpose applications. Some common uses are in calculators and electronic toys. In these types of equipment, low cost, not high performance, is the overriding requirement in the selection of a processor.

In the 1973–1974 period, second-generation microprocessors were introduced. These devices, such as Intel Corporation's 8008 and 8080, were 8-bit microprocessors; that is, they were designed to process 8-bit (one-byte-wide) data instead of 4-bit data.

The newer 8-bit microprocessors exhibited higher-performance operation, larger system capabilities, and greater ease of programming. They were able to provide the system requirements for many applications that could not be satisfied by 4-bit microcomputers. These extended capabilities led to widespread acceptance of multichip 8-bit microcomputers for special-purpose system designs. Examples of some of these dedicated applications are electronic instruments, cash registers, and printers.

Somewhat later, 8-bit microprocessors began to migrate into general-purpose microcomputer systems. In fact, the Z-80A is the host MPU in a number of today's popular personal computers.

Late in the 1970s, 8-bit single-chip microcomputers, such as Intel's 8048, became available. The full microcomputer capability of this single chip further reduced the cost of implementing designs for smaller, dedicated digital systems. In fact, 8-bit microcomputers are still being designed for introduction into today's marketplace. An example is Texas Instruments' new TMS7000 family of 8-bit microcomputers. Newer devices, such as the TMS7000, offer one order-of-magnitude higher performance, more powerful instruction sets, and special on-chip functions such as interval/event timers and universal asynchronous receiver/transmitters (UARTs).

Plans for the development of third-generation (16-bit) microprocessors were announced by many of the leading semiconductor manufacturers in the mid-1970s. The 9900, which was the forerunner of the 99000 that we describe in this book, was introduced commercially in 1977. It was followed by a number of other important devices, such as the 9981, 8086, 8088, Z8000, 68000, and 99000. These devices are all high in performance and have the ability to satisfy a broad scope of special-purpose and general-purpose computer applications. All have the ability to handle 8-bit as well as 16-bit data words. Some can even process data organized as 32-bit words. Moreover, their powerful instruction sets are more in line with those provided by minicomputers than of those associated with 8-bit microprocessors.

In terms of special-purpose applications, 16-bit microprocessors are replacing 8-bit processors in applications that require very high performance:

for example, certain types of electronic instruments. A single-chip 16-bit microcomputer, the 9940, is also available for use in this type of application.

Sixteen-bit microprocessors are also being used in applications that can benefit from their extended system capabilities. For instance, they are beginning to be used in word processing systems. This type of system requires a large number of character data to be active temporarily; therefore, it can benefit from the ability of a 16-bit microprocessor to access a much larger amount of data storage memory.

Most new personal computer designs incorporate 16-bit microprocessors. For example, IBM's personal computer and Texas Instruments' home computer use 16-bit microprocessors to implement their microcomputers.

ASSIGNMENT

Section 1.2

1. What guides a computer as to how it is to process data?
2. What type of electronic devices are revolutionizing the low-performance, low-cost computer market today?

Section 1.3

3. What is the principal difference between mainframe, mini-, and microcomputers?
4. What is meant by "general-purpose computer"?
5. What is meant by "special-purpose computer"?

Section 1.4

6. What are the building blocks of a general computer system?
7. What is the difference between primary and secondary storage?

Section 1.5

8. What are the basic building blocks of a microcomputer system?
9. What is the difference between program storage and data storage memory in a microcomputer?
10. What is the difference between internal and external storage memory in a microcomputer?

Section 1.6

11. What are the standard data word lengths of today's microprocessors and microcomputers?
12. What is the difference between a multichip microcomputer and a single-chip microcomputer?
13. Name five 16-bit microprocessor families.

2

THE 99000 MICROPROCESSOR

2.1 INTRODUCTION

Chapter 1 introduced general aspects of microprocessors and microcomputers. In this chapter we begin a detailed study of the 99000 microprocessor and its architecture. In the chapters that follow, its instruction set, external interfaces (memory, input/output, and interrupt), and special features such as macrostore and privileged mode are presented. The following list outlines the topics that are covered:

1. The 99000 microprocessor
2. Block diagram of the 99000 microprocessor
3. Internal architecture of the 99000 microprocessor
4. Instruction execution

2.2 THE 99000 MICROPROCESSOR

The 99000 is one of the higher-performance 16-bit microprocessors available in the marketplace today. As shown in Fig. 2.1, it is the third-generation member of a family of 16-bit microprocessors and microcomputers manufactured by Texas Instruments. The first family member, the 9900, was introduced in 1975. It was followed by a steady stream of new devices, including the 9980 in 1977 and the 9940 in 1979. The second-generation device, the 9995, was introduced in 1980, and the third-generation microprocessor, the 99000, in 1981.

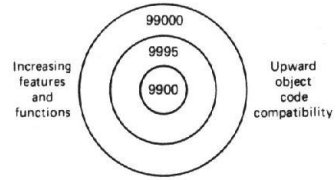


Figure 2.1 Evolution of the 9900/99000 family. (Courtesy of *Texas Instruments, Incorporated.*)

The 9900, 9995, and 99000 devices represent a continuity of hardware and software based on Texas Instrument's memory-to-memory architecture. Even though new family members have been introduced, software compatibility has been maintained for all devices. That is, the instruction sets of the newer devices are just supersets of those performed by the original 9900 microprocessor. Therefore, no modifications are required on software written for the older processors in order to run it on the newer 9995 and 99000 processors.

Through the evolution from the first 9900 microprocessor to today's 99000, many enhancements have been made to the features and functionality of the architecture and instruction set. For instance, on-chip memory known as *macrostore* permits high-speed instruction execution and at the same time permits custom and specialized instruction extensions to the baseline instruction set of the 99000. Moreover, the standard instruction set now includes additions such as signed multiply and signed divide as well as instructions that perform multiprecision arithmetic operations. An example of a hardware enhancement is the extension of the address reach of the 99000 to 256K bytes. It is these kinds of improvements that have resulted in the 99000's high throughput, efficient performance, and flexible system capabilities.

The name "99000" actually represents a family of microprocessors. At present, two family members have been defined. The first device announced, the *99105A*, is what is known as the *baseline 99000*. This means that it does not contain any special-purpose on-chip macrostore. The second family member, the *99110A*, is a processor designed for a specialized function. It has the ability to perform floating-point arithmetic operations in addition to the general-purpose capability of the 99105A. The floating-point instructions are built into its internal macrostore memory area. Throughout the book all references to the 99000 imply that we are referencing features of the 99105A that are common to the current family members.

All members of the 99000 family are manufactured using N-channel metal-oxide semiconductor (NMOS) technology. They use a variation on the basic process known as *scaled MOS* (SMOS), which is a high-performance NMOS technology. It results in a 3-micrometer active device size. The baseline 99000 chip contains approximately 25,000 transistors and when macrostore is added, this is increased to 35,000 transistors.

2.3 BLOCK DIAGRAM OF THE 99000 MICROPROCESSOR

Figure 2.2(a) shows a block diagram of the 99000 microprocessor. Here we see that the leads of the device are grouped into the *address bus, data bus, memory control lines, bus status lines, serial I/O interface, interrupt interface, DMA interface, attached processor interface, power supply lines,* and *clock signal lines.* In this diagram, we have shown all signal lines to be independent. However, the 99000 is packaged in a 40-pin package. For this reason, a number of its signals are actually multiplexed.

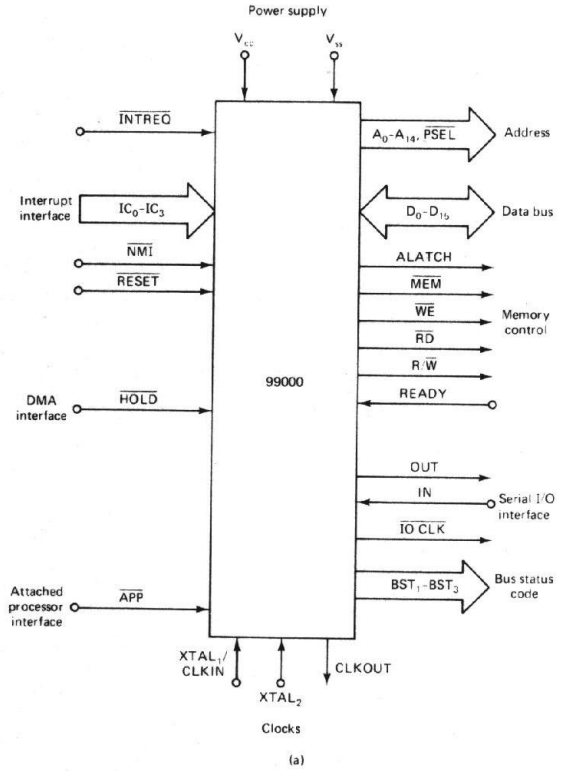


Figure 2.2(a) Block diagram of the 99000 microprocessor. (Courtesy of *Texas Instruments, Incorporated.*)

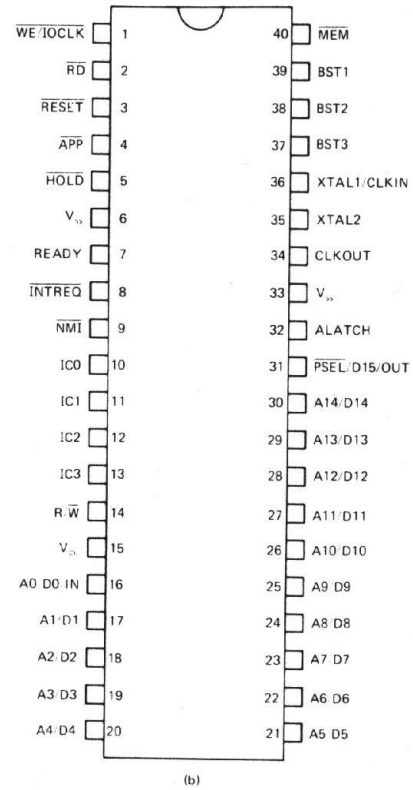


Figure 2.2(b) Pin layout. (Courtesy of *Texas Instruments, Incorporated.*)

Figure 2.2(b) shows the pin layout of the 99000. Note that the address bus and data bus signals are multiplexed over the same set of lines. These lines are labeled A_0/D_0 through A_{14}/D_{14}. Furthermore, bit D_{15} of the data bus is multiplexed with two other signals, $\overline{\text{PSEL}}$ and OUT, at pin 31.

Before going further, let us look briefly at each of these signal groups and their functions.

Address/Data Bus

The 99000 microprocessor has a 15-bit external address bus. In Fig. 2.2, these lines are labeled A_0 through A_{14}. The binary information output on this bus is used to specify the storage location that is to be accessed during data transfers between the 99000 and memory, registers within memory-mapped LSI peripherals, and I/O-mapped input/output ports. This is a *unidirectional bus*. That is, address words are output by the microprocessor only over these signal lines. Signal line $\overline{\text{PSEL}}$ is an additional address bit that is used to extend the address bus to 16 bits.

The data bus consists of 16 lines, D_0 through D_{15}. This is a *bidirectional bus* instead of a unidirectional bus. Therefore, its lines are used to carry data into or out of the microprocessor. It is over these lines that data are transferred during memory read or write operations, and for input or output of data for LSI peripherals or other I/O devices.

Memory Control Signals

A second group of signals in Fig. 2.2 are those labeled as memory control. They tell external memory circuitry whether or not a memory cycle is in progress; if the memory operation is a read or write of data; and the appropriate instants when valid address information, read data, or write data are on the bus.

In Fig. 2.3, each memory control signal is listed with its mnemonic, name, and a brief description of its function. Here we find that $\overline{\text{MEM}}$ (*memory cycle*) is the signal that indicates to external circuitry in the memory subsystem that a memory cycle is in progress. Moreover, signals ALATCH (*address latch*), $\overline{\text{WE}}$ (*write enable*), and $\overline{\text{RD}}$ (*read enable*) identify the moment that a valid address, valid write data, and valid read data, respectively, are on the multiplexed address/data bus.

Another signal is provided in this group to permit use of the 99000 with slow external memory devices. This is the READY signal. Effectively, it gives the ability to extend the bus cycle. That is, after the address is output on the bus, the MPU will wait for the READY signal to be returned from the memory system before it performs the read or write data transfer to complete the memory cycle.

Serial Input/Output Interface

The *serial input/output (I/O) interface* is the bit-addressable input/output path of the 99000. Looking at Fig. 2.2, we see that it consists of three signal lines: OUT, $\overline{\text{IOCLK}}$, and IN. These signals are listed in Fig. 2.4 together with

Mnemonic	Name	Function
$\overline{\text{MEM}}$	Memory cycle	0 indicates that a memory cycle is in progress
$\overline{\text{WE}}$	Write enable	0 indicates that valid write data is on the data bus
ALATCH	Address latch	1 indicates that the multiplexed address/data bus is set up to carry address data
R/$\overline{\text{W}}$	Early read/write	1 indicates that data bus lines will be carrying read data during data phase of the bus cycle
		0 indicates that data bus lines will be carrying write data during data phase of the bus cycle
$\overline{\text{RD}}$	Read	0 indicates that a read bus cycle is in progress and that data can be put on the data bus
READY	Ready	1 indicates that the current bus cycle can continue through to completion
		0 indicates that a wait state must be inserted into the current bus cycle

Figure 2.3 Memory control signals.

summaries of their functions. It is these lines that are used to interface with bit-oriented parallel I/O ports and some special-purpose serial-interfaced LSI peripherals.

These three I/O lines provide a bit-serial input/output mechanism. Bits of data are output in serial form over the OUT (*data output*) line synchronously with clock pulses at the $\overline{\text{IOCLK}}$ (*I/O clock*) line. Bit-serial data are input at IN (*data input*).

During a bit-serial I/O operation, the I/O ports to which serial bits of data are sent or from which they are received are identified by I/O addresses instead of memory addresses. These addresses are also output on the system address bus, A_0 through A_{14}. They are used either to control external multiplexing and decoding circuits that channel the bits of data output at OUT to the appropriate output port, or to select data from specific input ports and pass them onto the IN line.

Mnemonic	Name	Function
$\overline{\text{IOCLK}}$	Input/output clock	pulsed to logic 0 as each bit of bit-serial data is output
OUT	Data output	data output for the bit-serial I/O interface
IN	Data input	data input for the bit-serial I/O interface

Figure 2.4 I/O signals.

Status Bus and Bus Status Codes

The 99000 outputs codes on *status bus lines* BST_1 through BST_3 which identify the type of bus cycle that is being executed. Figure 2.5 is a list of the mnemonics and names of the 16 types of bus status codes. These codes, together with the $\overline{MEM}$ signal, indicate to external circuitry whether the MPU is performing a memory cycle, internal cycle, I/O cycle, or special function such as a hold-state acknowledge or interrupt acknowledge. They are decoded in external circuitry to produce control signals for demultiplexing the address, data, and serial I/O buses.

Notice that the codes repeat for the 0 and 1 logic levels of $\overline{MEM}$. The group where $\overline{MEM} = 0$ corresponds to memory cycles. For example, the *instruction acquisition* (IAQ) *code* $BST_1BST_2BST_3 = 011$ indicates that an instruction is being fetched over the data bus.

The other group, which occurs with $\overline{MEM} = 1$, corresponds to internal cycles and special functions such as IO, HOLDA, and RESET. An example is the IO code 011, which signals that an I/O operation is taking place.

$\overline{MEM}$	BST_1	BST_2	BST_3	Mnemonic	Name
0	0	0	0	SOPL	Source operand with MPILCK
0	0	0	1	SOP	Source operand
0	0	1	0	IOP	Immediate operand
0	0	1	1	IAQ	Instruction acquisition
0	1	0	0	DOP	Destination operand
0	1	0	1	INTA	Interrupt acknowledge
0	1	1	0	WS	Workspace
0	1	1	1	GM	General memory
1	0	0	0	AUMSL	ALU or macrostore with MPILCK
1	0	0	1	AUMS	ALU or macrostore
1	0	1	0	RESET	Reset
1	0	1	1	IO	Input/output
1	1	0	0	WP	Workspace pointer
1	1	0	1	ST	Status register
1	1	1	0	MID	Macroinstruction detect
1	1	1	1	HOLDA	Hold acknowledge

Figure 2.5 Bus status codes. (Courtesy of *Texas Instruments, Incorporated.*)

Interrupt Interface

The interrupt interface is used to signal the 99000 that an external device is requesting service. There are three types of hardware interrupt inputs on the 99000: the *reset function*, *nonmaskable interrupt*, and *external user-definable interrupts*.

The signal lines of the interrupt interface are listed in Fig. 2.6. The $\overline{RESET}$ (*reset*) interrupt input is used as a hardware initialization input to the microprocessor. On the other hand, the $\overline{NMI}$ line represents the nonmaskable interrupt input. It is typically used to implement a nonmaskable service request for an external device. The other five lines, $\overline{INTREQ}$ and IC_0 through IC_3, represent a prioritized maskable interrupt interface. They can be used to implement user-defined interrupt requests. Signal $\overline{INTREQ}$ is used to tell the 99000 that a valid code is present, while the binary combination at $IC_0IC_1IC_2IC_3$ identifies the priority level of the external device that is requesting service.

Mnemonic	Name	Function
$\overline{INTREQ}$	Interrupt request	0 signals that an external device is requesting service by an interrupt
$IC_0\,IC_1\,IC_2\,IC_3$	Interrupt code	code that identifies the external device requesting service
$\overline{NMI}$	Nonmaskable interrupt	0 indicates that the nonmaskable interrupt is active
$\overline{RESET}$	Reset	0 indicates that the hardware reset function is active

Figure 2.6 Interrupt interface signals.

Direct Memory Access Interface

The DMA (*direct memory access*) interface, which is identified in Fig. 2.2(a), is used to put the 99000 into what is known as the *hold state*. In this state, the microprocessor gives up control of the system bus. It does this by putting the address/data bus and control lines into the high-Z (*high-impedance*) state and then suspending operation. In this way, an external device such as a *DMA controller* can take control of the system bus and has the ability to access directly the memory subsystem of the microcomputer.

The $\overline{HOLD}$ line is for input of the external signal that is used to initiate transition to the hold state. When the 99000 enters this state, it acknowledges this fact to external circuitry by outputting a HOLDA (*hold acknowledge*) code on the status bus.

Attached Processor Interface

The attached processor interface corresponds to the signal line $\overline{APP}$ (*attached processor present*) in the block diagram of Fig. 2.2(a). The $\overline{APP}$ signal indicates to the 99000 that an *attached processor* in the system is ready to perform a function. In response, the 99000 suspends operation and passes control to the attached processor. The attached processor performs the required function and then returns control to the 99000.

Clock Signals

The 99000 contains internal clock drive circuitry. Therefore, to operate, it needs only an external *crystal*. The crystal is attached between the $XTAL_1$ and $XTAL_2$ leads as shown in Fig. 2.7(a). Note that shunt capacitors are required from each of these terminals to ground. C_1 and C_2 each have a typical value of 5 pF. The maximum rating for the crystal is 24 MHz.

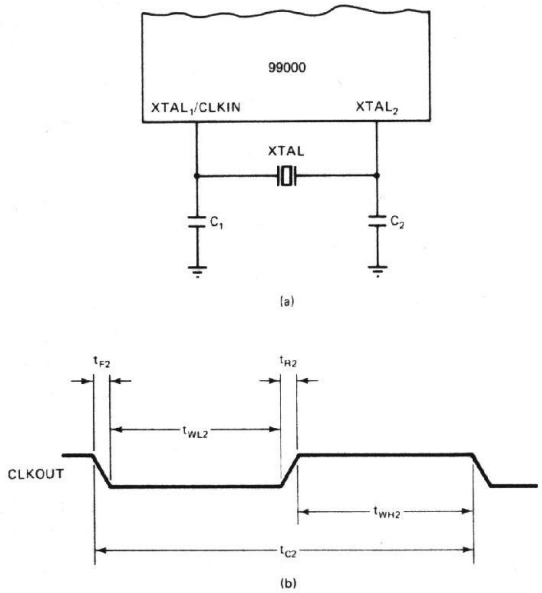


Figure 2.7(a) Crystal connection; **(b)** CLKOUT waveform. (Courtesy of *Texas Instruments, Incorporated.*)

Internally, the 24-MHz clock frequency is divided by 4 to give 6-MHz internal operation. This corresponds to a *machine cycle time* of

$$t_{\text{cycle}} = \frac{1}{6\text{ MHz}} = 167\text{ ns}$$

This time is also known as the *machine state time* of the 99000. The minimum duration that an external *bus cycle* requires is one machine state, or 167 ns.

The 6-MHz system clock is provided as an output at the CLKOUT line. Figure 2.7(b) shows this waveform. Notice that the *cycle time* of CLKOUT is labeled t_{C2} and has a nominal value of 167 ns. Its one-level pulse width t_{WH2} and zero-level pulse width t_{WL2} both have a typical value of 73.5 ns. The *rise* and *fall time* between the 0 and 1 levels are called t_{R2} and t_{F2}, respectively, and have the identical maximum value of 15 ns. This signal is provided for use in synchronizing the operation of external circuitry to that of the 99000 MPU.

If preferred, operation of the 99000 can be synchronized to an externally produced clock signal instead of generating it internally. This is done by applying the clock signal to the CLKIN input.

2.4 INTERNAL ARCHITECTURE OF THE 99000 MICROPROCESSOR

Now that we have introduced the 99000 microprocessor and looked briefly at its interface signals, let us proceed by examining its internal architecture.

Functional Blocks of the 99000

Figure 2.8 shows the *internal architecture* of the 99000. Here we find that its key sections are the *arithmetic-logic unit*, *internal registers*, *microcontrol* and *control ROM*, *interrupt logic*, *MQ shift register*, *clock generator*, and *macrostore*.

The *arithmetic-logic unit* (ALU) is the heart of the 99000 microprocessor. It is responsible for performing the mathematical operations or logical decisions that are required during the execution of an instruction. Notice that it has two inputs, A and B. The ALU takes these inputs and performs a mathematical operation on them such as addition or subtraction or a logical operation such as OR, AND, or exclusive-OR. The result of this operation is provided at the output of the ALU. The ALU has the ability to perform operations on either bytes or words of data.

The *internal registers* of the CPU are used to store data and status required by the MPU. For instance, the instruction register is used to store the operation code (opcode) of the instruction that is being executed.

Three of the most important registers within the 99000 are its *program counter* (PC), *workspace pointer* (WP), and *status register* (ST). They are called *user-accessible registers*. This is because their contents can be loaded or saved under software control.

The *microcontrol* and *control ROM* sections are used to decode the instruction operation code that is held in the instruction register. Based on

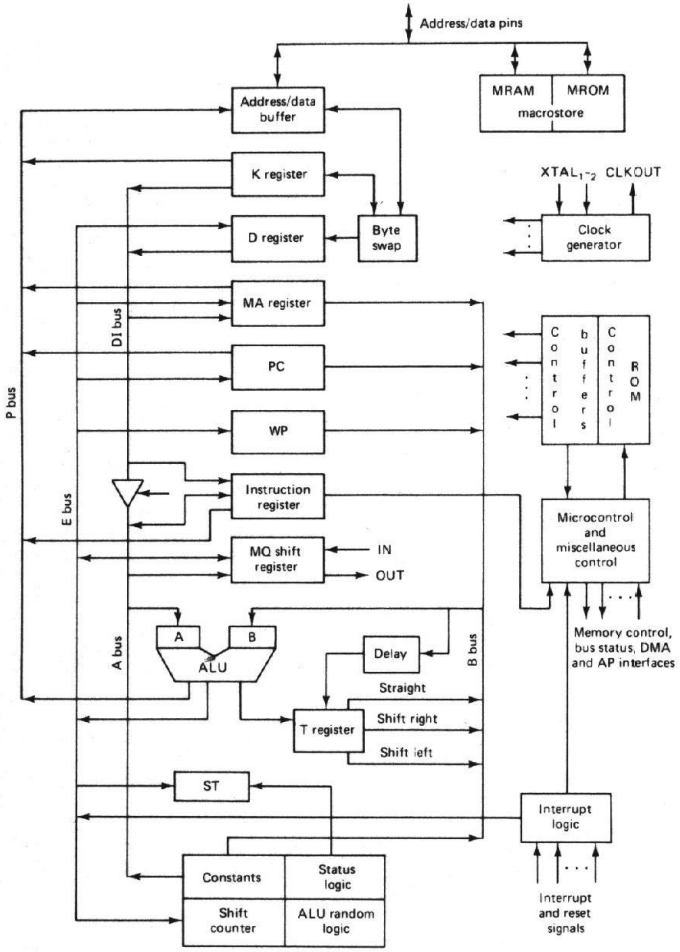


Figure 2.8 Internal architecture of the 99000. (Courtesy of *Texas Instruments, Incorporated.*)

this decoding, it outputs the buffered control signals. These signals control the sequence in which the 99000 performs internal and external operations such that the function specified in the instruction is performed.

The *interrupt logic* identifies that an external device is requesting service by the MPU through the interrupt interface. It indicates this fact to the microcontrol section.

The *MQ shift register* is the most important part of the serial I/O interface. It does the serial-to-parallel conversion required for input operations and parallel-to-serial conversion required for output operations.

The circuitry in the *clock driver* block generates the 6-MHz four-phase clock signals required by the 99000. These signals synchronize the internal and external operations of the 99000.

The last section, the *macrostore*, is an internal memory area of the 99000. It consists of a *macro-RAM* (MRAM) data storage section and a mask-programmable *macro-ROM* (MROM) instruction storage section. The purpose of macro-ROM is for on-chip storage of *macroinstruction emulation routines*. Typically, macroinstruction emulation routines are used to extend the instruction set of the 99000. This is done by implementing more complex functions, such as special instructions for floating-point arithmetic.

Memory-to-Memory Architecture

The *memory-to-memory architecture* is unique in that all of the registers used by the MPU for data operations and addressing are in external memory instead of internal to the device. This type of architecture is illustrated in Fig. 2.9(a). Moreover, these registers are simply memory locations and not random logic circuits such as those used in register-based microprocessor architectures.

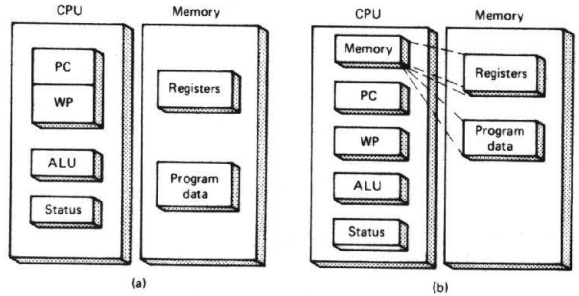


Figure 2.9(a) Memory-to-memory architecture; (b) memory-to-memory architecture with on-chip memory. (Courtesy of *Texas Instruments, Incorporated.*)

The important benefit of using memory instead of random logic for the registers of the MPU is that memory can be integrated more compactly. Memory registers can be moved onto the chip to be more consistent with traditional microprocessor architectures. However, because of the more compact nature of memory, a larger number of registers can be implemented. An example of a microprocessor that employs an on-chip register file is the 9995. The macrostore section of the 99000 is another example of on-chip memory. Its architecture is shown in Fig. 2.9(b).

User-Accessible Registers

As indicated earlier, by "user-accessible registers" we mean those registers whose contents can be accessed and altered through software. Figure 2.10 shows that the 99000 has just four of these registers: the *program counter* (PC), *workspace pointer* (WP), *status register* (ST), and *error register* (ER).

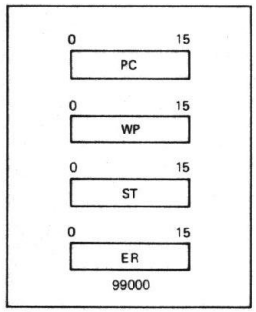


Figure 2.10 User-accessible registers.

Figure 2.11 shows the relationship between the internal registers of the 99000. The *program counter* (PC) is a 16-bit register that contains the address of the next instruction in the program that is to be fetched for execution or the address of an immediate operand that is required during execution of the current instruction. Actually, the least significant bit (LSB) of PC is always zero. This is because instructions are always accessed as words. The value in PC is multiplexed onto the address/data bus during the address phase of an instruction-acquisition bus cycle. The instruction word stored at this location in program storage memory is put on the data bus. During the data-transfer part of the bus cycle, the 99000 reads the word and stores it in its instruction register. After this, the value in PC is automatically incremented by 2 such that it points to the next instruction or operand of the program.

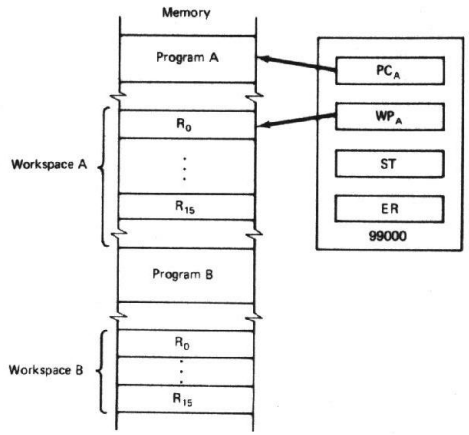


Figure 2.11 Relationship between the user-accessible registers. (Courtesy of *Texas Instruments, Incorporated.*)

On the other hand, the *workspace pointer* (WP) contains a 16-bit address that points to the first register in a block of 16 registers in data storage memory. These registers are also identified in Fig. 2.11. They serve as *source* and *destination registers* of data operands, *address registers*, or *index registers*. The LSB of this address is also always zero.

In Fig. 2.12, the 16-register *workspace* is shown in more detail. Here we see that the registers are labeled R_0 through R_{15}. Notice also that they are located at addresses that are displaced from the value in WP. The displacement value equals twice the register number. For instance, register R_2 is at address WP + 2(2) = WP + 4.

Example 2.1

Find the address of register R_{15} relative to the workspace pointer. Assume that the workspace pointer contains $F000_{16}$. Express the result in hexadecimal form.

Solution: Adding twice the register number to the contents of WP, we get

$$(WP) + 15(2)$$

$$(WP) + 30$$

Now replacing (WP) with $F000_{16}$ and expressing 30 in hexadecimal form gives the address of register 15 as

$$F000_{16} + 1E_{16} = F01E_{16}$$

Any of these 16 workspace registers can be used for the storage of data or addresses; however, some also have dedicated functions. For example,

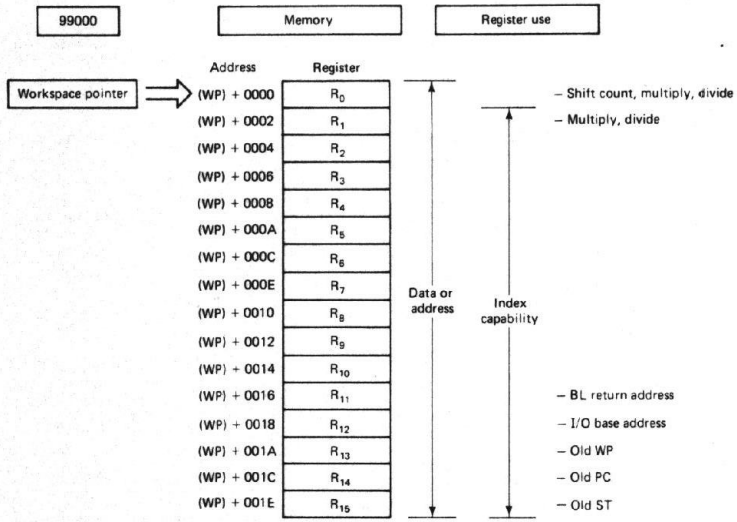


Figure 2.12 Workspace registers. (Courtesy of *Texas Instruments, Incorporated.*)

registers R_{13}, R_{14}, and R_{15} are used to save the old values of the workspace pointer, program counter, and status register, respectively, when a context switch initiates a change in program environment. Other examples are: register R_{12}, which is required for storage of a CRU base address for use in I/O operations; register R_{11}, which is reserved for storage of a return address when a "branch and link" (subroutine) instruction is executed; and register R_0, which is needed for storage of a shift count for some instructions.

The *status register* (ST) is another 16-bit internal register of the 99000. Figure 2.13 shows the individual bits of this register and their meanings. Notice that a number of these bits specify conditions that result during the execution of an instruction. Other bits are used to enable or disable functional capabilities of the 99000.

Bits in the group from ST_0 through ST_6 are those that are affected by the execution of instructions. These bits are either set or reset based on the results produced due to the execution of an instruction. For example, after the execution of an addition instruction, we might find changes in A> (*arithmetic greater than*), EQ (*equal*), and C (*carry*). However, it should be noted that not all instructions affect these status bits.

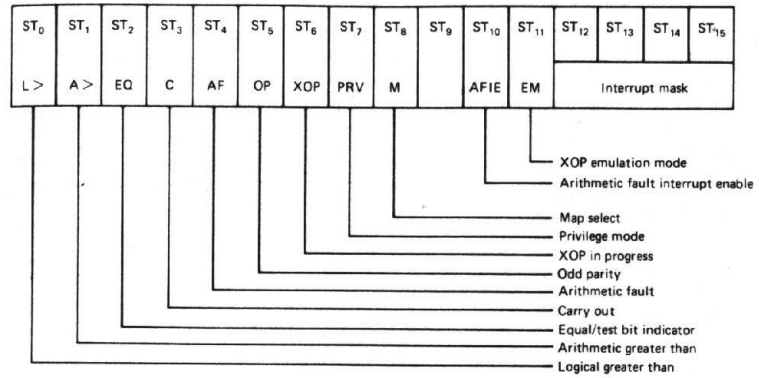


Figure 2.13 Status register.

Some instructions of the 99000 can be used to examine the status of specific bits in the status register and perform certain operations based on whether they are set or reset. An example is the "jump on equal" instruction (JEQ). It tests to see if the equal bit is set. If EQ is found to be set, a jump is initiated to another instruction located at a displaced PC value. Otherwise, the instruction at the next sequential value of PC is executed.

Example 2.2

Which of the status bits would you expect to be affected by an instruction that compares two operands?

Solution: The result of an operand comparison would be the fact that one is greater than, less than, or equal to the other. For this reason, we would expect changes in ST_0 through ST_2: logical greater than (L>), arithmetic greater than (A>), and equal (EQ).

Bits ST_7 through ST_{11} of the status register are those that enable or disable optional functions of the 99000. Examples are: ST_{10}, which is used to enable what is known as the *automatic arithmetic overflow detection mechanism*, and ST_7, which enables the *nonprivileged mode* of operation. These status bits must be set to the 1 logic level to perform their identified functions. For instance, setting PRV to 1 enables the nonprivileged mode of operation, and clearing it disables the nonprivileged mode.

The four least significant bits of the status register, ST_{12} through ST_{15}, hold an *interrupt mask code*. It always contains a 4-bit value equal to one less than that of the priority level of the currently active interrupt. In turn, all interrupts with lower priority are disabled from operating.

The last of the 99000's user-accessible registers is the *error register* (ER). Figure 2.14 shows that just three of its bits are internally implemented. These bits represent internal conditions that are considered to be errors when they occur within the 99000. For instance, if an *arithmetic overflow* occurs and this option is enabled through status bit ST_{10}, the error condition is flagged by automatically setting ER_4 to logic 1. This bit location can then be examined through software to identify the fact that an overflow has occurred.

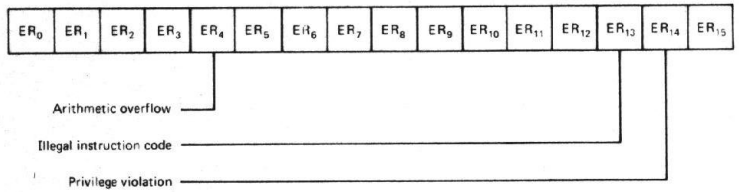


Figure 2.14 Error register.

The other two implemented bits of the error register, ER_{13} and ER_{14}, represent the *detection of an illegal instruction opcode* during the execution of the program and the occurrence of a *privileged mode violation*.

Example 2.3

What condition must exist in the status register before bit ER_{14} can be set due to the occurrence of a privileged mode violation?

Solution: The nonprivileged mode of operation must be enabled by setting PRV (status bit ST_7) to logic 1.

2.5 EXECUTION OF AN INSTRUCTION

Now that we have introduced the 99000 and its memory-to-memory architecture, let us continue by considering how it executes an instruction. As indicated in Section 2.4, the primary difference between register architecture and memory-to-memory architecture lies in the fact that memory-to-memory processors, such as the 99000, perform all operations directly on data operands in memory. Here we will describe in detail the effect of executing an addition instruction, the microstates that are performed during execution of the instruction, the 99000's instruction prefetch mechanism, and bus status codes.

Execution of an Addition Instruction

Figure 2.15(a) illustrates the execution of an instruction. Notice that the program counter points in program memory to the instruction

```
A  R0,R1
```

This instruction represents the addition of a source operand that is stored in workspace register R_0 to a destination operand that is located in register R_1. Their sum, which equals the original contents of R_0 plus the original contents of R_1, is returned to destination register R_1.

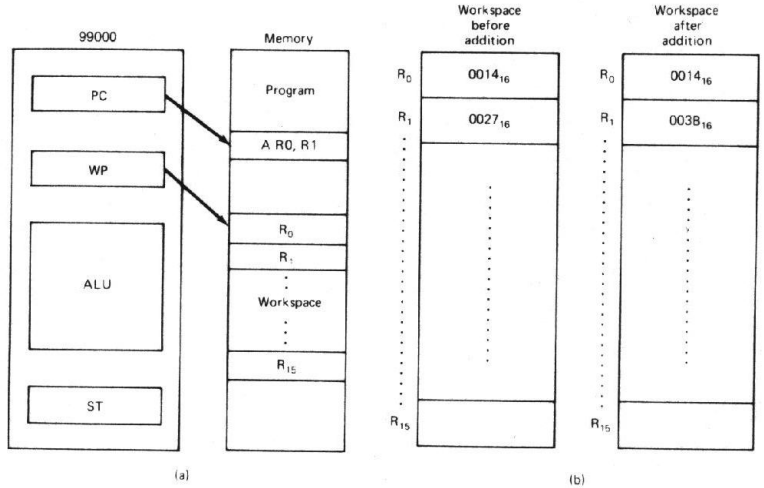


Figure 2.15(a) Execution of an instruction; (b) workspace registers before and after execution of the addition instruction. (Courtesy of *Texas Instruments, Incorporated.*)

The example in Fig. 2.15(b) indicates that before the addition instruction occurs, R_0 contains the data word 0014_{16} and R_1 contains 0027_{16}. Executing the add word instruction creates the sum

$$0014_{16} + 0027_{16} = 003B_{16}$$

and stores this value in R_1. The value held in R_0 remains unchanged. These results are shown in Fig. 2.15(b).

Steps in the Addition Instruction

The execution of an instruction actually requires the 99000 to go step by step through a series of internal and external operations. This sequence of events is *microcoded* into the control ROM section. When an instruction is fetched from program memory, it is loaded into the instruction register within the 99000. The microcontrol section decodes the opcode and initiates the appropriate *microcode sequence*. The control ROM outputs signals that are used to sequence and time the internal operations, and external bus operations that are required to perform the processing specified by the instruction.

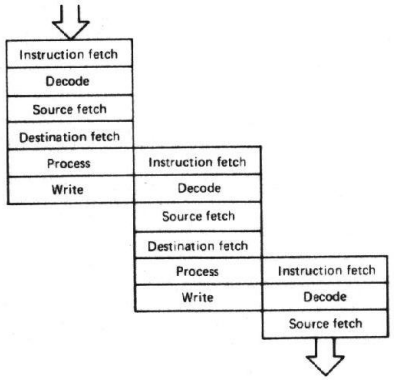
The actual sequence of operations that occur during the execution of the instruction

```
A  R0,R1
```

is shown in Fig. 2.16(a). Notice that there are six operations, of which four are memory bus cycles and two are internal cycles. The sequence begins with an *instruction acquisition memory bus cycle*. During this cycle, the instruction is fetched from the program memory. The instruction is loaded into the instruction register of the microprocessor. This represents the *instruction fetch* part of the instruction acquisition cycle.

Step	Function	Type
1	Fetch instruction	Memory
2	Decode instruction	Internal
3	Fetch source operand	Memory
4	Fetch destination operand	Memory
5	Add operands	Internal
6	Write result to destination	Memory

(a)



(b)

Figure 2.16(a) Register-to-register addition sequence; (b) prefetch mechanism. (Courtesy of *Texas Instruments, Incorporated.*)

The *execution phase* begins with an internal operation during which the instruction is decoded and the appropriate microcode sequence initiated. The microcode sequence next performs a *workspace memory bus cycle* during which the source operand is read from source register R_0. This operand is applied to the A input of the ALU. After this, a second workspace memory bus cycle is initiated to read the destination operand from destination register R_1. It is applied to the B input of the ALU.

Now the microprocessor has the data that it needs and is ready to perform the addition operation. This processing of operands represents the second internal operation shown in Fig. 2.16(a). The add operation is performed and the sum is obtained. The result must now be returned to destination register R_1 in memory. For this reason, a workspace write bus cycle is initiated.

Intelligent Prefetch Mechanism

Looking at the instruction sequence in Fig. 2.16(a), we see that during the two internal operations the system bus to external memory is not busy. To make more efficient use of the system bus, the 99000 is implemented with an *intelligent instruction prefetch mechanism*. With this mechanism, the execution of consecutive instructions is overlapped. This feature is also known as *pipelining*.

Figure 2.16(b) illustrates how the prefetch mechanism works. Notice that while the first instruction is being processed, the 99000 initiates an instruction acquisition memory bus cycle to fetch the next instruction. This second instruction is decoded while the results from the first instruction are being written to memory. In this way, we see that during each of the six steps of the addition instruction, the system bus is in use. This effectively eliminates any overhead due to the instruction acquisition part of the instruction execution cycle, thereby decreasing the amount of time it takes to execute an instruction. The results of prefetching instructions are improved system throughput, more efficient use of the system bus, and expansion of the bus bandwidth.

Earlier we said that the prefetch mechanism of the 99000 is intelligent. This is because it has the ability to detect automatically whether or not the present instruction is a branch or a jump type of instruction and can calculate the new program counter value before another instruction is fetched. Instead of fetching the next sequential instruction and then having to discard it, the program counter has already been modified such that the new instruction specified by the branch or jump is read and executed. This microcode sequence is shown in Fig. 2.17.

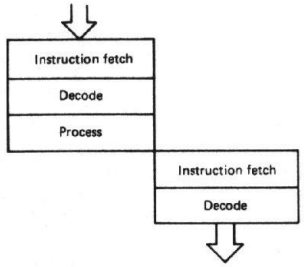


Figure 2.17 Prefetch sequence for the jump instruction.

Bus Status Codes of the Addition Instruction

During each step of the instruction execution sequence, a bus status code is output on status bus lines $\overline{MEM}$ and BST_1 through BST_3. As shown earlier, the 4 bits of this code are organized as $\overline{MEM}BST_1BST_2BST_3$ and the code indicates to external circuitry what type of operation is being performed by the 99000.

To understand this idea better, let us look at the bus status codes that occur during execution of the addition instruction we have been using as an example. The internal and external operations that take place during the execution of the register-to-register addition instruction are shown in Fig. 2.18(a). Notice that during step 1 the addition instruction is fetched from memory. This represents an instruction acquisition bus cycle and is accompanied by the IAQ code, which equals 0011, on the status bus.

Step 2 corresponds to the decoding of the addition instruction. This is an internal operation. But at the same time an external bus operation is performed to write the results of the previous instruction to memory. It is this destination write that determines which bus status code is output. Since we have not indicated what type of instruction this was, its mnemonic and status code are represented by a *don't know state* (X).

During the next two steps, the source and destination operands are fetched from memory. Since the data are stored in registers R_0 and R_1 of the workspace, their data bus cycles are accompanied by the *workspace transfer* (WS) *bus status code*, 0110.

The next operation corresponds to the internal addition of the two operands. A memory operation is also performed during this interval to prefetch the next instruction. Therefore, it is again accompanied by the IAQ code. During the last step, the new instruction is decoded and the results of

Step	Internal	External	Status code name	Status code
1	Process previous instruction	*Fetch addition instruction	IAQ	0011
2	Decode addition instruction	Write result of previous instruction	XXX	XXXX
3		*Read source operand for add instruction	WS	0110
4		*Read destination operand for add instruction	WS	0110
5	Add source and destination operands	Fetch next instruction	IAQ	0011
6	Decode next instruction	*Write result of addition to destination	WS	0110

*Memory cycles associated with the addition instruction

(a)

Step	Internal	External	Status code name	Status code
1	Process previous instruction	Fetch jump instruction	IAQ	0011
2	Decode jump instruction	Write results of previous instruction	XXX	XXXX
3	Process jump instruction	No operation	AUMS	1001
4	No operation	Fetch instruction from new location	IAQ	0011
5	Decode new instruction	No operation	AUMS	1001

(b)

Figure 2.18(a) Status codes for the addition instruction execution sequence; (b) status codes for execution of the jump instruction.

the addition instruction are written to memory. The destination register R_1 is in the workspace; therefore, the WS bus status code is again output.

The bus status codes that are output during a jump instruction are shown in Fig. 2.18(b). In this case, the 99000 processes the instruction immediately after decoding. During this time, no bus activity takes place; therefore, the *internal arithmetic logic unit bus status code* AUMS = 1001 is output. This is followed by the fetch of an instruction from the new address location accompanied by bus status code IAQ = 0001. Finally, the new instruction is decoded without an external bus cycle occurring for the jump instruction. This gives another AUMS = 1001 bus status code.

ASSIGNMENT

Section 2.2

1. Name the technology used to fabricate the 99000 family of microprocessors.
2. Name three advanced features of the 99000 that are not available on the older 9900 microprocessor.
3. What is meant by "macrostore"?
4. What is the main distinction between the 99105A and 99110A devices?

Section 2.3

5. What is the size of the 99000's package?
6. Which lines are used to output memory addresses?
7. Which lines form the data bus?
8. What function is served by the READY input?
9. Which line is used to input data during a serial input operation?
10. What lines are involved in the selection and output of data to an output port if the output operation is performed through the serial output interface?
11. What function is served by the bus status lines?
12. How many maskable interrupts are allowed in a 99000 system?
13. How does the 99000 acknowledge a DMA request?

Section 2.4

14. What are the three main user-accessible registers of the 99000?
15. What is the function of the ALU?
16. What function is served by microcontrol and the control ROM?
17. Which register is involved in serial I/O operations?
18. Specify the purpose of both MRAM and MROM.
19. What is meant by a memory-to-memory architecture? How does it differ from a register-based microprocessor architecture?
20. What function is served by the program counter?
21. Explain the function of the workspace pointer.
22. If the WP is loaded with 2000_{16}, identify each of the workspace registers and give their addresses.
23. Which workspace registers do not have an associated dedicated function?
24. What is the purpose of the status register?
25. Why is an error register provided within the 99000?

Section 2.5

26. After an instruction is fetched, which part of the processor initiates its execution steps?
27. Define "prefetch mechanism." How does it affect the processor's speed of program execution?
28. What signals are used by the processor to identify external memory operations during an instruction's execution? Which lines carry these output signals?

3

99000 MICROPROCESSOR PROGRAMMING I

3.1 INTRODUCTION

Having introduced the architecture of the 99000 microprocessor in Chapter 2, we are now ready to begin investigating its instruction set and some elementary programming concepts. In this chapter we begin by developing a software model of the 99000. This is followed by material that introduces the concepts of assembly language and machine language together with the symbols, notations, and formats used when coding them. Next, the addressing modes of the 99000 are described and their functions demonstrated. After this, we take our first look at the instruction set of the 99000. The data transfer, arithmetic, logical, and shift instructions are described in detail. The topics are presented in the following order:

1. Software model of the 99000
2. Assembly and machine languages
3. Instruction execution notations
4. Addressing modes
5. Instruction set
6. Data transfer instructions
7. Arithmetic instructions
8. Logic instructions
9. Shift instructions

3.2 SOFTWARE MODEL OF THE 99000 MICROPROCESSOR

The purpose of developing a software model is to aid the programmer in understanding the operation of the microcomputer system from a software point of view. To be able to program a microprocessor, one does not need to know all of its hardware features. For instance, we do not necessarily need to know the function of the signals at its various pins, their electrical connections, or their switching characteristics. Moreover, the function, interconnection, and operation of the internal circuits of the microprocessor also need not normally be considered.

What is important to the programmer is to know the various registers within the device and understand their purpose, functions, and operating capabilities and limitations. Furthermore, it is essential to know how external memory is organized and how it is addressed to obtain instructions and data.

A software model for the 99000 that includes its internal registers and external memory area is illustrated in Fig. 3.1. Looking at this model, we find four internal registers: the program counter (PC), workspace pointer (WP), status register (ST), and error register (ER). Each of these registers was discussed in detail in Chapter 2. For this reason, let us simply review them briefly here. This time we will concentrate on their relationship to software.

The program counter is a 16-bit register that produces word addresses for accessing the program storage part of memory. From a software point of

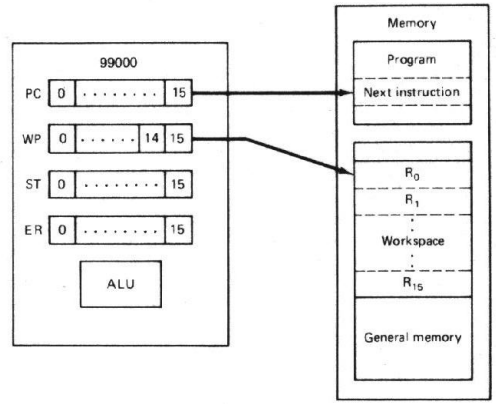


Figure 3.1 Software model of the 99000 microprocessor.

view, we must know that at any instant the contents of PC represent the address of the next instruction that is to be fetched for execution, or immediate data for the current instruction. In this way, we see that it determines the sequence in which instructions are executed.

In the 99000 microcomputer system, sequential word addresses differ by 2 and all instructions are stored at *word address boundaries.* Therefore, PC is automatically incremented by 2 after each instruction fetch. One exception is if the instruction that is currently being executed has the ability to modify the value in PC. In this case, the instruction is not fetched from the next sequential location in memory. Instead, the instruction fetch is from somewhere else in program storage memory. Examples of instructions that have the ability to modify the value in PC are those in the jump and branch groups.

When processing data, the 99000 uses a file of 16-word-wide registers in external memory as a working area. This area is the workspace and its registers are workspace registers R_0 through R_{15}.

These registers can be used to store intermediate results, pointers, counters, or any other type of information. For example, the addition instruction

```
A  R0,R1
```

uses register R_0 as the storage location of its source operand and R_1 as the storage location of its destination operand. The result of the addition ends up in destination register R_1.

As indicated in Chapter 2, some of the registers serve special functions. For instance, R_{13}, R_{14}, and R_{15} are required to save the old contents of internal registers WP, PC, and ST, respectively, whenever a context switch is initiated.

Registers in the workspace are accessed by the 99000 using the address held in its workspace pointer register. The contents of WP is the address of the first register in the workspace. This is the register denoted by R_0 and is located at the lowest workspace address. Addresses of the other registers in the workspace are formed by adding a number equal to twice the register number to the value in the workspace pointer register. For example, if WP contains 0120_{16}, R_0 is located at 0120_{16} and R_1 is located at $0120_{16} + 2_{16} = 0122_{16}$.

It is necessary to initialize the WP at the beginning of the program with the value of the address for the starting workspace. This can be done by executing a special instruction that is provided for loading the workspace pointer register from memory. If WP is not loaded correctly, instructions that involve workspace register references may not execute correctly.

During execution of the program, new workspaces can be created by simply changing the value in WP. This can be done under software control

with the "load workspace" instruction. Moreover, certain functions, such as interrupts and extended operations, automatically change the value in WP.

The status register (ST) is also 16 bits long. It keeps track of the status of the processor as the instructions of the program are executed. From a software point of view, the first 6 bits of this register, ST_0 through ST_5, are the most important. They are the logical greater than (L>), arithmetic greater than (A>), equal (EQ), carry (C), arithmetic fault (AF), and odd-parity (OP) bits, respectively. These flags reflect the status of the processor that results due to the execution of an instruction. For example, the carry bit is set to indicate when a carry out occurs from the MSB as a result of an arithmetic operation.

Software can reference the flag bits of the status register and, based on their logic level, make a decision. For instance, a "jump on carry" instruction checks the carry bit. If C is logic 0, the next sequential instruction executes. On the other hand, if it is 1, a jump is initiated to another place in the program.

The error register is the last of the 99000's internal user-accessible registers. It is also 16 bits long; however, just 3 of its bits are implemented. They are the arithmetic overflow bit ER_4, illegal opcode detection bit ER_{13}, and privileged mode violation bit ER_{14}. These bits are set if their corresponding error condition occurs during the execution of an instruction.

External memory in the 99000 microcomputer system is organized as 16-bit words. These words are selected by the 15-bit address output on lines A_0 through A_{14}. Notice that external addresses are 15 bits long, not 16 as in the internal PC and WP registers. Their LSB is truncated when the address is output to the memory subsystem. Thus memory word addresses proceed as 0000_{16}, 0002_{16}, 0004_6 up through $FFFE_{16}$. This gives a total memory storage capacity of 32K words.

Some of the instructions of the 99000 also allow access to these word locations in memory as an odd or even byte. Which byte is to be accessed is determined by the LSB of the address, A_{15}, which is retained internal to the 99000. For this reason, byte addresses proceed sequentially as 0000_{16}, 0001_{16}, 0002_{16} through $FFFF_{16}$. Even-addressed bytes reside at even address boundaries and odd-addressed bytes at odd address boundaries. Therefore, the address space of the 99000 can also be expressed as 64K bytes.

The memory address space of the 99000 can be expanded in a *paged mode* to 64K words (128K bytes). This is done by using an extended 16-bit address that consists of A_0 through A_{14} and $\overline{PSEL}$. ST_8 is the *map select bit* of the status register. Its logic level is complemented and then multiplexed onto the $D_{15}/\overline{PSEL}$ bus line during the address phase of all bus cycles. This bit can be manipulated under software control to select one of two 64K-byte pages of memory.

For the purpose of programming, memory can be viewed as two independent sections: a program segment that contains instructions of the pro-

gram and a data segment that contains workspaces and general memory for storage of data. With this architecture, the 99000 has the ability to perform memory-to-memory operations. Since the workspace of the 99000 resides in memory, a *register-to-register operation* is also a memory-to-memory operation. For instance, we can directly add the contents of two workspace registers. Moreover, for input/output, data can be transferred directly between memory and I/O devices.

Using segmentation, the memory subsystem of the 99000 can be partitioned into a 128K-byte code segment and 128K-byte data segment. This gives a total address space of 256K bytes. To achieve this configuration, memory bus status codes must be decoded with external circuits to produce enable signals for the code segment and data segment of memory.

3.3 ASSEMBLY LANGUAGE AND MACHINE LANGUAGE

Now that we have introduced the software model of the 99000, let us continue with the concepts of *assembly language* and *machine language* instructions and the symbols, notations, and formats that are used in their descriptions. It is essential to become familiar with these ideas before attempting to learn the functions of the instructions in the instruction set and their use in writing programs.

Assembly Language Instructions

Assembly language instructions are provided to describe each of the basic operations that can be performed by a microprocessor. They are written using *alphanumeric symbols* instead of the 0s and 1s of the microprocessor's machine code. An example of a short assembly language program is shown in Fig. 3.2(a). The assembly language statements are located on the left. Frequently, comments describing the statements are included on the right. This type of documentation makes it easier for programmers to write, read, and debug code. By the term "code" we mean programs written in the language of the microprocessor. Programs written in assembly language are called *source code.*

Each instruction in the source program corresponds to one assembly language statement. The statement must specify which operation is to be performed and what data operands are to be processed. For this reason, an instruction can be divided into two separate parts: its opcode and its operands. The *opcode* is the part of the instruction that identifies the operation that is to be performed. For example, typical operations are add, subtract, or load immediate.

In assembly language, we assign a unique one-, two-, or three-letter combination to each operation. This letter combination is referred to as a

mnemonic for the instruction. For instance, the 99000 assembly language mnemonics for add, subtract, and load immediate are A, S, and LI, respectively.

Operands identify the data that are to be processed by the microprocessor as it carries out the operation specified by the opcode. For instance, in an instruction that adds the contents of two workspace registers R_0 and R_1, R_0 and R_1 are the operands. An assembly language description of this instruction is

```
A  R0,R1
```

```
      IDT  'EXAMPL'
*****************************************************************
* CLEAR SCREEN                                                  *
* THIS ROUTINE FILLS THE CHARACTER BUFFER WITH NULLS, AND       *
* MOVES THE CURSOR TO HOME POSITION.                            *
* CALLED BY THE MAIN ROUTINE ONLY.                              *
* NO PARAMETERS ARE PASSED.                                     *
*****************************************************************
      RORG
      OPTION XREF
      REF  ENDSCR,WCURCH,UPCURS,BEGSCR
CLRSCR LI   R8,BEGSCR        POINTS TO START OF CHAR BUFFER
      LI   R9,ENDSCR         POINTS TO END OF CHAR BUFFER
      MOV  R7,R6             GET FLAG REGISTER
      ANDI R6,>0080          IS SCREEN PROTECTED?
      JNE  CLR2              YES- CLEAR ONLY UNPROTECTED CHAR
CLR3  CLR  *R8+              CLEAR THE CHAR AND BUMP POINTER
      C    R8,R9             IS IT THE LAST BUFFER CHAR?
      JNE  CLR3              NO- LOOP
      JMP  HOMCUR            REPOSITION CURSOR TO "HOME"
CLR2  MOV  *R8,R6            GET THE CHARACTER
      ANDI R6,>8000          IS CHAR PROTECTED?
      JNE  CLR4              YES- SKIP THE "CLEAR" STEP
      CLR  *R8               CLEAR THE CHAR
CLR4  INCT R8                BUMP BUFFER POINTER
      C    R8,R9             IS IT THE LAST BUFFER CHAR?
      JNE  CLR2              NO- CLEAR NEXT BUFFER POSITION
*****************************************************************
* CURSOR HOME                                                   *
* THIS ROUTINE MOVES THE CURSOR TO THE BEGINNING OF THE         *
* TOP LINE ON THE SCREEN.                                       *
* PARAMETERS ARE INPUTTED IN R7 (5 BIT SCREEN LINE COUNTER)     *
* AND ARE PASSED TO THE TMS9927 CURSOR POSITION REGISTERS.      *
* CALLED BY THE MAIN ROUTINE ONLY.                              *
*****************************************************************
HOMCUR MOV  R7,R6            GET ABSOLUTE LINE POSITION
      ANDI R6,>001F          IS IT THE TOP LINE?
      JEQ  HMC1              YES- JUMP
      BL   @UPCURS           NO- MOVE THE CURSOR UP
      JMP  HOMCUR            LOOP UNTIL AT TOP
HMC1  CLR  @WCURCH           ZERO CURSOR CHARACTER REGISTER
      RTWP                   RETURN TO CALLING ROUTINE
      END
```

Figure 3.2(a) Typical 99000 assembly language program;

In this example the contents of R_0 and R_1 are added together and their sum is put in R_1. Therefore, R_0 is considered as the *source operand* and R_1 is the *destination operand.*

Here is another example of an assembly language statement:

```
                    LOOP    MOV     R0,R1    COPY R0 INTO R1
```

```
0001                        IDT  'EXAMPL'
0002                *****************************************************
0003                * CLEAR SCREEN                                      *
0004                * THIS ROUTINE FILLS THE CHARACTER BUFFER WITH NULLS, AND *
0005                * MOVES THE CURSOR TO HOME POSITION.                *
0006                * CALLED BY THE MAIN ROUTINE ONLY.                  *
0007                * NO PARAMETERS ARE PASSED.                         *
0008                *****************************************************
0009 0000                   RORG
0010                        OPTION XREF
0011                        REF  ENDSCR,WCURCH,UPCURS,BEGSCR
0012 0000 0208  CLRSCR LI   R8,BEGSCR       POINTS TO START OF CHAR BUFFER
     0002 0000
0013 0004 0209         LI   R9,ENDSCR       POINTS TO END OF CHAR BUFFER
     0006 0000
0014 0008 C187         MOV  R7,R6           GET FLAG REGISTER
0015 000A 0246         ANDI R6,>0080        IS SCREEN PROTECTED?
     000C 0080
0016 000E 1604         JNE  CLR2            YES- CLEAR ONLY UNPROTECTED CHAR
0017 0010 04F8  CLR3   CLR  *R8+            CLEAR THE CHAR AND BUMP POINTER
0018 0012 8248         C    R8,R9           IS IT THE LAST BUFFER CHAR?
0019 0014 16FD         JNE  CLR3            NO- LOOP
0020 0016 1008         JMP  HOMCUR          REPOSITION CURSOR TO "HOME"
0021 0018 C198  CLR2   MOV  *R8,R6          GET THE CHARACTER
0022 001A 0246         ANDI R6,>8000        IS CHAR PROTECTED?
     001C 8000
0023 001E 1601         JNE  CLR4            YES- SKIP THE "CLEAR" STEP
0024 0020 04D8         CLR  *R8             CLEAR THE CHAR
0025 0022 05C8  CLR4   INCT R8              BUMP BUFFER POINTER
0026 0024 8248         C    R8,R9           IS IT THE LAST BUFFER CHAR?
0027 0026 16F8         JNE  CLR2            NO- CLEAR NEXT BUFFER POSITION
0028                *****************************************************
0029                * CURSOR HOME                                       *
0030                * THIS ROUTINE MOVES THE CURSOR TO THE BEGINNING OF THE *
0031                * TOP LINE ON THE SCREEN.                           *
0032                * PARAMETERS ARE INPUTTED IN R7 (5 BIT SCREEN LINE COUNTER)*
0033                * AND ARE PASSED TO THE TMS9927 CURSOR POSITION REGISTERS. *
0034                * CALLED BY THE MAIN ROUTINE ONLY.                  *
0035                *****************************************************
0036 0028 C187  HOMCUR MOV  R7,R6           GET ABSOLUTE LINE POSITION
0037 002A 0246         ANDI R6,>001F        IS IT THE TOP LINE?
     002C 001F
0038 002E 1303         JEQ  HMC1            YES- JUMP
0039 0030 06A0         BL   @UPCURS         NO- MOVE THE CURSOR UP
     0032 0000
0040 0034 10F9         JMP  HOMCUR          LOOP UNTIL AT TOP
0041 0036 04E0  HMC1   CLR  @WCURCH         ZERO CURSOR CHARACTER REGISTER
     0038 0000
0042 003A 0380         RTWP                 RETURN TO CALLING ROUTINE
0043                   END
NO ERRORS, NO WARNINGS
```

Figure 3.2(b) assembled version of the program.

This instruction statement starts with the word LOOP. It is an address identifier for the instruction

```
MOV  R0,R1
```

This type of identifier is called a *label* or *tag.* The instruction is followed by "COPY R0 INTO R1." This part of the statement is called a *comment.* Thus a general format for writing an assembly language statement is

```
LABEL    INSTRUCTION    COMMENT
```

Machine Language

Before a source program can be executed by the microprocessor, it must first be run through a process known as *assembling.* This is normally done on a minicomputer or microcomputer with a program called an *assembler.* The result produced by this step is an equivalent program expressed in the *machine code* that is executed by the microprocessor. That is, it is the equivalent of the source program but is now written in 0s and 1s. This program is also referred to as *object code.*

Figure 3.2(b) is a listing that includes the machine language program for the assembly language program in Fig. 3.2(a). It was produced by a 99000 assembler. Reading from left to right, this list contains addresses of memory locations, followed by the machine code instructions, the original assembly language statements, and any comments. Note that for simplicity the machine code instructions are expressed in hexadecimal notation, not as binary numbers.

3.4 INSTRUCTION EXECUTION NOTATIONS

A standard notation is also used to describe the results produced by executing a 99000 instruction. An example is

$$(\text{SOURCE}) \rightarrow (\text{DESTINATION})$$

which indicates that the source operand is placed in the destination operand.

Symbols are used to indicate which operands are to be processed, what operation is to be performed on them, how the operands are to be accessed, and the destination to which the result is transferred. The table in Fig. 3.3 lists the symbols that are used to describe assembly language instructions of the 99000.

Symbol	Meaning
(.)	Contents of .
((.))	Contents of the memory location addressed the contents of .
Rn	Workspace register n
→	Copy the information on the left of → into the right of →
↔	Exchange the information on the two sides of ↔
@.	At the memory location .
>.	Hexadecimal .
*.	Indirect addressing using .
*. +	Indirect autoincrement addressing using .
@A(B)	Direct indexed addressing using A and B
$	Current content of PC
LSbyte	Least significant 8 bits of a word
MSbyte	Most significant 8 bits of a word

Figure 3.3 Symbols and notations used to describe instructions.

Let us begin by taking an example that uses a few of these symbols.

```
MOV  R1,R15
```

$$(R1) \rightarrow (R15)$$

This is a register-to-register move instruction. The parentheses around operands R_1 and R_{15} stand for "the contents of." Therefore, it means that the contents of register R_1 are copied into register R_{15}. At the end of the operation, both registers contain the same value.

Another example that appears similar but produces a very different result is

```
MOV  *R1,R15
```

$$(*R1) \rightarrow (R15)$$

This stands for "copy the contents of the memory location whose address is stored in register R_1 into register R_{15}." In this case, at the end of the operation, register R_1 still contains the address of the storage location that is to be accessed, but the value that is stored at this address is now also held in register R_{15}.

The move operation that we just described uses what is known as *indirect addressing*. It is denoted by the symbol * included with the description of the source operand.

Up to now our examples have always involved the destination as a workspace register. However, it can also be a storage location in general

memory. For this to be the case, we change the destination side of the expression by using the symbol @ before the description of the memory location. Here is how we would change our last example to show that the destination is the memory location called LOC instead of register R_{15}:

```
MOV  *R1,@LOC
```

$$(*R1) \rightarrow @LOC$$

Example 3.1

Show how to denote the operation that causes the value in memory location LOC to be swapped with the value in workspace register R_5.

Solution: To express this operation, we denote the source operand with @LOC, the destination operand with R_5, and the exchange of data with ↔. This gives

$$@LOC \leftrightarrow (R5)$$

3.5 ADDRESSING MODES

The purpose of *addressing* is to specify the location of a source or destination operand. There are a number of different ways in which the address of an operand can be generated by the 99000 microprocessor. These methods are known as its *addressing modes*.

The eight addressing modes that are available on the 99000 are: *immediate addressing*, *direct addressing*, *workspace register addressing*, *workspace register indirect addressing*, *workspace register indirect autoincrement addressing*, *indexed addressing*, *program counter relative addressing*, and *I/O relative addressing*. In this section we discuss each of these addressing modes in detail and demonstrate their use.

Immediate Addressing Mode

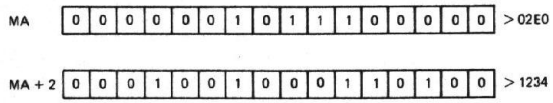
Let us begin with the immediate addressing mode. It is the simplest of all the addressing modes. In this case the value of the operand that is to be used in the execution of the instruction is put in the memory location that follows the instruction. That is, it immediately follows the opcode.

An example of an instruction written with immediate addressing mode is

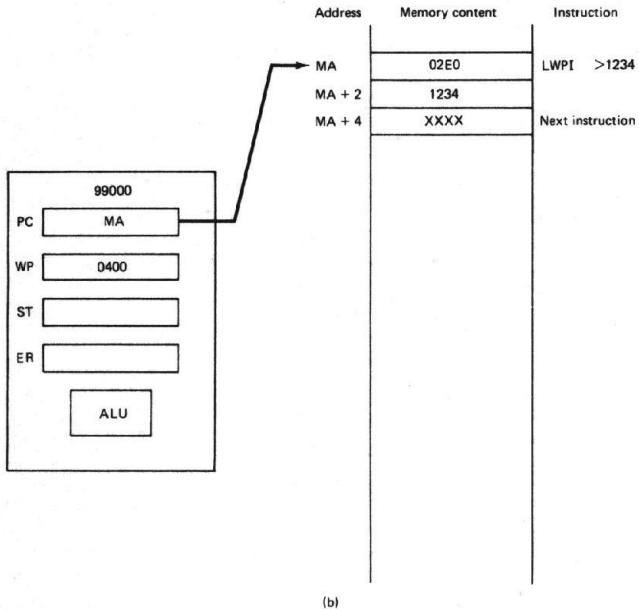
```
LWPI  >1234
```

This instruction loads workspace pointer register WP with the immediate value 1234_{16}.

In the instruction, the mnemonic LWPI is the opcode of the load workspace pointer immediate instruction and >1234 is the value of the source operand. When coded in machine language, this instruction takes up two consecutive word addresses in memory. As shown in Fig. 3.4(a), the first word contains the opcode for LWPI and is $02E0_{16}$. The second word contains the number that is to be loaded into WP. Here we find the immediate operand 1234_{16}.

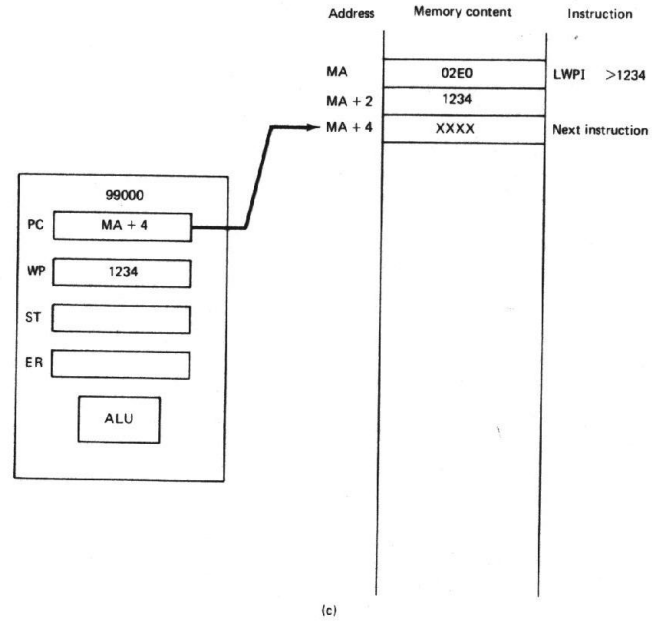


(a)



(b)

Figure 3.4(a) Coding of the LWPI > 1234 instruction; (b) immediate addressing-mode instruction before execution;



(c)

Figure 3.4(c) after execution.

The values in register WP before and after execution of the instruction are shown in Fig. 3.4(b) and (c), respectively. Notice that prior to execution of the instruction the value in WP is 0400_{16}. However, after execution, its contents are 1234_{16}, which is the value of the immediate operand.

Direct Addressing Mode

Direct addressing differs from the immediate addressing we just described in that the memory location following the instruction no longer contains the value of the opcrand. Instead, it contains the address of the operand that is to be processed during execution of the instruction.

Here is an example of an instruction that employs direct addressing for both its source and destination operands.

```
MOV  @>ABCD,@>1234
```

In the instruction, MOV represents the operation, >ABCD is the address of the source operand, and >1234 is the address of the destination operand. Therefore, it says to move the contents at address >ABCD to address >1234.

Figure 3.5 illustrates the result of executing this instruction. Looking at the before-execution memory state in Fig. 3.5(a), we see that the instruction word, which is coded as $C820_{16}$, is located at an arbitrary address in memory which is identified by MA. The next word address, MA + 2, contains $ABCD_{16}$. This is the address of the source operand. It is followed at address MA + 4 by the destination operand address, 1234_{16}. Notice that before the instruction is executed, the contents at the destination address are a don't-care state. On the other hand, the source address contains the data word $BEED_{16}$.

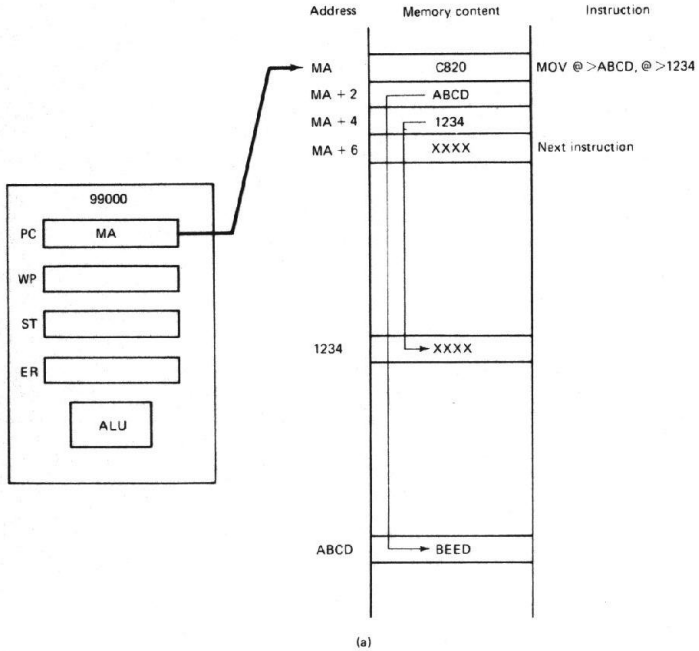


Figure 3.5(a) Instruction using direct addressing before execution;

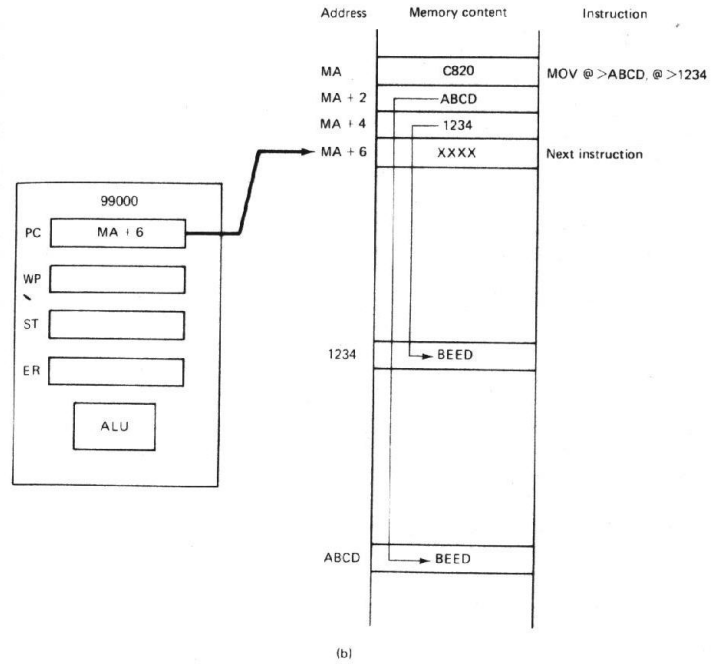


Figure 3.5(b) after execution.

As the instruction is executed, the data at $ABCD_{16}$ are moved to address 1234_{16}. Therefore, in the after-execution memory state of Fig. 3.5(b), both addresses, $ABCD_{16}$ and 1234_{16}, contain data word $BEED_{16}$. In this way, we see that the move instruction specified with the direct addressing mode copies the contents of one storage location in data memory into another location.

Workspace Register Direct Addressing Mode

The workspace register direct addressing mode is similar to the direct addressing mode except that this time a workspace register is specified for the location of the source or destination operand instead of the address of a location in data memory. For example, a MOV instruction can be written to

copy the contents of workspace register R_1 to workspace register R_5. The instruction reads

```
MOV  R1,R5
```

In this case, the actual memory addresses of the source and destination operands are derived from the value in the workspace pointer register and the register numbers specified in the instructions.

An example is illustrated in Fig. 3.6. Here we find that WP equals $A120_{16}$. Therefore, source register R_1 is located at address $A122_{16}$ and destination register R_5 at $A12A_{16}$. Before execution of the instruction, the contents of the destination register are a don't-care state and the contents of the source register are data word $ABCD_{16}$. As shown in Fig. 3.6(b), executing the instruction causes the value $ABCD_{16}$ to be copied into R_5.

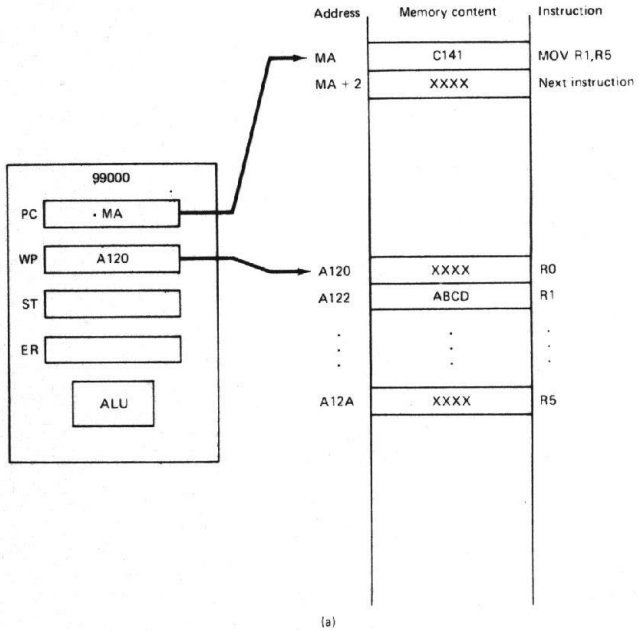


Figure 3.6(a) Instruction using workspace register direct addressing before execution;

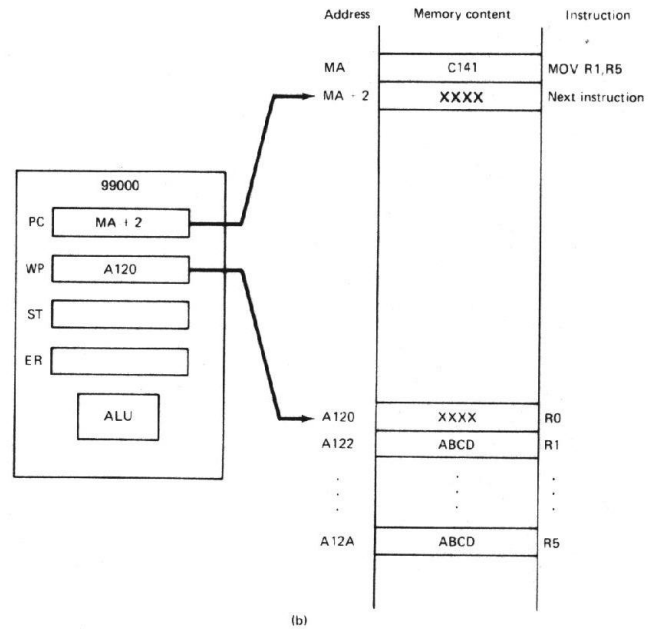


Figure 3.6(b) after execution.

Workspace Register Indirect Addressing Mode

Workspace register indirect addressing differs from workspace register direct addressing in that the register specified in the instruction contains the address of the operand instead of its value. This addressing mode is identified in the instruction by including the symbol * preceding the register symbol. Here is an example:

```
MOV  R1,*R3
```

In this statement, source operand R_1 is expressed using the workspace register direct addressing mode, but the destination operand R_3 uses the workspace register indirect addressing mode. This instruction means that the contents of R_1 should be moved to the memory location whose address is held in R_3.

Figure 3.7 demonstrates the execution of this instruction. Notice that the instruction is coded as $C4C1_{16}$ and is stored at address MA. If as shown in Fig. 3.7(a), R_1 contains $ABCD_{16}$ and R_3 contains 9122_{16}, the result obtained by executing the instruction is that the value $ABCD_{16}$ from R_1 is copied into the memory location at address 9122_{16}. This result is shown in Fig. 3.7(b). Thus the destination in memory was specified indirectly by the contents of R_3. This value must exist in R_3 prior to execution of the instruction. Moreover, note that it does not change due to the execution of the instruction.

Workspace Register Indirect Autoincrement Addressing Mode

This addressing mode is similar to the workspace indirect addressing mode we just described. However, an autoincrementing feature has been added. That is, after completing the operation specified by the instruction, the value

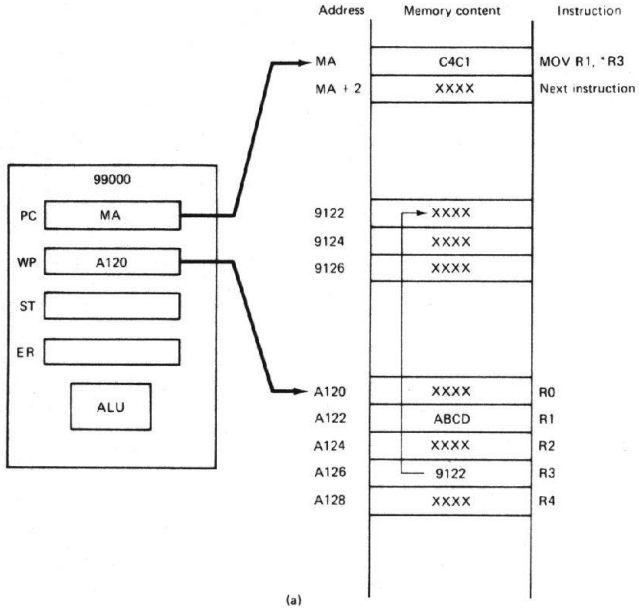


Figure 3.7(a) Instruction using workspace register indirect addressing before execution;

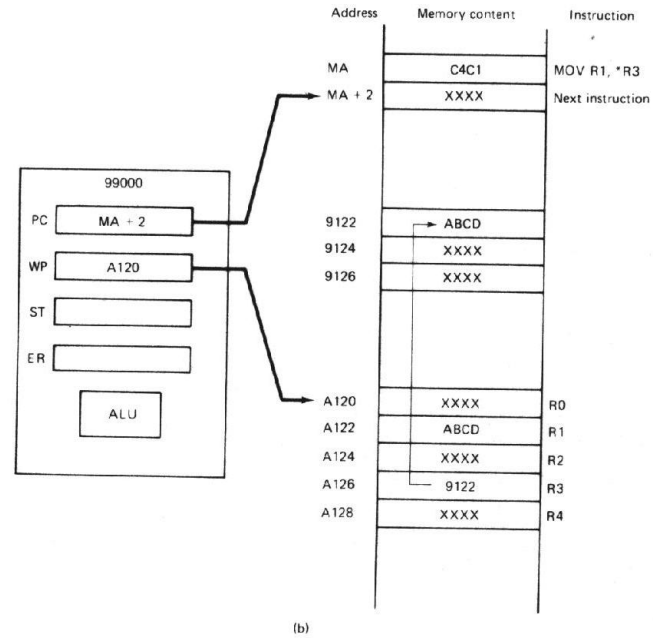


Figure 3.7(b) after execution.

of the indirect address in the workspace register is incremented automatically. The increment is by 2 for instructions that process words of data and by 1 for those that process bytes of data. In this way, it points to the next word or byte storage location in memory.

Here is an example of a MOV instruction in which the destination operand is specified with the workspace register indirect autoincrement addressing mode:

```
MOV  R1,*R3+
```

Notice that this time the destination register symbol is still preceded by * but that a + also follows the symbol. It is the "+" that signifies the autoincrement mode.

The diagram in Fig. 3.8 shows the effect of executing this instruction. Notice that the example is the same as that given for workspace indirect addressing in Fig. 3.7. In Fig. 3.8 we find that the contents of R_1, which are $ABCD_{16}$, are copied into the memory location at address 9122_{16}. However, there is a difference in the result. This time the original indirect address in R_3, which was 9122_{16}, has been incremented to 9124_{16}.

Indexed Addressing Mode

The indexed addressing mode is significantly different from those we have considered up to this point. In this case, a workspace register and base address are specified in the instruction to identify the location of the operand.

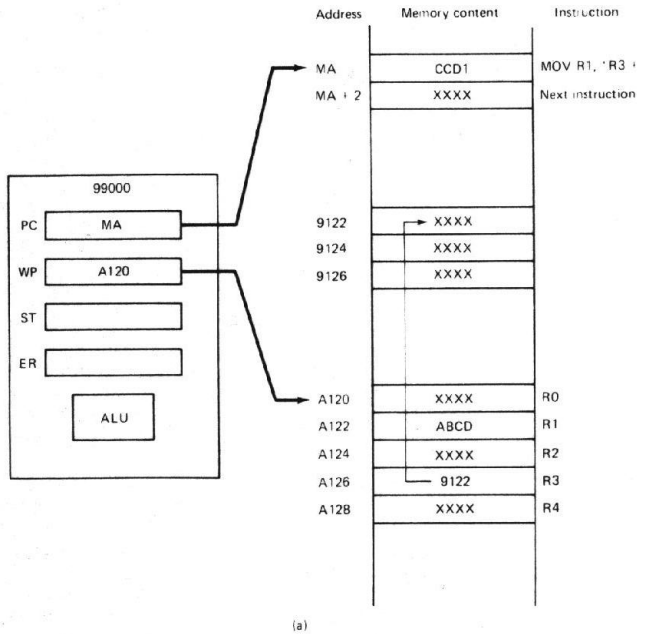


Figure 3.8(a) Instruction using workspace register indirect autoincrement addressing before execution;

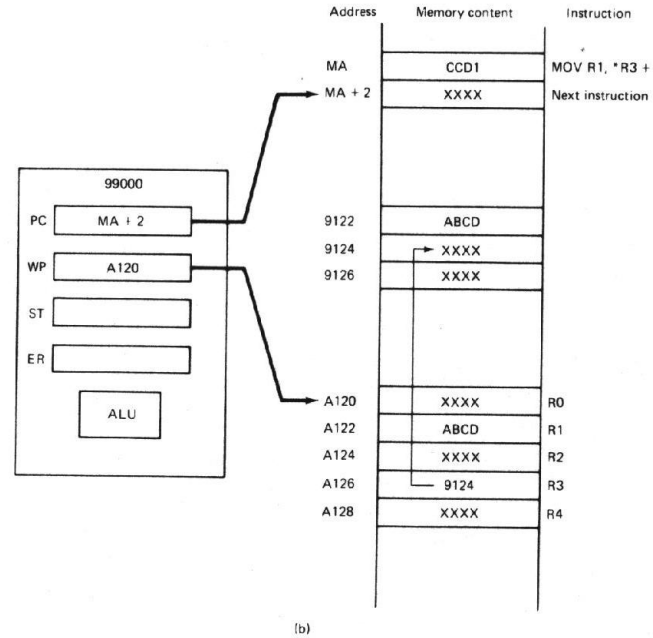


Figure 3.8(b) after execution.

The workspace register must already contain a value called the *index*. This index value is added to the base address to determine the address of the operand.

As an example, let us take the statement

```
MOV  R1,@>1234(R5)
```

Here the workspace register in the destination operand is enclosed in parentheses. This indicates that it holds an index value. The base address is expressed directly as 1234_{16}.

Figure 3.9 illustrates the execution of this instruction. Notice that the instruction, which is located at address MA, is coded as $C941_{16}$. This instruction word is followed by the base address 1234_{16} at MA + 2. Moreover, we

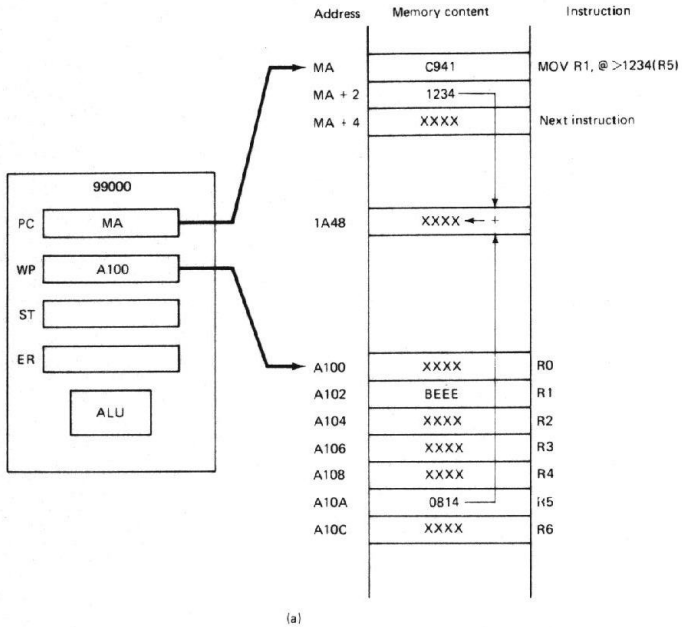
find that the data to be moved are represented by $BEEE_{16}$ in register R_1 and the index in R_5 is 0814_{16}.

The instruction calls for moving the contents of R_1 to the storage location whose address equals the sum of the base address 1234_{16} and the index 0814_{16}. Therefore, the effective address of the destination is

$$EA = 1234_{16} + 0814_{16} = 1A48_{16}$$

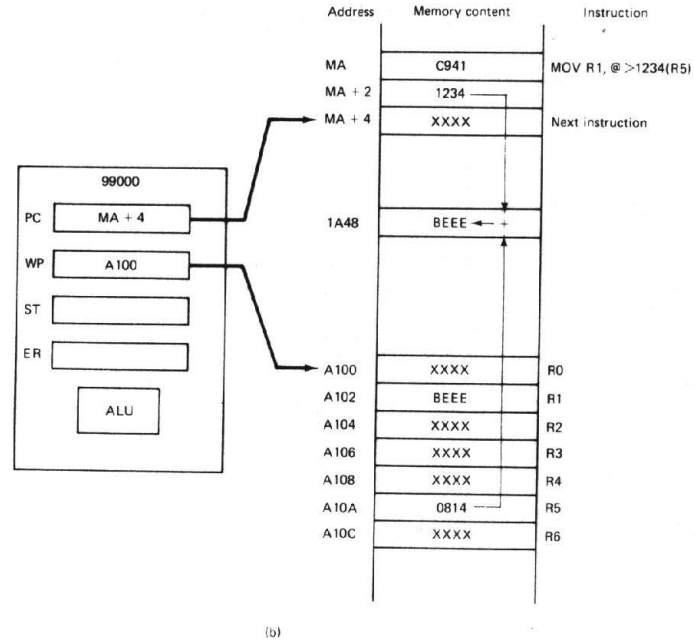
As shown in Fig. 3.9(a), the contents of this location are initially a don't-care state. But after execution of the instruction, the data word $BEEE_{16}$ has been copied into the address as shown in Fig. 3.9(b).

It should be noted that workspace register R_0 is not to be used for storing index values. This is a restriction that should be kept in mind whenever indexed addressing is being used. If R_0 is accidentally specified as the



(a)

Figure 3.9(a) Instruction using indexed addressing before execution;



(b)

Figure 3.9(b) after execution.

index register, the index value is automatically assumed to be 0000_{16}, even if R_0 contains a nonzero value.

Program Counter Relative Addressing Mode

Program counter relative addressing mode is not used to identify the location of an operand. Instead, it is used to specify a displacement relative to the present contents of the program counter. In this way, it can be used to pass control to another place in the program. This addressing mode is used only with jump instructions.

Typically, a jump instruction contains an 8-bit signed number called the *displacement*. It is this value that indicates how far forward (+) or backward (−) in the program the point is to which control is to be passed. The new ad-

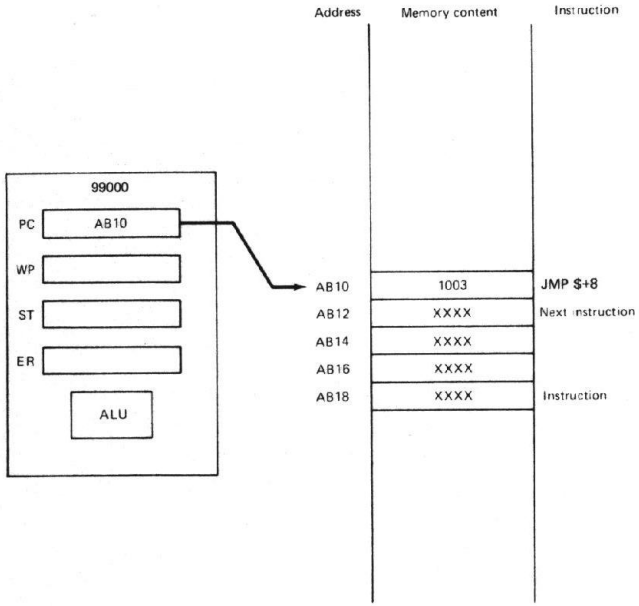
dress is computed automatically by adding 2 times the displacement to the updated contents of program counter PC.

An example of an instruction that uses this addressing mode is

```
JMP  $+8
```

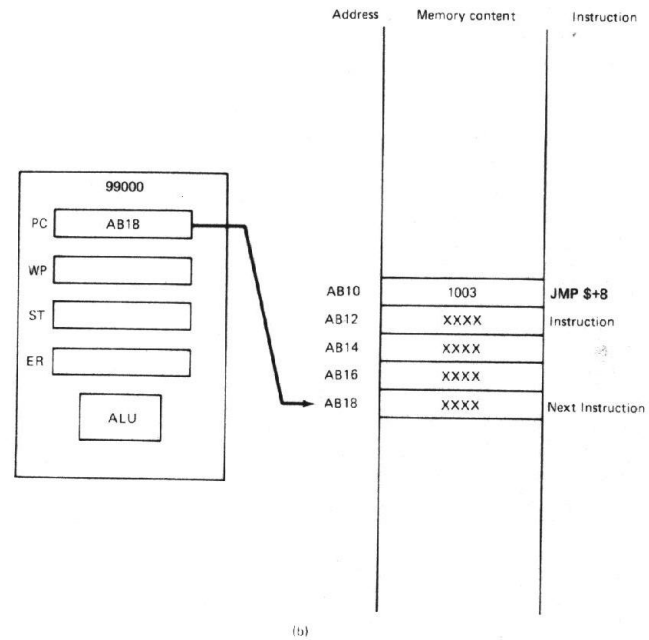
In this expression, $ indicates the current value in PC and +8 is the signed displacement. Execution of the instruction causes a jump to be initiated to a point that is four word addresses forward from the address of the current instruction.

For instance, Fig. 3.10(a) shows our example jump instruction coded as 1003_{16} and stored at address $AB10_{16}$. As it executes, PC is already updated



(a)

Figure 3.10(a) Instruction using program counter relative addressing before execution;



(b)

Figure 3.10(b) after execution.

to $AB12_{16}$. The 99000 calculates the new address as $AB12_{16} + 2 \times 0003_{16}$. This gives $AB18_{16}$. Thus, as shown in Fig. 3.10(b), the next instruction is fetched from this displaced location.

In a similar way, the instruction

```
JMP  $-8
```

specifies jumping back four memory word locations from that of the jump instruction.

The format of the JMP instruction provides 8 bits for specification of the displacement. With 8 bits, the range of jump permitted through program counter relative addressing is limited to +128 words or −127 words with respect to the current value in PC.

I/O Relative Addressing Mode

I/O relative addressing is the addressing mode that is used by the input/output instructions of the 99000. It relates to the I/O address space, not the memory address space. We will not introduce this addressing mode at this time. Instead, it will be considered in detail when the I/O instructions are introduced in Chapter 6.

3.6 INSTRUCTION SET

In Sections 3.1 through 3.5 we introduced the software model of the 99000, assembly language and machine language programming, and addressing modes. With this background, we are ready to begin our study of the *instruction set* of the 99000.

The 99105A microprocessor, which is the first processor available in the 99000 family, has a very powerful minicomputerlike instruction set that includes 84 instructions. It is known as the *baseline instruction set* of the 99000. This baseline instruction set can be expanded in macrostore memory. Based on their functionality, the instructions in the baseline instruction set can be categorized into groups. In this chapter we consider the instructions from four groups: the *data transfer instructions*, *arithmetic instructions*, *logic instructions*, and *shift instructions*.

3.7 DATA-TRANSFER INSTRUCTIONS

The instructions in the data-transfer group allow the programmer to initialize the values in registers or memory of the 99000 microcomputer system through software. The data-transfer group includes instructions that allow the internal WP and ST registers to be loaded with new values or to have their old values saved in memory. Moreover, there are instructions that permit data to be transferred between workspace registers and general memory.

For ease of understanding, we will subdivide the data-transfer group of instructions into three subgroups. The first group that we will look at consists of instructions known as the *immediate mode instructions*. This is because their source operand is always specified as an *immediate operand* and is accessed using the immediate addressing mode. That is, it is coded following the instruction's opcode in program memory.

There are three instructions in this category: *load immediate* (LI), *load interrupt mask immediate* (LIMI), and *load workspace pointer immediate* (LWPI). The meaning, format, function, whether result is compared to zero, and effect on status bits for each of these instructions are summarized in Fig. 3.11(a).

Let us first consider the "load workspace pointer immediate" instruction. The general format of this instruction is given in Fig. 3.11(a) as

```
LWPI  I
```

Here I identifies the immediate operand.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
LI	Load immediate	LI R,I	I → (R)	Y	L>, A>, EQ
LIMI	Load interrupt mask immediate	LIMI I	I → (ST_{12}-ST_{15})	N	ST_{12}-ST_{14}
LWPI	Load workspace pointer immediate	LWPI I	I → (WP)	N	None

(a)

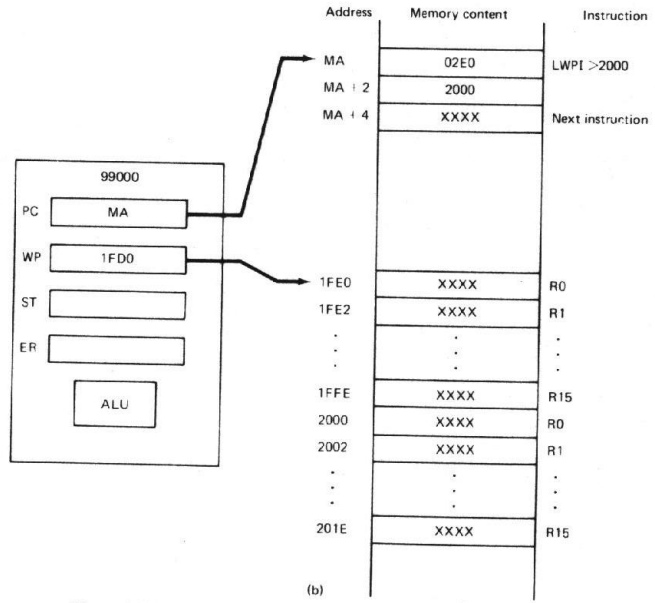


Figure 3.11(a) Immediate-mode instructions; (b) LWPI instruction before execution;

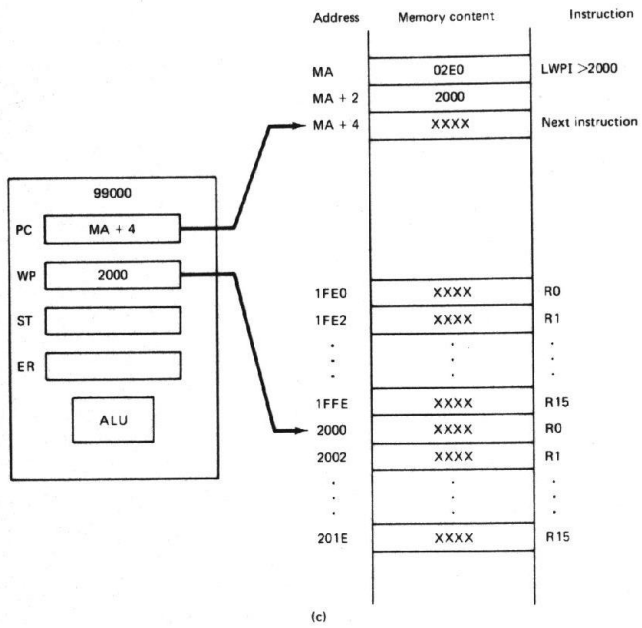


Figure 3.11(c) after execution.

The general description of this instruction's operation says that upon execution the immediate operand is loaded into the workspace pointer register within the 99000. Moreover, notice that execution of the instruction does not compare the result to zero or affect the status register.

An example of the instruction is

```
LWPI  >2000
```

When coded, this instruction takes up two memory locations: the first for the instruction word and a second for the immediate operand. Thus it is called a *two-word instruction.* As shown in Fig. 3.11(b), the first word, which in machine code is $02E0_{16}$, is located at an arbitrary address in program memory called MA. It is followed at MA + 2 by the immediate operand, which in our example is 2000_{16}. This is the new workspace pointer address.

The result of executing the instruction is illustrated in Fig. 3.11(c). Here we see that the old value of WP, which was $1FD0_{16}$, is replaced by the value 2000_{16}. In this way, a new workspace has been defined in memory. It starts with register R_0 at address 2000_{16} and continues up through R_{15} at address $201E_{16}$. The value of the old workspace pointer is lost.

The other two instructions, LI and LIMI, are used to load workspace registers and the interrupt mask ST_{12} through ST_{15}, respectively. It is important to note that the LI instruction can be used to initialize a workspace register, but cannot be used to load data into general memory.

Example 3.2

What is the result of executing the following instructions?

```
LI    R3,>0200
LIMI  6
```

Solution: Execution of the load immediate instruction causes register R_3 of the current workspace to be loaded with the value 0200_{16}.

$$(R3) = 0200_{16}$$

When the "load interrupt mask immediate" instruction is executed, the value 0110_2 is loaded into the interrupt mask part of the status register. This gives

$$(ST_{12}ST_{13}ST_{14}ST_{15}) = 0110_2$$

The second kind of data-transfer instruction we will consider are those that move data from one memory location to another. These instructions can process either 16-bit words or 8-bit bytes of data. As shown in Fig. 3.12, there are three such instructions: *move word* (MOV), *move byte* (MOVB), and *swap bytes* (SWPB).

The *move words* (MOV) instruction is a good example to demonstrate how these instructions work. Notice that its general description is

```
MOV  S,D
```

Here S and D represent the source and destination operands, respectively.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
MOV	Move words	MOV S,D	(S) → (D)	Y	L >, A >, EQ
MOVB	Move bytes	MOVB S,D	(Sbyte) → (Dbyte)	Y	L >, A >, EQ, OP
SWPB	Swap bytes	SWPB S	$(S_0\text{-}S_7) \leftrightarrow (S_8\text{-}S_{15})$	N	None

Figure 3.12 "Move" data-transfer instruction.

An example of the instruction using direct workspace addressing mode is

```
MOV  R3,R5
```

Looking at Fig. 3.12, we see that its operation can be represented by

$$(R3) \rightarrow (R5)$$

For instance, let us assume that R_3 initially contains $ABCD_{16}$ and that the contents of R_5 are a don't-care state.

$$(R3) = ABCD_{16}$$

$$(R5) = XXXX_{16}$$

The result after executing the instruction is that the value $ABCD_{16}$ is moved to R_5.

$$ABCD_{16} \rightarrow (R5)$$

Thus both R_3 and R_5 now contain the same value.

$$(R3) = ABCD_{16}$$

$$(R5) = ABCD_{16}$$

In Fig. 3.12, we see that the function performed by *move bytes* (MOVB) is similar to that just described for MOV. However, it transfers bytes of data instead of words. For instance, the instruction

```
MOVB  @>1056,@>1055
```

moves the contents of the most significant byte at word memory address 1056_{16} to the least significant byte location at word address 1054_{16}.

The last instruction, *swap bytes* (SWPB), is somewhat different. Note that it works on the same register or memory word and causes the most significant byte to be swapped with the least significant byte.

Example 3.3

Describe what happens when the following sequence of instructions is executed.

```
LI    R2,>3AB5
MOV   R2,R4
MOV   @>ABCD,R3
SWPB  R2
```

Assume that memory address $ABCD_{16}$ initially contains $FF00_{16}$.

Solution: As the "load immediate" instruction is executed, its immediate value is loaded into R_2. This gives

$$3AB5_{16} \rightarrow (R2)$$

Execution of the first move instruction copies the value just loaded into R_2 into register R_4. We get

$$3AB5_{16} \rightarrow (R4)$$

The effect of executing the second move instruction is that the value at address $ABCD_{16}$ in memory is moved to register R_3. We assumed that this memory location was initialized to $FF00_{16}$. Therefore, the results are

$$FF00_{16} \rightarrow (R3)$$

The last instruction swaps the least significant byte of R_2 ($B5_{16}$) with its most significant byte ($3A_{16}$). The result in the register is

$$B53A_{16} \rightarrow (R2)$$

The last group of data-transfer instructions are shown in Fig. 3.13. Here we see that it includes the *store status register instruction* (STST), *store workspace pointer instruction* (STWP), *load status register instruction* (LST), and *load workspace pointer instruction* (LWP). The purpose of these instructions is to permit the contents of ST and WP to be saved in or loaded from a workspace register.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
STST	Store status	STST R	(ST) → (R)	N	None
STWP	Store workspace pointer register	STWP R	(WP) → (R)	N	None
LST	Load status register	LST R	(R) → (ST)	N	None
LWP	Load workspace pointer register	LWP R	(R) → (WP)	N	None

Figure 3.13 "Store and load" data-transfer instructions.

Notice in Fig. 3.13 the general format of the "store workspace pointer" instruction. It can be implemented for a specific register as

```
STWP  R5
```

Executing this instruction causes the contents of WP to be copied into R_5 of the current workspace. For example, if WP contains 1234_{16}, the result after

executing the instruction is that R_5 contains 1234_{16}. Of course, this value also remains in WP. That is,

$$(WP) = (R5) = 1234_{16}$$

Example 3.4

If we assume that ST and WP initially contain 1000_{16} and $ABC0_{16}$, respectively, what is the result of executing the following sequence of instructions?

```
STST  R1
STWP  R3
MOV   R1,R2
MOV   R3,R4
LI    R1,>0000
LI    R3,>ABE0
LWP   R3
LST   R1
```

Solution: Execution of the first two instructions causes the current values in ST and WP to be saved in workspace registers R_1 and R_3, respectively.

$$1000_{16} \rightarrow (R1)$$

$$ABC0_{16} \rightarrow (R3)$$

The next two instructions move these values to R_2 and R_4, respectively.

$$(R1) \rightarrow (R2) = 1000_{16}$$

$$(R3) \rightarrow (R4) = ABC0_{16}$$

They are followed by two "load immediate" instructions that when executed load new values into R_1 and R_3. These values are included in the instructions as immediate operands. The results are

$$0000_{16} \rightarrow (R1)$$

$$ABE0_{16} \rightarrow (R3)$$

Finally, the workspace pointer and status registers are loaded from workspace registers R_3 and R_1, respectively, with LWP and LST instructions. They become

$$(R3) \rightarrow (WP) = ABE0_{16}$$

$$(R1) \rightarrow (ST) = 0000_{16}$$

Notice that we have saved the old ST and old WP in workspace registers and then cleared ST and loaded a new value into the workspace pointer.

3.8 ARITHMETIC INSTRUCTIONS

The 99000 microprocessor also has an extensive group of arithmetic instructions. This group includes instructions for the basic arithmetic functions, such as addition, subtraction, multiplication, and division, as well as special

functions, such as increment, decrement, negate, and absolute value. With these instructions, the programmer can implement more complicated mathematical functions such as floating-point addition and subtraction.

Here again, for study purposes, we will subdivide the larger group of arithmetic instructions into smaller groups. Let us start with the *addition* and *subtraction instructions*. They are shown in Fig. 3.14. Notice that the mnemonic, name, format, function, and status affected are provided for each instruction in this table.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
A	Add words	A S,D	(S) + (D) → (D)	Y	L>, A>, EQ, C, AF
AB	Add bytes	AB S,D	(Sbyte) + (Dbyte) → (Dbyte)	Y	L>, A>, EQ, C, AF, OP
AI	Add immediate	AI R,I	(R) + I → (R)	Y	L>, A>, EQ, C, AF
S	Subtract words	S S,D	(D) − (S) → (D)	Y	L>, A>, EQ, C, AF
SB	Subtract bytes	SB S,D	(Dbyte) − (Sbyte) → (Dbyte)	Y	L>, A>, EQ, C, AF, OP

Figure 3.14 Addition and subtraction instructions.

Let us take a detailed look at the *add words* (A) instruction. Its general format is

```
A  S,D
```

where the source and destination operands can be specified with many of the earlier studied addressing modes. If we use direct workspace addressing mode, a typical example is

```
A  R1,R3
```

The functional description of the add instruction in Fig. 3.14 shows that it adds the value of the source operand to the value of the destination operand and then puts the sum that results in the destination location. For our example, the 99000 would read the contents of both R_1 and R_3 out of memory, add them in its ALU, and then write the result back out into R_3. This operation is described as

$$(R1) + (R3) \rightarrow (R3)$$

If we assume that R_1 contains 1234_{16} and R_3 contains $ABCD_{16}$, the result is found to be

$$1234_{16} + ABCD_{16} \rightarrow (R3) = BE01_{16}$$

Figure 3.14 also indicates that status bits are affected whenever an addition or subtraction instruction is executed. Notice that for the "add words" instruction the result that occurs is compared to zero. Based on this comparison, the logical greater than (L>), arithmetic greater than (A>), and equal (EQ) status bits are affected. Moreover, if a carry out occurs from the MSB during this addition, the carry (C) bit is set and if an overflow error occurs, the arithmetic fault (AF) bit is set.

The other two addition instructions identified in Fig. 3.14 are similar. In the case of the *add bytes* (AB) instruction, a source byte is added to a destination byte and the result is put into the location of the destination byte. On the other hand, the "add immediate" instruction is used to add the value in a workspace register to an immediate operand. The result is put back into the specified workspace register.

The two subtraction instructions *subtract words* (S) and *subtract bytes* (SB) are similar to their corresponding addition instructions. But in this case, the source operand is subtracted from the destination operand and their difference is returned to the location of the destination operand.

Example 3.5

Describe the results found in registers R_2 and R_5, memory location LOC, and the carry status bit after execution of each instruction in the sequence that follows.

```
LI   R2,>0000
LI   R5,>0000
LI   @LOC,>1111
AI   R2,>A234
AI   R5,>1BCD
A    R2,@LOC
AB   R2,R5
S    R2,@LOC
SB   @LOC,R2
```

Assume that R_2, R_5, LOC, and C are initially don't-care states.

Solution: Executing the first "load immediate" instruction clears R_2 to zero as shown in Fig. 3.15. Executing the second LI instruction does the same for R_5. However, the third instruction initializes the contents of LOC to 1111_{16}. The carry bit is not affected by these three operations. This completes initialization of the registers and memory.

The first "add immediate" instruction causes the value $A234_{16}$ to be added to the contents of R_2. Since R_2 contained zero, the result as shown in Fig. 3.15 is that the contents of R_2 becomes equal to $A234_{16}$. No carry occurs; therefore, C is cleared to 0. The same happens as the second AI instruction is executed, but this time the result in R_5 is $1BCD_{16}$.

Next, an "add word" instruction is executed. It causes the contents of R_2, which are $A234_{16}$, to be added to the contents of memory location LOC. LOC was initialized to 1111_{16}; therefore, the result stored in LOC is

$$1111_{16} + A234_{16} = B345_{16}$$

Again, no carry is produced and C equals 0.

This word addition is followed by an "add byte" instruction. Execution of this instruction adds the most significant byte of R_2 to the most significant byte of R_5. That is,

$$A2_{16} + 1B_{16} = BD_{16}$$

In this way, we get $BDCD_{16}$ in R_5. C stays at 0.

Now the value in R_2, which is still $A234_{16}$, is subtracted from the contents of LOC, $B345_{16}$. Their difference is

$$B345_{16} - A234_{16} = 1111_{16}$$

This result is stored in LOC, thereby restoring it to its initial value. No borrow is required; therefore, C is cleared to 0.

Finally, a "byte subtract" instruction is executed. This causes the most significant byte of LOC, which is 11_{16}, to be subtracted from the most significant byte of R_2, which is $A2_{16}$.

$$A2_{16} - 11_{16} = 91_{16}$$

This produces the result 9134_{16} in R_2 and C stays at 0.

Instruction	(R2)	(R5)	(LOC)	(C)
	XXXX	XXXX	XXXX	X
LI R2,>0000	0000	XXXX	XXXX	X
LI R5,>0000	0000	0000	XXXX	X
LI @LOC,>1111	0000	0000	1111	X
AI R2,>A234	A234	0000	1111	0
AI R5,>1BCD	A234	1BCD	1111	0
A R2,@LOC	A234	1BCD	B345	0
AB R2,R5	A234	BDCD	B345	0
S R2,@LOC	A234	BDCD	1111	0
SB @LOC,R2	9134	BDCD	1111	0

Figure 3.15 Example using add and subtract instructions.

The 99000 microprocessor also provides instructions for 32-bit addition and subtraction. These are its *double-precision arithmetic instructions*. Their functions are summarized in Fig. 3.16.

The mnemonic for the *double-precision addition instruction* is AM and its general format is

```
AM  S,D
```

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
AM	Add double	AM S,D	(S, S + 2) + (D, D + 2) → (D, D + 2)	Y	A >, L >, EQ, C
SM	Subtract double	SM S,D	(D, D + 2) − (S, S + 2) → (D, D + 2)	Y	A >, L >, EQ, C

Figure 3.16 Double-precision addition and subtraction instructions.

The functional description of AM is

$$(S,S+2) + (D,D+2) \rightarrow (D,D+2)$$

Here the operands are 32 bits long (*long words*); therefore, they exist at two consecutive word addresses in memory. For this reason, the source operand is identified as residing in S and S+2 and destination operand in D and D+2.

Example 3.6

Describe the result of executing the instruction sequence that follows:

```
LWPI  >1100
LI    R1,>1234
LI    R2,>5678
LI    @>1108,>1110
LI    @>1110,>1111
LI    @>1112,>1010
AM    R1,@>1110
SM    R1,*R4
```

Solution: The first instruction loads the workspace pointer register within the 99000. Its value becomes

$$(WP) = 1100_{16}$$

This establishes a 16-register workspace starting with R_0 at address 1100_{16} and ending with R_{15} at $111E_{16}$.

The five "load immediate" instructions that follow initialize the contents of registers within the workspace. The results of these operations are as follows:

$$(R1) = 1234_{16}$$

$$(R2) = 5678_{16}$$

$$(R4) = (>1108) = 1110_{16}$$

$$(R8) = (>1110) = 1111_{16}$$

$$(R9) = (>1112) = 1010_{16}$$

The AM instruction causes the long-word source operand in registers R_1 and R_2 to be added to the long-word destination operand at addresses >1110 (R_8) and >1112 (R_9). The 32-bit sum that results is placed at the destination addresses. This result is found to be

$$11111010_{16} + 12345678_{16} = 23456688_{16}$$

It is organized in memory as

$$(@>1110) = 2345_{16}$$

and

$$(@>1112) = 6688_{16}$$

The SM instruction employs a different addressing mode, indirect workspace addressing, for its destination operand. However, R_4 was initialized such that the same memory locations are involved as were for the addition operation. This is because R_4 contains destination address >1110. The contents of source registers R_1 and R_2 are subtracted from the new value in >1110 and >1112. This gives

$$23456688_{16} - 12345678_{16} = 11111010_{16}$$

which is organized in memory as

$$(>@1110) = 1111_{16}$$

and

$$(>@1112) = 1010_{16}$$

This restores the contents of the registers to their original values.

Another subgroup of the arithmetic instructions is shown in Fig. 3.17. These are the *increment* and *decrement instructions*. Here we find two different increment instructions, INC and INCT, and two decrement instructions, DEC and DECT. Notice from their functional descriptions that the INC and DEC instructions are used to increment or decrement the specified source operand by 1, respectively. On the other hand, INCT and DECT increment or decrement their operand by 2 instead of 1.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
INC	Increment	INC S	(S) + 1 → (S)	Y	L >, A >, EQ, C, AF
INCT	Increment by two	INCT S	(S) + 2 → (S)	Y	L >, A >, EQ, C, AF
DEC	Decrement	DEC S	(S) − 1 → (S)	Y	L >, A >, EQ, C, AF
DECT	Decrement by two	DECT S	(S) − 2 → (S)	Y	L >, A >, EQ, C, AF

Figure 3.17 Increment and decrement instructions.

Consider the instruction

```
INCT  @LOC
```

It stands for increment the contents of memory location LOC by 2. Assuming that address LOC contains 1234_{16}, execution of the instruction yields

$$(@LOC) + 2 \rightarrow (@LOC)$$

which gives

$$(@LOC) = 1234_{16} + 0002_{16} = 1236_{16}$$

Example 3.7

What is the result of executing the following sequence of instructions?

```
INC   R3
INCT  @>12BC
DEC   @>12BC
DECT  R3
```

Assume that the original contents of R_3 are $AB12_{16}$ and that of address >12BC are 1111_{16}.

Solution: Executing the first instruction causes the contents of R_3 to be incremented by 1. This gives

$$(R3) = AB13_{16}$$

The second instruction causes the value in the memory location at address >12BC to be incremented by 2. Therefore, we get

$$(>12BC) = 1113_{16}$$

The first decrement instruction also affects memory location >12BC. It decreases its value by 1 to give

$$(>12BC) = 1112_{16}$$

Finally, the last instruction causes the value in R_3 to be decremented by 2. The result is

$$(R3) = AB11_{16}$$

Two other instructions in the arithmetic group are the *negate* and *absolute value instructions*. As indicated in Fig. 3.18, the format of the negate instruction is

```
NEG  S
```

Execution of this instruction replaces the source operand with its 2's complement. That is, all of its 0s are converted to 1s, all of its 1s are converted to 0s, and then 1 is added.

On the other hand, the ABS instruction is shown to cause the absolute

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
NEG	Negate	NEG S	2s complement of (S) → (S)	Y	L >, A >, EQ, C, AF
ABS	Absolute value	ABS S	| (S) | → (S)	N	None

Figure 3.18 Negate and absolute value instructions.

value of the source operand to be returned to the source location. Here is an example of the absolute value instruction:

```
ABS  R3
```

This means to replace the contents of R_3 by its absolute value. For instance, if

$$(R3) = 1234_{16} = 0001001000110100_2$$

Since the MSB, which is the sign bit, is logic 0, it represents a positive number. The absolute value is the same binary number. For this reason, the same value is returned to R_3.

As another example, let us assume that the value in R_3 is changed to $A000_{16}$. In binary form this is 1010000000000000_2. In this case, the sign bit is 1 to identify a negative number. Execution of the ABS R3 instruction causes the number in R_3 to be replaced by its absolute value. This gives

$$(R3) = 0010000000000000_2 = 2000_{16}$$

Example 3.8

If R_1 contains 3579_{16} and the storage location at address $ABCD_{16}$ contains 1234_{16}, what is the result of executing the following sequence of instructions?

```
NEG  R1
ABS  @>ABCD
ABS  R1
```

Solution: When the first instruction is executed, the contents of R_1 are replaced by their 2's complement. The original register contents are

$$(R1) = 3579_{16} = 0011010101111001_2$$

Inverting its bits, we get

$$\overline{3579}_{16} = 1100101010000110_2 = CA86_{16}$$

Adding 1_{16} for the 2's complement results in

$$(R1) = CA86_{16} + 1_{16} = CA87_{16}$$

The second instruction replaces the contents of address $ABCD_{16}$ with its absolute value. The original number 1234_{16} is a positive number; therefore, the result after the instruction is the same number.

$$(>ABCD) = 1234_{16}$$

The third instruction takes the absolute value of the contents of R_1. It now contains $CA87_{16}$, which is a negative signed number. This number is replaced by

$$(R1) = 3579_{16}$$

The last set of instructions in the arithmetic group are those for multiplication and division. As shown in Fig. 3.19, there are four of these instructions. They are *unsigned multiply* (MPY), *unsigned divide* (DIV), *signed multiply* (MPYS), and *signed divide* (DIVS).

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
MPY	Unsigned multiply	MPY S,R	Most significant word of (S) * (R) → (R) Least significant word of (S) * (R) → (R + 1)	N	None
DIV	Unsigned divide	DIV S,R	(R, R + 1)/(S) Quotient → (R) Remainder → (R + 1)	N	AF
MPYS	Signed multiply	MPYS S	Most significant word of signed product (S) * (R0) → (R0) Least significant word of signed product (S) * (R0) → (R1)	Y	L >, A >, EQ
DIVS	Signed divide	DIVS S	(R0, R1)/(S) Quotient → (R0) Remainder → (R1)	Y	L >, A >, EQ, AF

Figure 3.19 Multiplication and division instructions.

In the case of *unsigned multiply* (MPY), the contents of source operand S and destination workspace register R are treated as unsigned numbers. Notice that the destination must always be a workspace register. The 32-bit product that results from the multiplication is always placed in destination workspace register R and the next consecutive register R+1.

For the *signed divide* instruction (DIVS), the MSB of the source operand and destination operand are considered as sign bits. The source operand is the 16-bit divisor and it can be located anywhere in memory. On the other hand, the dividend, which is 32 bits, must be held in workspace

registers R_0 and R_1. The quotient that results is produced in R_0 with the remainder in R_1.

Example 3.9

What is the result produced by the following sequence of instructions?

```
MPY  @NUM1,R2
DIV  @NUM2,R2
```

Assume that the original contents of R_2, NUM1, and NUM2 are $C111_{16}$, 0003_{16}, and 0004_{16}, respectively.

Solution: Executing the first instruction results in the word stored at address NUM1 being multiplied by the word in R_2. The unsigned result that is produced in R_2 and R_3 is

$$(@NUM1) \times (R2) = 0003_{16} \times C111_{16} = 00024333_{16}$$

$$(R2) = 0002_{16}$$

$$(R3) = 4333_{16}$$

The division instruction causes the 32-bit product in R_2 and R_3 to be divided by 0004_{16}, which is at address NUM 2. This yields

$$R2,R3/NUM2 = 00024333_{16}/0004_{16}$$

$$(R2) = 90CC_{16}$$

$$(R3) = 0003_{16}$$

Example 3.10

Write a program to compute the value of E using the following equation:

$$E = |AB - C/D|$$

Assume that A, B, and D are 8-bit signed numbers whose values are to be assigned as immediate operands. On the other hand, C is a 32-bit signed number already held in R_7 and R_8. Also assume that the workspace pointer has already been loaded with 0400_{16}.

Solution: Let us start by putting A and B into registers R_2 and R_3, respectively. At this time, for convenience, we will also load D into R_4. This can be done with the following "load immediate" instructions:

```
LI  R2,A
LI  R3,B
LI  R4,D
```

Now we will multiply A and B. To use the MPYS instruction, one operand must be in R_0. Let us move A from R_2 to R_0. This is done with

```
MOV  R2,R0
```

Now the multiplication is performed by the instruction

```
MPYS  R3
```

Let us save just the least significant word of the result in R_5.

```
MOV  R1,R5
```

To implement the signed division of C by D, we must move the value of C from R_7 and R_8 to R_0 and R_1, respectively. Two move instructions are required.

```
MOV  R7,R0
MOV  R8,R1
```

Now we can divide.

```
DIVS  R4
```

Dropping the remainder, we will just move R_0 to R_6.

```
MOV  R0,R6
```

Finally, we can subtract C/D in R_6 from AB in R_5 and then take the absolute value. This is done with the instructions

```
S    R6,R5
ABS  R5
```

The final result, E, resides in R_5. The complete program is shown in Fig. 3.20.

```
LI    R2,A
LI    R3,B
LI    R4,D
MOV   R2, R0
MPYS  R3
MOV   R1, R5
MOV   R7, R0
MOV   R8, R1
DIVS  R4
MOV   R0, R6
S     R6, R5
ABS   R5
```

Figure 3.20 Program for evaluating the equation $F = |AB - C/D|$.

3.9 LOGIC INSTRUCTIONS

The logic instructions of the 99000's instruction set allow logic operations to be performed in software. The logic instructions can be divided into three groups. The first group provides the basic *Boolean logic functions*, such as AND, OR, and NOT. The second type are those instructions that can be used to clear or set all of the bits in a register or memory location. The last type permits a programmer to clear or set individual bits of a word or byte in a register or memory location.

Let us begin by looking more closely at the first type, the basic logic instructions. These four instructions, *AND immediate* (ANDI), *OR immediate* (ORI), *exclusive-OR* (XOR), and *invert* (INV), are described in Fig. 3.21. All of these instructions always perform their operations on a word of data.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
ANDI	AND immediate	ANDI R,I	(R) AND I → (R)	Y	L >, A >, EQ
ORI	OR immediate	ORI R,I	(R) OR I → (R)	Y	L >, A >, EQ
XOR	Exclusive-OR	XOR S,R	(S) XOR (R) → (R)	Y	L >, A >, EQ
INV	Invert	INV S	$(\overline{S})$ → (S)	Y	L >, A >, EQ

Figure 3.21 Logic instructions.

In the case of the ANDI and ORI instructions, we see in Fig. 3.21 that the bits in the specified workspace register are logically ANDed or ORed, respectively, with their corresponding bits in the immediate operand. This immediate operand is frequently referred to as the *mask*. The result produced by ANDing or ORing the data word with the mask is put in the assigned workspace register.

The result of an AND operation tells which bits are logic 1 in both the mask and workspace register. On the other hand, the result of an OR operation identifies which bits are logic 0 in both the mask and workspace register.

The XOR instruction causes the contents of the source operand to be exclusive-ORed with the value in the specified workspace register. The result is returned to the workspace register. It identifies which bits are of opposite logic levels in the source operand and workspace register.

The INV instruction simply inverts the logic level of each bit in the source operand.

Example 3.11

Describe what happens to the contents of R_2 and R_3 as the following instructions are executed.

```
LI    R2,>ABCD
ANDI  R2,>F017
ORI   R2,>100A
LI    R3,>AEB9
XOR   R2,R3
INV   R3
```

Solution: As these instructions are executed, the contents of R_2 and R_3 are affected as shown in Fig. 3.22. The first instruction causes the value $ABCD_{16}$ to be loaded into R_2. Then this value is ANDed with the immediate operand $F017_{16}$. Therefore, we get

$$(R2) = ABCD_{16} = 1010101111001101_2$$

$$F017_{16} = 1111000000010111_2$$

$$\text{NEW (R2)} = ABCD_{16} \cdot F017_{16}$$

$$= 1010101111001101_2 \cdot 1111000000010111_2$$

$$= 1010000000000101_2$$

$$= A005_{16}$$

Next, the new contents of R_2 are ORed with $100A_{16}$. This gives

$$100A_{16} = 0001000000001010_2$$

$$\text{NEW (R2)} = A005_{16} + 100A_{16}$$

$$= 1010000000000101_2 + 0001000000001010_2$$

$$= 1011000000001111_2$$

$$= B00F_{16}$$

Now the value $AEB9_{16}$ is loaded into R_3 and then its value is exclusive-ORed with $B00F_{16}$. The results are

$$(R3) = AEB9_{16} = 1010111010111001_2$$

$$\text{NEW (R3)} = (R2) \oplus (R3)$$

$$= 1011000000001111_2 \oplus 1010111010111001_2$$

$$= 0001111010110110_2$$

$$= 1EB6_{16}$$

The last instruction inverts each of the bits in R_3.

$$\text{NEW (R3)} = (\overline{R3}) = 1110000101001001_2$$

$$= E149_{16}$$

Instruction		(R2)	(R3)
LI	R2, > ABCD	ABCD	XXXX
ANDI	R2, > F017	A005	XXXX
ORI	R2, > 100A	B00F	XXXX
LI	R3, > AEB9	B00F	AEB9
XOR	R2, R3	B00F	1EB6
INV	R3	B00F	E149

Figure 3.22 Example using logic instructions.

The next two instructions in the logic group are *clear* (CLR) and *set to ones* (SETO). These instructions are handy when the value of a register or memory location is to be cleared to zero or loaded with all 1s. Notice in Fig. 3.23 that CLR causes the value 0000_{16} to be loaded into the source location and that SETO causes $FFFF_{16}$ to be loaded.

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
CLR	Clear	CLR S	>0000 → (S)	N	None
SETO	Set to ones	SETO S	>FFFF → (S)	N	None

Figure 3.23 "Clear" and "set to ones" instructions.

The last group of logic instructions are those that manipulate bits in either a register or memory location. The first two of these instructions, as shown in Fig. 3.24, are *set ones corresponding* (SOC) and *set ones corresponding byte* (SOCB). Notice that the SOC instruction has the ability to

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
SOC	Set ones corresponding	SOC S,D	(S) OR (D) → (D)	Y	L >, A >, EQ
SOCB	Set ones corresponding bytes	SOCB S,D	(Sbyte) OR (Dbyte) → (Dbyte)	Y	L >, A >, EQ, OP
SZC	Set zeros corresponding	SZC S,D	Bits in S that are 1 have their corresponding bits in D set to 0	Y	L >, A >, EQ
SZCB	Set zeros corresponding bytes	SZCB S,D	Bits in Sbyte that are 1 have their corresponding bits in Dbyte set to 0	Y	L >, A >, EQ, OP

Figure 3.24 "Set ones corresponding" and "set zeros corresponding" instructions.

directly OR the contents of a source operand with that of a destination operand instead of an immediate operand such as the ORI instruction. In the result, those bits of the destination that correspond to bits in the source that are logic 1 are switched to the 1 logic level. The SOCB instruction works in the same way except that it operates on the most significant byte or least significant byte of the data word.

Notice that the other two instructions in the group, *set zeros corresponding* (SZC) and *set zeros corresponding byte* (SZCB), work in the opposite way. They cause those bit locations of the destination operand that are logic 1 in the source operand to be set to logic 0.

Example 3.12

Describe the results of executing the following sequence of instructions.

```
LI   R3,>8005
LI   R2,>7FFF
INV  R2
SZC  R2,R3
```

Solution: The first instruction loads R_3 with its immediate operand and the next instruction loads R_2.

$$(R3) = 8005_{16} = 1000000000000101_2$$

$$(R2) = 7FFF_{16} = 0111111111111111_2$$

Now the bits in R_2 are complemented, giving

$$(R2) = (\overline{R2}) = 1000000000000000_2$$

Finally, the SZC instruction sets the MSB of R_3 to 0.

$$\text{OLD (R3)} = 1000000000000101_2 \qquad \text{negative number}$$

$$\text{NEW (R3)} = 0000000000000101_2 \qquad \text{positive number}$$

Example 3.13

Write a program that will disassemble the four hexadecimal digits in memory location LOC. That is, starting from the least significant digit (LSD) in LOC and ending with its most significant digit (MSD), they must end up in memory locations identified by HEX0, HEX1, HEX2, and HEX3, respectively. Assume that the value in LOC is $ABCD_{16}$.

Solution: The ANDI instruction can be used to mask off the hex digits of LOC. However, to use it, the destination must be a workspace register. Therefore, let us set up a workspace starting at >0400 by loading this value into the workspace pointer with the instruction

```
LWPI  >0400
```


We will use workspace register R_2 to store temporary results. Therefore, LOC must first be moved there and then its LSD masked off with the ANDI instruction. To do this, we execute the following instructions:

```
MOV  @LOC,R2
ANDI R2,>000F
```

Executing these instructions causes the contents of R_2 to become equal to $000D_{16}$. This value must now be moved to memory location HEX0. This is done with the instruction

```
MOV  R2,@HEX0
```

The next digit in LOC can be masked off and stored at HEX1 in a similar way.

```
MOV  @LOC,R2
ANDI R2,>00F0
MOV  R2,@HEX1
```

The other two digits are masked off and moved to HEX2 and HEX3, respectively, using the same instruction sequence. In these two cases, the masks used in the ANDI instruction are $0F00_{16}$ and $F000_{16}$, respectively. The complete program is listed in Fig. 3.25.

```
LWPI   >0400
MOV    @LOC, R2
ANDI   R2, >000F
MOV    R2, @HEX0
MOV    @LOC, R2
ANDI   R2, >00F0
MOV    R2, @HEX1
MOV    @LOC, R2
ANDI   R2, >0F00
MOV    R2, @HEX2
MOV    @LOC, R2
ANDI   R2, >F000
MOV    R2, @HEX3
```

Figure 3.25 Program for disassembling packed hexadecimal digits.

3.10 SHIFT INSTRUCTIONS

The shift instructions are provided in the instruction set of the 99000 so that the bits of data in a workspace register can be shifted to the right or to the left. The four shift instructions, as shown in Fig. 3.26, are *shift right arith-*

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
SRA	Shift right arithmetic	SRA R,I	Shift (R) right by I bit positions, fill the vacated MSB positions with the original content of bit 0 of R	Y	L >, A >, EQ, C
SLA	Shift left arithmetic	SLA R,I	Shift (R) left by I bit positions, fill the vacated LSB positions with 0	Y	L >, A >, EQ, C, AF
SRL	Shift right logical	SRL R,I	Shift (R) right by I bit positions, fill vacated MSB positions with zeros	Y	L >, A >, EQ, C
SRC	Shift right circular	SRC R,I	Shift (R) right by I bit positions, fill the vacated MSB positions by those shifted out at the LSB	Y	L >, A >, EQ, C

Figure 3.26 Shift instructions.

metic (SRA), *shift left arithmetic* (SLA), *shift right logical* (SRL), and *shift right circular* (SRC).

Notice in Fig. 3.26 that the SRA instruction has the format

```
SRA  R,I
```

Here R identifies the workspace register whose contents are to be shifted and I indicates by how many bit positions. The value of I can be any number from 0 through 15. Each time a bit shifts to the right, the MSB location that is vacated is filled with the original logic level of the MSB. Moreover, the last bit shifted out of the LSB location is stored in the carry bit of the status register.

An example is the instruction

```
SRA  R3,2
```

Assume that R_3 originally contains 1234_{16}.

$$\text{ORIGINAL (R3)} = 1234_{16} = 0001001000110100_2$$

Execution of the instruction causes all bits in R_3 to be shifted two bit positions to the right. Since the MSB of the original number is logic 0, the two vacated MSBs are filled with 0s. This maintains the sign of the number. Moreover, since the last bit shifted out was logic 0, carry bit C is also logic 0. This gives

$$\text{NEW (R3)} = 0000010010001101_2 \quad (C) = 0$$

Looking at Fig. 3.26, we also find descriptions of the other three instructions. Notice that SLA R,I causes bits of data in R to be shifted I bits to the left. This time the vacated LSB locations are filled with zeros and the last bit shifted out from the MSB location ends up in carry.

As shown in Fig. 3.26, SRL works the opposite of SLA. In this case, the bits are shifted to the right and the vacated MSBs are filled with 0s. The last bit shifted out at the LSB location is found in C.

The shift right circular instruction, as indicated in Fig. 3.26, also causes the contents of the register to be shifted right by I bits. But this time the bits shifted out from the LSB position are reloaded at the MSB position. Carry ends up holding the logic level of the last bit shifted out at the right.

There is one restriction to the use of workspace registers with the shift instructions. This is that R can equal R_1 through R_{15} but not R_0. The reason for this is that when I is specified as 0, the *shift count* is specified by the four LSBs of R_0. This count must be loaded prior to the execution of the shift instruction.

Here is an example of how R_0 is used to specify a shift count.

```
LI   R0,>0002
SRA  R3,0
```

The result of executing these two instructions is identical to that obtained by executing the instruction SRA R3,2.

One use of shift instructions is to multiply or divide the contents of a register by powers of 2.

Example 3.14

If the contents of R_3 are 1234_{16}, what is the effect of executing the following shift instructions?

```
SLA  R3,2
SRL  R3,3
SRC  R3,2
```

Solution: The original register R_3 contents are

$$\text{(R3)} = 0001001000110100_2$$

The first instruction shifts the contents of R_3 left by two bit positions. Logic 0 is loaded into the two LSBs and 0 into carry. That is,

$$\text{(R3)} = 0100100011010000_2 \quad (C) = 0$$

This operation is illustrated in Fig. 3.27(a).

Next the contents of R_3 are shifted right 3 bits. This time three 0s are loaded at the MSB location and the last bit shifted out of the LSB, which is also 0, ends up in C.

$$(R3) = 0000100100011010_2 \quad (C) = 0$$

This result is shown in Fig. 3.27(b).

Finally, the shift right circular instruction is executed. This causes the two LSBs of R_3 to be shifted out and reloaded at the MSB end. The carry status ends up equal to the value of the last bit shifted out and equals logic 1.

$$(R3) = 1000001001000110_2 \quad (C) = 1$$

This result is illustrated in Fig. 3.27(c).

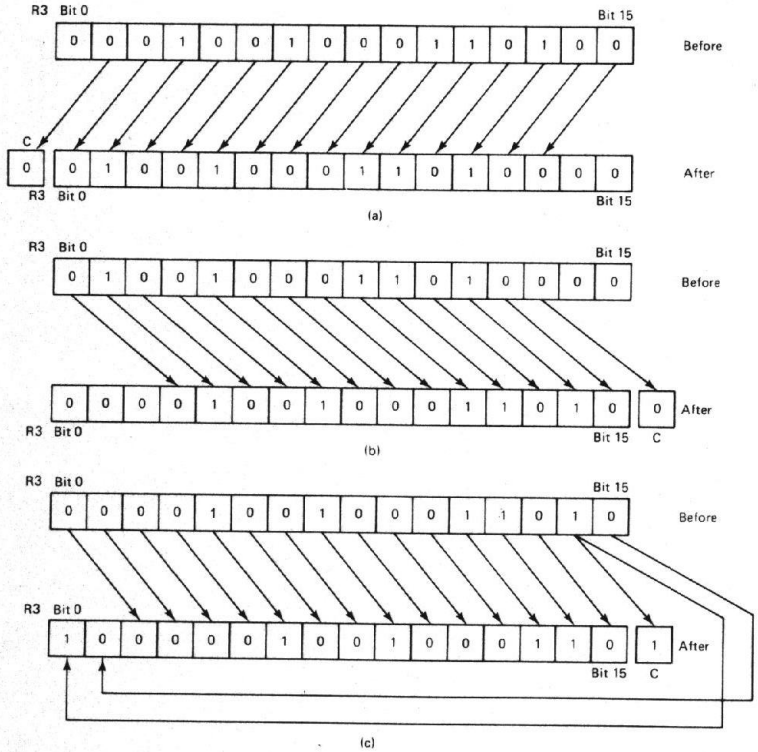


Figure 3.27 (a) Results from execution of SLA R3,2; (b) results from execution of SRL R3,3; (c) results from execution of SRC R3,2.

Example 3.15

Implement the following expression using primarily shift instructions to perform the arithmetic.

$$3(R_5) + 7(R_6) \rightarrow (R_7)$$

Solution: Shifting left by 1 bit gives a multiplication by 2. However, to perform multiplication by an odd number, we can use a shift instruction to multiply to the nearest multiple of 2 and then add or subtract the appropriate value to get the desired results.

The algorithm for performing the expression starts by shifting (R_5) left by 1 bit. This gives 2 times (R_5). Adding the original (R_5) gives multiplication by 3. Next, the contents of R_6 are shifted left by 3 bits to give 8 times its value. Subtracting the original (R_6) once gives multiplication by 7. Expressing this with instructions, we get

```
MOV  R5,R4
SLA  R4,1
A    R5,R4
MOV  R6,R7
SLA  R7,3
S    R6,R7
A    R4,R7
```

It is important to note that we have assumed that 3 times (R_5) and 8 times (R_6) do not yield numbers with more than 16 bits. In other words, an overflow does not occur.

The 99000 also provides instructions that implement 32-bit shift operations in addition to the 16-bit shift instructions we just described. These are the double-precision shift instructions: *shift right arithmetic double* (SRAM) and *shift left arithmetic double* (SLAM). Brief descriptions of the functions of these instructions are provided in Fig. 3.28. Notice that these functions

Instruction	Meaning	Format	Explanation	Result compared to zero	Status bits affected
SRAM	Shift right arithmetic double	SRAM S,I	Shift (S, S + 2) right by I bit positions, fill vacated MSB positions in S with the original value of its MSB, bits shifted out at the LSB of S are shifted in at the MSB of S + 2	Y	L >, A >, EQ, C
SLAM	Shift left arithmetic double	SLAM S,I	Shift (S, S + 2) left by I bit positions, fill vacated LSB positions with 0 and bits shifted out at the MSB of S + 2 are shifted in at the LSB of S	Y	L >, A >, EQ, C, AF

Figure 3.28 "Shift double" instructions.

are identical to the SRA and SLA instructions, respectively, other than that they perform their shift operation on the 32-bit number in S and S+2.

Example 3.16

Describe the result of executing the following instruction sequence:

```
LI   R5,>1234
LI   R6,>5678
SLAM R5,2
```

Solution: The first two instructions load R_5 and R_6, respectively. This gives the 32-bit value

$$(R5,R6) = 12345678_{16}$$
$$= 00010010001101000101011001111000_2$$

The SLAM instruction shifts these 32 bits two positions to the left. The LSBs are filled with 0s and the last bit shifted out resides in carry. The result is

$$(R5,R6) = 01001000110100010101100111100000_2 \quad (C) = 0$$

Notice that the contents of the registers have changed:

$$(R5) = 0100100011010001_2 = 48D1_{16}$$

and

$$(R6) = 0101100111100000_2 = 59E0_{16}$$

ASSIGNMENT

Section 3.2

1. Can the 99000 directly store a word of data starting at an odd address? Explain your answer.
2. Show how memory addresses $B000_{16}$ to $B003_{16}$ are used to store the 32-bit number 76543210_{16}.
3. If the address $B000_{16}$ corresponds to workspace register R_{12}, what are the contents of workspace register WP?
4. Identify the three parts of an assembly language instruction in each of the following statements:

```
(a) AGAIN   A     R0,R3   ADD THE REGISTERS
(b)         MOV   R3,R7   SAVE RESULTS
```

5. Identify the source and destination operands for each statement in problem 4.

Section 3.4

6. Describe the difference between each of the following transfer notations:

$$(R1) \rightarrow @LOC$$
$$(*R1) \rightarrow @LOC$$
$$(*R1+) \rightarrow @LOC$$

7. What is the purpose of addressing modes? Why are there so many different kinds of addressing?
8. Identify the addressing modes that are used for the source and destination operands in the instructions that follow:

```
(a) MOV  >1234,@>1234      (d) MOV  R1,*R2+
(b) MOV  @>1234,R2         (e) MOV  R1,@>1234(R2)
(c) MOV  R1,*R2            (f) MOV  $-10
```

9. If the contents of WP, R_1, R_2, and PC are >0800, >1000, >2000, and >0100, respectively, compute the memory address of the source and destination operands (if any) in each of the statements in problem 8.

Section 3.7

10. Determine the results of executing each of the instructions in problem 8 and their effect on status. Assume that at the start of each instruction, the contents of WP, R_1, R_2, and PC are >0800, >1000, >2000, and >0100, respectively.
11. Write an instruction sequence which stores the current contents of WP and ST in R_0 and R_1, respectively, and then reloads them from R_2 and R_3, respectively.

Section 3.8

12. Are the following instruction sequences equivalent? Explain.

Sequence A

```
LWPI  >1000
A     R0,R2
A     R1,R3
```

Sequence B

```
LWPI  >1000
AM    R0,R2
```

13. Consider the following two instruction sequences:

Sequence A

```
LWPI  >1000
LI    R0,>A000
LI    R1,>B000
MOV   *R0+,*R1+
```

Sequence B

```
LWPI  >1000
LI    R0,>A000
LI    R1,>B000
MOV   *R0,*R1
INC   R0
INC   R1
MOV   *R0,*R1
```

Are they equivalent? If not, modify B such that it provides an alternative way of performing the same operation as A.

14. Given two 16-bit signed numbers in locations >A000 and >B000, write an instruction sequence that generates the following:
    (a) The product of the two numbers stored at address >C000.
    (b) The quotient of the number at >A000 divided by the number at >B000 stored at >C004.
    (c) The remainder of the division in part (b) stored at >C006.

Section 3.9

15. Use only logic instructions to write an equivalent of the instruction

```
MOV  R5,@>ABCD
```

Section 3.10

16. Write an instruction sequence that stores the word in register R_0 at odd address >A001.
17. Implement the following arithmetic function:

$$F = 9(R_2) - 7(R_3) + \frac{1}{8}(R_4)$$

Save the value of F at address >A000 in memory. Specify all assumptions made when writing this segment of program.

4

99000 MICROPROCESSOR PROGRAMMING II

4.1 INTRODUCTION

In Chapter 3 we discussed part of the instruction set of the 99000 microprocessor. Using those instructions, we also covered some preliminary programming techniques. Here we continue our study of the instruction set and introduce some more sophisticated programming techniques. Specifically, the following topics are presented in this chapter:

1. Compare instructions
2. Jump instructions
3. Example programs employing loops
4. Subroutines and subroutine handling instructions

4.2 COMPARE INSTRUCTIONS

The instruction set of the 99000 provides instructions to compare the contents of words in memory, bytes in memory, and workspace registers with a mask. The compare instructions are shown in Fig. 4.1. Notice that there are five instructions: *compare immediate* (CI), *compare words* (C), *compare bytes* (CB), *compare ones corresponding* (COC), and *compare zeros corresponding* (CZC). Depending on which instruction is used, the mask can be specified as an immediate operand, the contents of a register, or even a value in general data memory. The results of the comparison are indicated by setting or clearing appropriate bits in the status register.

Instruction	Meaning	Format	Explanation	Results compared to zero	Status bits affected
CI	Compare immediate	CI R,I	I is compared to (R) and the appropriate status bits are set or reset	Y	L>, A>, EQ
C	Compare words	C S,D	(S) are compared to the (D) and the appropriate status bits are set or reset	Y	L>, A>, EQ
CB	Compare bytes	CB S,D	(Sbyte) are compared to the (Dbyte) and the appropriate status bits are set or reset	Y	L>, A>, EQ, OP
COC	Compare ones corresponding	COC S,R	(R) are compared with the (S) and the EQ bit is set if R has 1 in every bit that corresponds to a bit in S that is 1; otherwise, EQ is reset	N	EQ
CZC	Compare zeros corresponding	CZC S,R	(R) are compared with the (S) and the EQ bit is set if R has 0 in every bit that corresponds to a bit in S that is 1; otherwise, EQ is reset	N	EQ

Figure 4.1 Compare instructions.

The simplest of the compare instructions is CI. As indicated in Fig. 4.1, it compares the contents of the specified workspace register with an immediate operand. The immediate operand is coded as the second word of the instruction. Notice that execution of this instruction affects just three status bits: logical greater than (L>), arithmetic greater than (A>), and equal (EQ).

The next two instructions in Fig. 4.1, C and CB, are similar to each other in that they both compare the value of a source operand to that of a destination operand. However, C compares all 16 bits of the operands, while CB compares just their upper or lower bytes. Again the L>, A>, and EQ status bits are set or reset to represent the results of the comparison. However, notice that CB also affects the odd parity (OP) bit. If the operands of a CB instruction are specified as workspace registers, their most significant bytes are compared and their least significant bytes remain unaffected.

For instance, let us consider the following instruction sequence:

```
LI   R2,>AB34
LI   R3,>1234
CB   R2,R3
```


Here the most significant byte of R_2, which is AB_{16}, is compared to the most significant byte of R_3, which is 12_{16}. From a logic point of view, AB_{16} is greater than 12_{16}. Therefore, status bit L> is set. From an arithmetic point of view, AB_{16} is a negative number, while 12_{16} is a positive number. In this case AB_{16} is less than 12_{16}. For this reason, A> is cleared. Moreover, the two bytes are not equal, so the EQ bit is also cleared. The OP status bit is set based on the number of bits that are logic 1 in the source operand. For our example, the most significant byte of the source operand has an odd number of 1s; therefore, OP is set.

The COC and CZC instructions are used to compare a source operand with the contents of a workspace register. For COC, if all bits that are 1 in the source operand are also 1 in the workspace register, the EQ status bit is set. Moreover, CZC tests only those bits in the destination workspace register that correspond to bits that are 1 in the source operand; however, EQ is set only if these bits are all logic 0.

Example 4.1

Describe what happens to the bits in the status register as the following sequence of instructions is executed.

```
LWPI   >0400
LI     R1,>0000
LST    R1
LI     R2,>ABCD
LI     R3,>1234
C      R2,R3
CB     R3,R2
CI     R2,>ABCD
MOV    R2,@LOC
COC    @LOC,R3
CLR    R2
CZC    R3,R2
```

Solution: What happens in the status register as the instructions are executed is summarized in Fig. 4.2. Arrows have been used to identify which bits are affected by each instruction. Here we see that the first instruction loads WP with 0400_{16}. This sets up a workspace starting at address 0400_{16}. Status is not affected due to the execution of this instruction. Moreover, the status bits are initially identified as don't-care states.

The next instruction loads R_1 with 0000_{16}. During the execution of this instruction, the resulting contents of the register are compared to zero and the L>, A>, and EQ bits are adjusted appropriately. 0000_{16} is not logically greater than zero; therefore, L> is reset to 0. Furthermore, it is not arithmetically greater than zero and A> is also cleared. However, the result is equal to zero, so EQ is set to 1.

The LST R1 instruction loads the status register from R_1. R_1 contains 0000_{16}; therefore, all bits of the status register are cleared.

Instruction	Function	Status register
LWPI >400	Establish workspace starting at address >0400	xxxxxxxxxxxxxxxx
LI R1, >0000	Load R1 with >0000	↓↓↓ 001xxxxxxxxxxxxx
LST R1	Load ST with (R1)	↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓ 0000000000000000 = >0000
LI R2, >ABCD	Load R2 with >ABCD	↓↓↓ 1000000000000000 = >8000
LI R3, >1234	Load R3 with >1234	↓↓↓ 1100000000000000 = >C000
C R2, R3	Compare the (R2) with the (R3)	↓↓↓ 1000000000000000 = >8000
CB R3, R2	Compare the MSbyte of (R3) with the LSbyte of (R2)	↓↓↓ ↓ 0100000000000000 = >4000
CI R2, >ABCD	Compare the (R2) with >ABCD	↓↓↓ 0010000000000000 = >2000
MOV R2, @LOC	Copy the (R2) into @LOC	↓↓↓ 1000000000000000 = >8000
COC @LOC, R3	Compare the (R3) with the (@LOC) for the corresponding ones	↓ 1000000000000000 = >8000
CLR R2	Load R2 with zeros	1000000000000000 = >8000
CZC R3, R2	Compare the (R3) with the (R2) for the corresponding zeros	↓ 1010000000000000 = >A000

Figure 4.2 Example program employing compare instructions.

The two LI instructions that follow load R_2 and R_3 with the values $ABCD_{16}$ and 1234_{16}, respectively. As each of these instructions is executed, the resulting register value is compared to zero and bits ST_0 through ST_2 are adjusted. Notice that loading $ABCD_{16}$ into R_2 causes L> to be set, A> to be cleared, and EQ to be cleared. On the other hand, loading 1234_{16} into R_3 causes both L> and A> to be set and EQ to be cleared.

The next instruction is from the compare group. It compares the word in R_2 to that in R_3. Since $ABCD_{16}$ is logically greater than 1234_{16} but arithmetically less than it, L> is set, A> is cleared, and EQ is cleared.

The next instruction is a "compare byte" instruction. It compares the most significant byte of R_3 (12_{16}) to the most significant byte of R_2 (AB_{16}). In this case, the L> bit is cleared, A> bit is set, and EQ is cleared. This instruction also affects the odd-parity (OP) bit. In this case, the source byte is 12_{16}, which has 2 bits at the 1 level; therefore, OP is cleared.

The contents of R_2 are now compared to the immediate operand $ABCD_{16}$. R_2 contains the same number. Since they are both equal, EQ is set, and both L> and A> are cleared.

Next, the contents of R_2 are moved to address LOC. Therefore, the new data at LOC equal $ABCD_{16}$. This affects status in this way: L> = 1, A> = 0, and EQ = 0.

Now the COC instruction tests the bits in R_3 which correspond to bits that are 1 in LOC to determine if they are all 1. This comparison is as follows:

$$(LOC) = ABCD_{16} = 1010101111001101_{16}$$

$$(R3) = 1234_{16} = 0001001000110100_{16}$$

In this way, we find that not all of these bits in R_3 are logic 1. Therefore, EQ is cleared.

Execution of the next instruction clears R_2 and the status remains unchanged. Finally, the CZC instruction is executed. It tests the contents of R_3 to determine if all bits that are logic 1 in R_3 are logic 0 in R_2. Since R_2 was just cleared, all of its bits are 0 and the comparison condition is satisfied. Therefore, EQ is set.

4.3 JUMP INSTRUCTIONS

Earlier we found that the contents of the program counter identify the address of the next instruction to be executed. This is because after an instruction is fetched from program storage memory and before its execution is completed, the value in PC is incremented such that it points to the next sequential word.

The programs we introduced in Chapter 3 were all examples of *straight-line programs*. That is, one instruction after the other is fetched and executed. For this reason, during their execution, PC increments sequentially until each instruction in the routine has been fetched and executed.

However, most practical programs require that some parts of the program be executed only if certain conditions have been met. It is for this purpose that the jump group of instructions are included in the 99000's instruction set.

The 99000 microprocessor has two types of jump instructions: the unconditional and conditional jump instructions. An *unconditional jump instruction* always initiates a change in PC. This concept is illustrated in Fig. 4.3(a). Notice that when the instruction JMP AA is executed in part I, program control is passed to a point in part III identified by label AA. Execution resumes with the instruction corresponding to AA. The locations in part II of program memory have been bypassed; that is, they have been jumped over.

On the other hand, a *conditional jump instruction* performs the jump (change in PC contents) only if the condition or conditions specified in the instruction are met. When a conditional jump instruction is fetched, a test is made prior to its execution to identify if the specified condition or conditions exist. If they exist, the contents of PC are modified such that the next

instruction is fetched from a new location in program memory; otherwise, execution continues with the next sequential instruction.

Typically, the test conditions represent the present logic level of one or more bits in the status register. For example, a jump may be initiated only if the EQ bit is set.

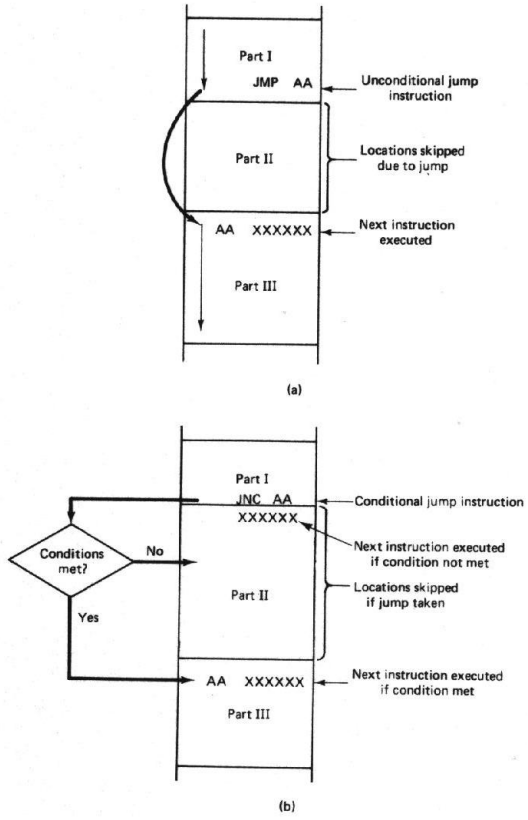


Figure 4.3(a) Unconditional jump program sequence; (b) conditional jump program sequence.

Looking at Fig. 4.3(b), we see that execution of the conditional jump instruction in part I causes a test to be initiated. If the conditions are not met, the *NO* path is taken and execution continues with the next sequential instruction. This corresponds to the first instruction in part II. However, if the result of the conditional test is *YES*, a jump is initiated to the segment of program identified as part III.

The jump instructions of the 99000 are shown in Fig. 4.4. Notice that there is just one instruction used for unconditional jumps and its mnemonic is JMP. The place to which the jump is to occur is specified as follows:

```
JMP  LABEL
```

This says to jump to the statement with the tag LABEL.

Instruction	Meaning	Status conditions for jump
JMP	Jump unconditional	None
JEQ	Jump on equal	EQ = 1
JNE	Jump on not equal	EQ = 0
JGT	Jump on greater than (arithmetic)	A > = 1
JLT	Jump on less than (arithmetic)	A > = 0 AND EQ = 0
JH	Jump on high (logical)	L > = 1 AND EQ = 0
JHE	Jump on high or equal (logical)	L > = 1 OR EQ = 1
JL	Jump on low (logical)	L > = 0 AND EQ = 0
JLE	Jump on low or equal (logical)	L > = 0 OR EQ = 1
JOC	Jump on carry	C = 1
JNC	Jump on no carry	C = 0
JOP	Jump on odd parity	OP = 1
JNO	Jump on no overflow	AF = 0

Figure 4.4 Jump instructions.

Figure 4.5 illustrates this example. Notice that the jump instruction at address 0400_{16} is coded as 1003_{16}. Here the 3 in the LSD location is the displacement of the jump to address with respect to the address of the JMP instruction. When this instruction is executed, the displacement is multiplied by 2 and added to the updated value in PC. This gives 0408_{16}. Therefore, a jump takes place to the address corresponding to the tag LABEL and execution continues with the fetch of the instruction located at address 0408_{16}.

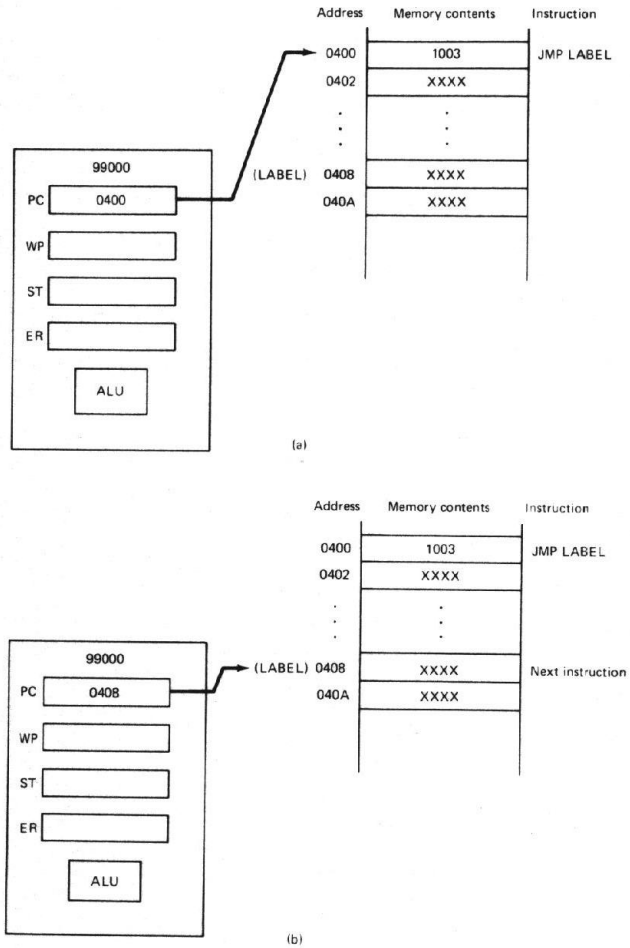


Figure 4.5 (a) Execution of the JMP LABEL instruction; (b) result after execution.

In this way, we see that the instructions from address 0402_{16} through 0406_{16} are not executed.

Our example instruction could be written in several different ways and still get coded and executed the same. For instance, if it were written as

```
JMP  >408
```

execution would continue at the same point. Another way that is frequently used is to specify the number of bytes that are to be jumped over. Using this approach, the instruction is written as

```
JMP  $+8
```

The $ sign indicates that the signed number that follows represents the number of bytes to the jump to address with respect to the current value in PC.

The *displacement* is coded into the eight least significant bits of the JMP instruction word. With this 8-bit signed displacement, the maximum range of jump permitted is limited to −127 to +128 words from the JMP instruction.

The other 12 instructions in the jump group fall into the conditional jump category. Looking at Fig. 4.4, we see that the status bits that can be used as conditions for initiating a jump are logical greater than (L>), arithmetic greater than (A>), equal (EQ), carry (C), arithmetic overflow (AF), and odd parity (OP).

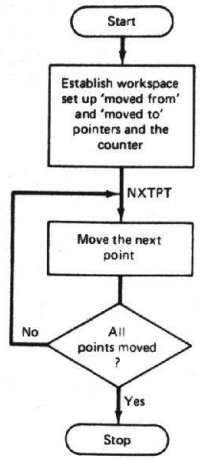
Notice that instructions are provided for different conditions of these status bits. For example, the *jump on equal* (JEQ) instruction tests the equal bit and initiates a jump if it is logic 1. Furthermore, a *jump on not equal* (JNE) instruction is provided to test the opposite condition. It initiates a jump if EQ tests as logic 0.

A similar set of instructions are provided for initiating jumps based on the carry bit. They are *jump on carry* (JOC) and *jump on no carry* (JNC).

The instructions JLT, JH, JHE, JL, and JLE test two status bits instead of one. Notice that in some cases both conditions must occur to initiate the jump, but in other cases either of the two conditions can occur. An example is the JLT instruction, which stands for "jump on less than arithmetic." We do not have a single status bit that indicates this condition. However, if both A> and EQ are cleared after the execution of an instruction, the result represents the arithmetic less than condition. Therefore, JLT tests both A> and EQ, and if they are both logic 0, the jump is initiated.

Example 4.2

It is required to move a set of N 16-bit data points that are stored in a block of memory starting at location BLK1 to a new block starting at location BLK2. Write a program to perform this function.



(R1) = Pointer to the data point to be moved

(R2) = Pointer to the location where the data point is to be placed

(R3) = Number of points to be moved

(a)

```
        LWPI    >0400
        LI      R1, BLK1
        LI      R2, BLK2
        LI      R3, N
NXTPT   MOV     *R1 +, *R2 +
        DEC     R3
        JNZ     NXTPT
```

(b)

Figure 4.6(a) Data block transfer program flowchart; (b) program.

Solution: The flowchart in Fig. 4.6(a) shows a plan for implementing the block data move function. Here we begin by setting up a workspace and then defining two of its registers as pointers and another as a counter. Workspace register R_1 is one pointer. It initially contains starting address BLK1 of the first block in memory. The other pointer is R_2 and it initially contains starting address BLK2 of the second block in memory. That is, R_1 points to the first storage location in the block of memory locations from which data are to be removed and R_2 points to the first storage location in the new block of memory locations where data are to be placed. The counter R_3 keeps track of how many words of data have been transferred. It is initially loaded with the value N.

The workspace pointer register, block address pointers, and counter can be initialized with the following sequence of instructions:


```
        LWPI    >0400
        LI      R1,BLK1
        LI      R2,BLK2
        LI      R3,N
```

A MOV instruction is used to perform the data transfer. To permit simple transfer of the next data point, indirect autoincrement addressing mode can be employed for both its source address pointer (R_1) and destination address pointer (R_2). In this way, the address is automatically incremented and no additional instructions are needed to increment the block address pointers. Each time a word is transferred, the count N in R_3 is decremented by 1. In this way, it tells how many more transfers must still take place. When it reaches zero, the block transfer is complete. The value of the count can be used to indicate whether or not to stop repeating the instruction sequence that performs the data transfer.

A sequence of instructions that perform the block move include a move operation, decrement of R_3, and a test and jump to the move instruction if the value in R_3 is not zero. This is performed with the following instruction sequence:

```
NXTPT   MOV     *R1+,*R2+
        DEC     R3
        JNZ     NXTPT
```

This part of the program is what is known as a *loop* and it is repeated N times. That is, there will be a jump performed from the JNZ instruction back to the statement with the NXTPT (next point) label. This jump occurs each time the JNZ instruction is executed until R_3 equals zero. When R_3 equals zero, the jump is not performed; instead, program execution continues with the next sequential instruction. The complete program is listed in Fig. 4.6(b).

4.4 PROGRAM EXAMPLES INVOLVING LOOPS

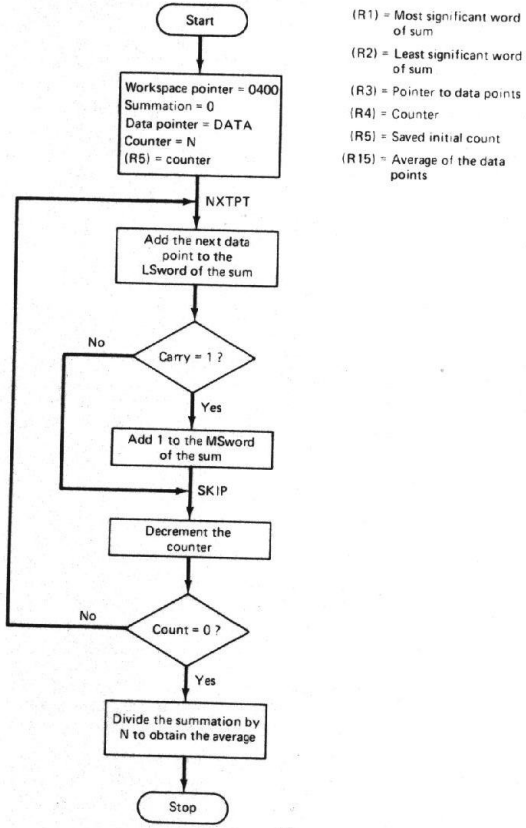
Now that we have introduced the compare and jump instructions and have shown how to form a software loop, let us consider some more complex programming examples that use these instructions and techniques.

Example 4.3

Given N data points that are unsigned 16-bit numbers and stored in consecutive memory locations starting at address DATA, write a program that finds their average value. Assume that the sum found by adding all N data points will not yield a result bigger than 32 bits.

Solution: A flowchart for solving this problem is shown in Fig. 4.7(a). It implements an algorithm that finds the average of N data points by adding their values and then dividing the sum by N. To do this in software, we must first set up a workspace. Then one of its registers can be dedicated as a pointer to the starting location of the table of data points in memory. For this pointer we have chosen R_3.

Another register can be used as a counter that indicates how many numbers have been added. This will be R_4. Two other registers, R_1 and R_2, will be assigned to hold the 32-bit sum and another register, R_{15}, will contain the average at the end of execution of the program. Moreover, N representing the number of data points to be averaged is saved in R_5.



(a)

```
        LWPI  >0400
        CLR   R2
        CLR   R1
        LI    R3, DATA
        LI    R4, N
        MOV   R4, R5
NXTPT   A     *R3+, R2
        JNC   SKIP
        INC   R1
SKIP    DEC   R4
        JNZ   NXTPT
        DIV   R5, R1
        MOV   R1, R15
```

(b)

Figure 4.7 (a) Flowchart of a program for finding the average of N unsigned numbers; (b) program.

The program begins by loading the workspace pointer and clearing the registers that are to accumulate the sum, R_1 and R_2. This is done with the instructions

```
LWPI  >0400
CLR   R2
CLR   R1
```

Next, R_3 is loaded with the address of DATA to act as a pointer and R_4 is loaded with the value of N. N is then saved in R_5 for later use in calculating the average. The instructions for this purpose are

```
LI    R3,DATA
LI    R4,N
MOV   R4,R5
```

For the summation of the N numbers in the table, we use the word addition instruction of the 99000. This instruction can be coded with indirect workspace addressing with autoincrement for its source operand R_3 and workspace direct addressing for its destination operand R_2. In this way, R_3 will always point to the next value in the table and the sum is generated in register R_2. Each time an addition is performed, the carry bit is tested. If it is 1, the contents of R_1 are also incremented by 1. Then the count in R_4 is decremented by 1. In this way, the value in R_4 represents the number of data points that remain to be added. The loop is terminated when the count is tested to be zero. If the count is nonzero, the summation routine is repeated to add the next data point.

The summation loop is written as follows:

```
NXTPT   A     *R3+,R2
        JNC   SKIP
        INC   R1
SKIP    DEC   R4
        JNZ   NXTPT
```

After this loop has executed N times, we will have accumulated the sum in R_1 and R_2. Once the sum of the N numbers in the table is available, the average can be obtained by simply dividing it by the original value of N held in R_5. For this reason, the program continues as follows:

```
DIV   R5,R1
MOV   R1,R15
```

This average calculation program is listed in Fig. 4.7 (b).

Example 4.4

It is common in computer systems to have a need to convert numeric data expressed in binary-coded decimal (BCD) code to binary form. This conversion can be done with hardware; however, it is more common to perform the conversion in

software. Write a program that will convert a four-digit BCD number to its equivalent binary number. The BCD number is stored as a 16-bit word.

Solution: Let us begin by defining an algorithm that can be used to convert a BCD number to a binary number. For the general BCD number,

$$N_{BCD} = D_3D_2D_1D_0$$

its equivalent decimal number is given by the expression

$$N_{10} = 1000(D_3) + 100(D_2) + 10(D_1) + D_0$$

This expression can be reorganized to give

$$N_{10} = D_0 + 10(D_1 + 10(D_2 + 10(D_3)))$$

This expression suggests an algorithm that can be implemented using a software loop. Notice that if we start with the MSD D_3, multiply it by 10, and then add the next MSD D_2, we will get our first temporary result. This same sequence can be performed twice more on the temporary result, adding first D_1 to the product and then D_0, to produce the final result.

The flowchart in Fig. 4.8(a) shows the implementation of this algorithm. We begin by establishing a workspace in memory. Registers R_1 and R_2 are assigned to hold the BCD number and its equivalent binary number, respectively. R_1 must be loaded with the BCD number, which is stored at the location called BCDN. Then R_2 is cleared. Moreover, we will assign R_4 as the digit counter and R_5 as the decimal scaler. These two registers must be initialized to 4 and 10, respectively. Initialization is performed with the following instruction sequence:

```
LWPI  >0400
LI    R1,@BCDN
CLR   R2
LI    R4,>0004
LI    R5,>000A
```

Looking at the algorithm, we see that it requires multiplication of the prior result by 10 and then its addition with the next less significant digit. This routine can be repeated until the conversion is complete. This part of the algorithm is implemented with the following sequence of instructions:

```
NXTDGT  MPY   R5,R2
        MOV   R3,R2
        MOV   R1,R3
        SRL   R3,12
        A     R3,R2
        SLA   R1,4
        DEC   R4
        JNE   NXTDGT
```

This part of the program starts with a statement identified with label NXTDGT. In this statement, we multiply the result, wich is the contents of R_2, by the scaler held in R_5. Initially, this is 0×10, equal to 0. Then the value of the BCD

number is moved into R_3 and shifted right by 12 bit positions. R_3 now contains the MSD D_3. This value is added to the result in R_2. Since the original contents of R_2 were zero, we have initialized the result with D_3. Next, R_1 is shifted left 4 bits;

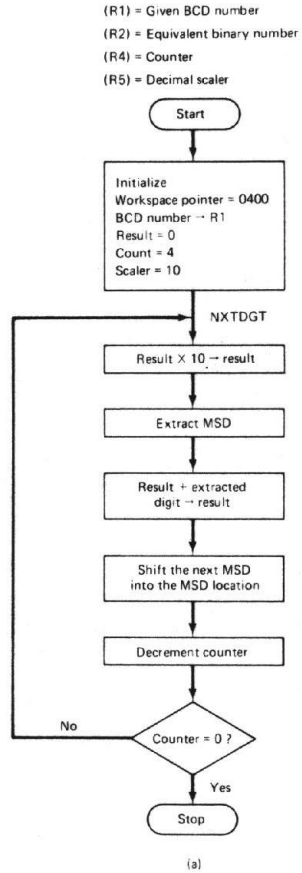


Figure 4.8(a) Flowchart for a BCD-to-binary conversion program; (b) program.

therefore, BCD digit D_2 resides in the MSD location. The value R_4 is decremented by 1 and then tested for 0. Since the condition is not satisfied, control is returned to NXTDGT. This sequence repeats until the value in R_2 is the equivalent binary number. The complete program is shown in Fig. 4.8(b).

Example 4.5

It is required to sort an array of 16-bit signed binary numbers such that they are arranged in ascending order. For instance, if the original array is

$$5, 1, 29, 15, 38, 3, -8, -32$$

After sorting, the array that results would be

$$-32, -8, 1, 3, 5, 15, 29, 38$$

Assuming that the numbers of the array are stored in memory at consecutive addresses from 0400_{16} through $04FE_{16}$, write a program that will sort them into ascending order.

Solution: First, we will develop an algorithm that can be used to sort an array of elements A_0, A_1, A_2 through A_N into ascending order. One way of doing this is to take the first number in the array, which is A_0, and compare it to the second number, A_1. If $A_0 > A_1$, the two numbers are swapped; otherwise, they are left alone. Next, A_0 is compared to A_2 and based on the results of this comparison, they are either swapped or left alone. This sequence is repeated until A_0 has been compared with all numbers up through A_N. When this is complete, the smallest number will be in the A_0 position.

Now A_1 must be compared to A_2 through A_N in the same way. After this is done, the second smallest number is in the A_1 position. Up to this point, just two of the N numbers have been put in ascending order. Therefore, the procedure must be continued for A_2 through A_{N-1} to complete the sort.

Figure 4.9 illustrates the use of this algorithm for an array with just four numbers. The numbers are $A_0 = 5$, $A_1 = 1$, $A_2 = 29$, and $A_3 = -8$. During the sort

I	0	1	2	3	Status
A(I)	5	1	29	−8	Original array
A(I)	1	5	29	−8	Array after comparing A(0) and A(1)
A(I)	1	5	29	−8	Array after comparing A(0) and A(2)
A(I)	−8	5	29	1	Array after comparing A(0) and A(3)
A(I)	−8	5	29	1	Array after comparing A(1) and A(2)
A(I)	−8	1	29	5	Array after comparing A(1) and A(3)
A(I)	−8	1	5	29	Array after comparing A(2) and A(3)

Figure 4.9 Sort algorithm demonstration.

sequence, $A_0 = 5$ is first compared to $A_1 = 1$. Since $5 > 1$, A_0 and A_1 are swapped. Now $A_0 = 1$ is compared to $A_2 = 29$. This time $1 < 29$; therefore, the numbers are not swapped and A_0 remains equal to 1. Next, $A_0 = 1$ is compared with $A_3 = -8$. A_0 is greater than A_3. Thus A_0 and A_3 are swapped and A_0 becomes equal to -8. Note in Fig. 4.9 that the lowest of the four numbers now resides in A_0.

The sort sequence in Fig. 4.9 continues with $A_1 = 5$ being compared first to $A_2 = 29$ and then $A_3 = 1$. In the first comparison, $A_1 < A_2$. For this reason, their values are not swapped. But in the second comparison, $A_1 > A_3$; therefore, the two values are swapped. In this way, the second lowest number, which is 1, is sorted into A_1.

It just remains to sort A_2 and A_3. Comparing these two values, we see that $29 > 5$. This causes the two values to be swapped such that $A_2 = 5$ and $A_3 = 29$. As shown in Fig. 4.9, the sorting of the array is now complete.

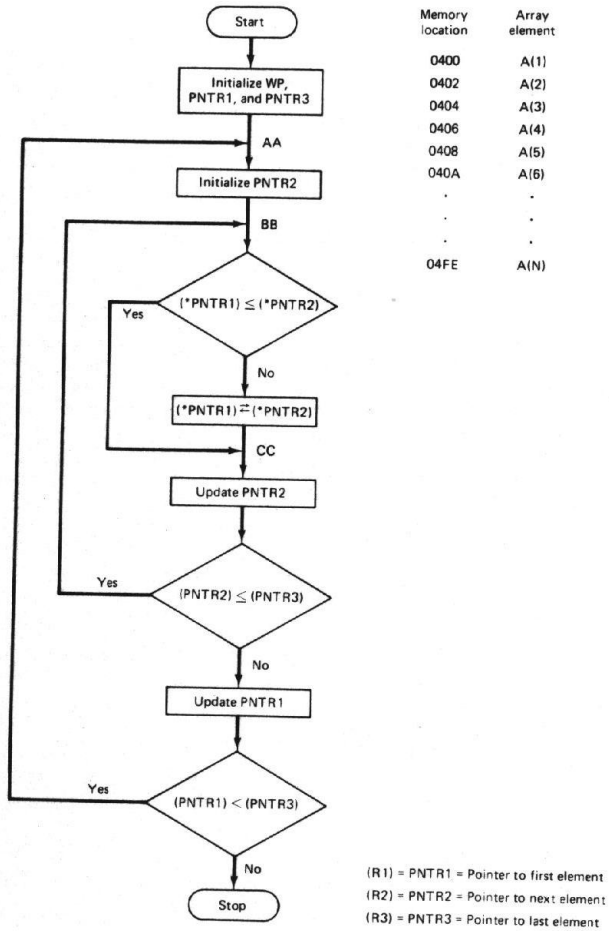
A flowchart showing how this algorithm can be implemented in software is shown in Fig. 4.10(a). The first block represents the initialization of the workspace pointer and two pointers called $PNTR_1$ and $PNTR_3$. The workspace pointer will be initialized to 0200_{16}. $PNTR_1$ is an address pointer to the first element in the array and is loaded into R_1 of the workspace. This element resides at address 0400_{16}. The other address pointer, $PNTR_3$, which identifies the location of the last element in the array, is loaded into R_3. This value is $04FE_{16}$. The instructions required to perform this are

```
LWPI  >0200
LI    R1,>0400
LI    R3,>04FE
```

Next, another pointer is established. This is $PNTR_2$, which is a pointer to the next element that is to be used for comparison. Initially, we make it equal to $PNTR_1$ and then increment it by 2 such that it points to the next word in the array. In the flowchart, we see that this is one of the places to which a conditional jump may be performed. Therefore, it is identified with a label in the program. We are using the label AA for this purpose. This requires the following instructions:

```
AA    MOV   R1,R2
      INCT  R2
```

Now comes the part in the program where the values in the array are compared to each other. Based on the results of these comparisons, the elements are swapped or left alone. We will start this part of the program with the label BB. First, the array element pointed to by $PNTR_1$ is compared to that pointed to by $PNTR_2$. Then the L> and EQ bits of the status register are tested to determine the relationship between the two elements. If the previous element is less than or equal to the next element, the numbers are not to be swapped and a jump is initiated to a part of the program identified by the label CC. Otherwise, the value stored at the location pointed to by $PNTR_2$ is stored temporarily in R_4. Then the value of the array element pointed to by $PNTR_1$ is moved to the location pointed to by $PNTR_2$



(a)

Figure 4.10(a) Flowchart for sort algorithm;


```
        LWPI    >0200
        LI      R1,>0400
        LI      R3,>04FE
AA      MOV     R1,R2
        INCT    R2
BB      C       *R1,*R2
        JLT     CC
        JEQ     CC
        MOV     *R2,R4
        MOV     *R1,*R2
        MOV     R4,*R1
CC      INCT    R2
        C       R2,R3
        JLT     BB
        JEQ     BB
        INCT    R1
        C       R1,R3
        JLT     AA
```

(b)

Figure 4.10(b) program.

and the value saved in R_4 is moved to the location pointed to by $PNTR_1$. This performs the swap of the two 16-bit numbers. This part of the program is

```
BB      C       *R1,*R2
        JLT     CC
        JEQ     CC
        MOV     *R2,R4
        MOV     *R1,*R2
        MOV     R4,*R1
```

This completes the first comparison. The address in R_2 is incremented by 2 such that it points to the next number in the array. Then the address in R_2 is compared to the end of array pointer address in R_3. If it is less than or equal to R_3, the first series of comparisons is not complete and program control returns to BB. This is performed with the following sequence of instructions, which correspond to the start of the part in the program denoted by CC:

```
CC      INCT    R2
        C       R2,R3
        JLT     BB
        JEQ     BB
```

When the address in R_2 is greater than that in R_3, the first number in the array has been compared to all other elements in the array. Now we must incre-

ment R_1 and then determine if it is pointing to the last element in the array. This is done by comparing it to R_3. If they are equal, the sort is complete. If not, control must be returned to the spot in the program labeled AA. This is achieved with the following program statements:

```
INCT   R1
C      R1,R3
JLT    AA
```

The complete program is listed in Fig. 4.10(b).

4.5 SUBROUTINES AND SUBROUTINE HANDLING INSTRUCTIONS

A *subroutine* is a special segment of program that can be called for execution from any point in a program. Figure 4.11 illustrates the concept of a subroutine. Here we see a program structure where one part of the program is called the *main program*. In addition to this, we find a smaller segment attached to the main program, known as a subroutine. The subroutine is written to provide a function that must be performed at various points in the main program. Instead of including this piece of code in the main program each time the function is needed, it is put into the program just once as a subroutine.

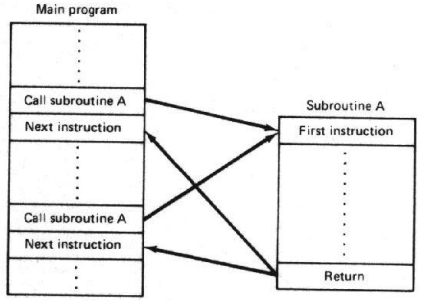


Figure 4.11 Subroutine concept.

Wherever the function must be performed, a single instruction is inserted into the main body of the program to "call" the subroutine. The *branch* (B), *branch and link* (BL), *branch and link with workspace* (BLWP), and *return workspace* (RTWP) *instructions* are included in the instruction set of the 99000 to deal with subroutines.

Remember that the contents of PC always identifies the next instruction to be executed by the 99000. Thus, to branch to a subroutine that starts elsewhere in memory, the value in PC must be modified. The B, BL, and BLWP instructions have this ability and can be used to pass control to the starting point of a subroutine.

After executing the subroutine, we want to return control to the instruction that follows the one that called the subroutine. In this way, program execution resumes in the main program at the point where it left off due to the subroutine call. However, just the BL and BLWP instructions automatically save the linkage required to complete this return. The B or RTWP instruction must be included at the end of the subroutine to initiate the *return sequence* to the main program environment.

The operations of the subroutine handling instructions are summarized in Fig. 4.12. Here we see that the BL instruction causes control to be transferred to a subroutine located at the address specified by its source operand. For instance, in Fig. 4.13, the instruction

```
BL  @SBRTA
```

causes the old value of PC, which points to the next instruction in the main program, to be saved in register R_{11} of the current workspace. In this way, we see that the return linkage is the old value of PC that is saved in R_{11}. The PC is loaded with the address value represented by the label SBRTA. SBRTA identifies the starting point of the subroutine in memory.

Instruction	Meaning	Format	Explanation
B	Branch	B S	(SA) → (PC)
BL	Branch and link	BL S	(PC) → (R11) (SA) → (PC)
BLWP	Branch and load workspace	BLWP S	(SA) → (WP) (SA) + 2 → (PC) (OLD WP) → (NEW R13) (OLD PC) → (NEW R14) (OLD ST) → (NEW R15) The $\overline{\text{INTREQ}}$ input is not tested after completion of the instruction
RTWP	Return workspace pointer	RTWP	(R13) → (WP) (R14) → (PC) (R15) → (ST)

Figure 4.12 Subroutine handling instructions.

To return to the main program, the last instruction of the subroutine should be

```
B  *R11
```

Execution of this branch indirect through R_{11} instruction causes the old value of PC to be returned from R_{11} to the program counter.

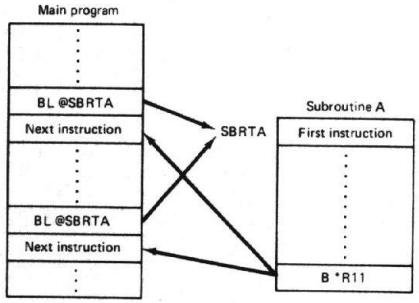


Figure 4.13 Subroutine program structure using the BL and B instructions.

It is usually necessary for the main program to pass data to the subroutine that it is calling. These data are known as *parameters* and are passed to the subroutine by simply moving them into specific workspace registers or memory locations prior to calling the subroutine.

Let us consider an example of passing parameters to a subroutine. Assume that the function of the subroutine is to add the contents of registers R_1 and R_2 and place the sum in R_3. This can be done with the following instructions:

```
A     R1,R2
MOV   R2,R3
```

Whenever this function must be performed, it is called from the main program as a subroutine. However, before calling it, registers R_1 and R_2 of the current workspace must be loaded with the numbers to be added. This is called *passing parameters.*

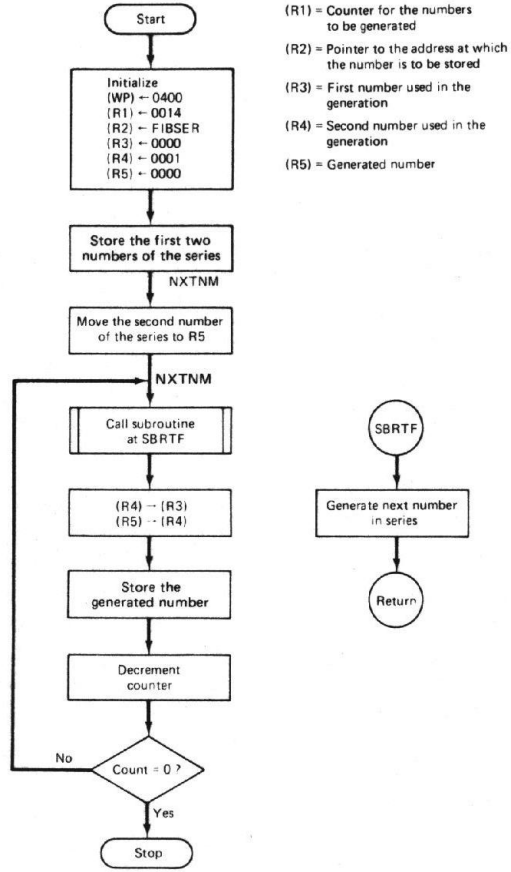
Example 4.6

Write a program to generate the first 22 elements of a Fibonacci series. In such a series, the first number is 0, the second is 1, and each subsequent number is obtained by adding the previous two numbers. For example, the first 10 numbers of the series are

0, 1, 1, 2, 3, 5, 8, 13, 21, 34

The part of the procedure by which the next number is obtained from the previous two is to be performed in a subroutine. The series of numbers are to be stored in consecutive memory locations starting at address FIBSER.

Solution: A flowchart for the program together with the assignments of various workspace registers are shown in Fig. 4.14(a). The first part of the program defines a workspace and initializes some of its registers. This is done with the instructions



(a)

```
        LWPI   >0400
        LI     R1, >0014
        LI     R2, FIBSER
        CLR    R3
        LI     R4, >0001
        CLR    R5
        MOV    R5, *R2 +
        MOV    R4, *R2 +
        MOV    R4, R5
NXTNM   BL     @SBRTF
        MOV    R4, R3
        MOV    R5, R4
        MOV    R5, *R2 +
        DEC    R1
        JNE    @NXTNM
SBRTF   A      R3, R5
        B      *R11
```

(b)

Figure 4.14(a) Flowchart of a program for generation of a Fibonacci series; **(b)** program.

```
LWPI  >0400      SET UP WORKSPACE
LI    R1,>0014   INITIALIZE COUNTER
LI    R2,FIBSER  INITIALIZE POINTER
CLR   R3         INITIALIZE FIRST NUMBER FOR GENERATION
LI    R4,>0001   INITIALIZE SECOND NUMBER FOR GENERATION
CLR   R5         INITIALIZE GENERATED NUMBER
MOV   R5,*R2+    FIRST NUMBER OF SERIES
MOV   R4,*R2+    SECOND NUMBER OF SERIES
```

The last two instructions cause 0 and 1 to be loaded into addresses FIBSER and FIBSER+2, respectively. Therefore, the first two value of the series has been stored and R_2 points to the storage location of the third number.

To generate the third number in the series, R_5 is loaded from R_4 with the instruction

```
MOV  R4,R5
```

We are now ready to call the subroutine that does the addition to form the next number in the series. This instruction must be a reference point to which the program can loop; therefore, it is identified with the label NXTNM. It is implemented with the instruction

```
NXTNM  BL  @SBRTF
```

From the branch instruction, we see that the start of the subroutine is identified by the label SBRTF. This sequence of instructions must add the present element of the series to the previous element to get the new element. This is done with the instructions

```
SBRTF  A  R3,R5
       B  *R11
```

After executing the subroutine, the branch instruction returns control to the main program. Now some housekeeping is performed. The previous series number is saved in R_3 and the present number is saved in R_4. This represents passing of the next set of parameters to the subroutine. The present number is stored in the series location identified by the address in R_2. Autoincrement addressing is used such that R_2 always points to the storage location for the next element of the series. This is achieved with the following instructions:

```
MOV  R4,R3
MOV  R5,R4
MOV  R5,*R2+
```

Now the count in R_1 is decremented and tested for zero. If it is zero, the program is complete; otherwise, control is returned to NXTNM for generation of

the next number in the series. This part of the program is represented by the instructions

```
DEC  R1
JNE  @NXTNM
```

The complete program is presented in Fig. 4.14(b).

The BLWP instruction provides an alternative means for calling a subroutine. The mechanism that it performs is called a *context switch*. The operation of the instruction is described in Fig. 4.12. Notice that execution of BLWP causes new addresses to be loaded into the WP and PC registers of the 99000. The locations of these addresses are specified by the source operand. The address in WP defines a new workspace and that in PC the instruction where execution is to resume. Moreover, the old values of WP, PC, and ST are automatically saved in registers R_{13}, R_{14}, and R_{15}, respectively, of the new workspace. In this way, a new program context is defined.

At the end of the subroutine, a RTWP instruction must be included to restore the original program environment. As shown in Fig. 4.12, execution of RTWP causes the old values of WP, PC, and ST to be returned to their corresponding registers within the 99000.

Example 4.7

Describe the structure and execution of the context switch mechanism for calling subroutines relative to the diagram in Fig. 4.15.

Solution: Looking at Fig. 4.15, we see that the main program starts at the point marked MPGM (main program) and it uses a workspace whose location is identified by WSM (main workspace). There are two subroutine calls from the main program, one to SUBA (subroutine A) and the other to SUBB (subroutine B). These subroutine calls are initiated by BLWP instructions at addresses CSBA (call subroutine A) and CSBB (call subroutine B), respectively, of the main program. The vectors PCA (program counter A) and WSA (workspace pointer A) that identify the starting point of subroutine A and its workspace are stored at VA and VA+2 (vector A), respectively. Moreover, PCB (program counter B) and WSB (workspace pointer B) are stored at VB and VB+2 (vector B), respectively. In this way, we see that each subroutine as well as the main program has its own workspace.

Execution of the program begins with the instruction located at the location identified by MPGM in program memory. From this point, instructions are executed sequentially until the BLWP instruction at CSBA is encountered. As this instruction is executed, a context switch is initiated. The vector is specified as the contents of VA and VA+2. At VA, address WSA is stored and at VA+2 address PCA. These values are loaded into the WP and PC of the 99000, respectively. Then the 99000 stores the old values of WP (WSM), PC (CSBA+2), and status (ST) in registers R_{13} through R_{15} of the workspace at WSA.

Now the 99000 continues execution with the first instruction of subroutine A. Execution proceeds sequentially through the instructions of the subroutine until

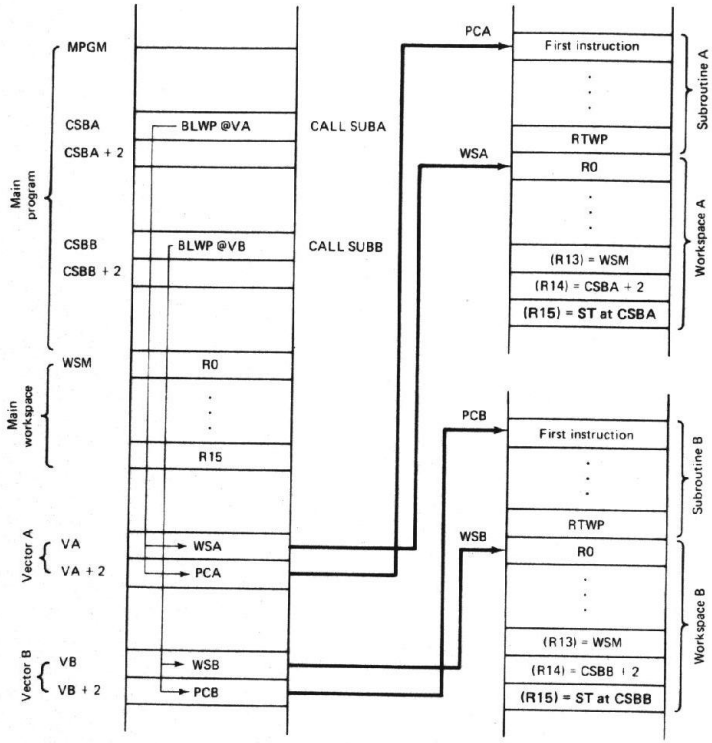


Figure 4.15 Context switch mechanism for the BLWP instruction.

the RTWP instruction is executed. This instruction initiates the return context switch to the main program. It causes the values of WSM, CSBA+2, and ST to be returned to the registers of the 99000 and execution resumes with the instruction at CSBA+2. From here, execution continues sequentially in the main program.

When the second BLWP instruction is executed, another context switch is performed. This time the vector WSB and PCB at VB and VB+2 are loaded into the WP and PC; the old WP (WSM) is saved in R_{13}, the old PC (CSBB+2) is saved in R_{14}, and the old status (ST) is saved in R_{15} of the workspace at WSB; and program control continues with the first instruction of subroutine B. Once again the last instruction executed in the subroutine is RTWP, which causes control to be returned to the main program at address CSBB+2.

ASSIGNMENT

Section 4.2

1. Describe the difference in operation and affect on status due to execution of the subtract words and compare words instructions.
2. What happens to the EQ status bit as the following sequence of instructions is executed? Assume that EQ is initially cleared.

```
LI   R1,>1234
MOV  R1,R2
C    R1,R2
CZC  R1,R2
COC  R1,R2
```

Section 4.3

3. The program that follows implements what is known as a delay loop.

```
     LI   R7,>1000
DLY  DEC  R7
     JGT  DLY
NXT  --   ---
```

(a) How many times does the JGT DLY instruction get executed?
(b) Change the program so that JGT DLY is executed just 17 times.
(c) Change the program so that JGT DLY is executed 2^{32} times.

Section 4.4

4. Given a number N in the range $0 < N \leq 5$, write a program that computes its factorial and saves the result in memory location FACT.
5. Write a program that compares the elements of two arrays A(I) and B(I). Each array contains 100 16-bit signed numbers. The comparison is to be done by comparing the corresponding elements of the two arrays until either two elements are found to be unequal or all elements of the arrays have been compared and found to be equal. Assume that the arrays start at addresses >A000 and >B000, respectively. If the two arrays are found to be unequal, save the address of the first unequal element of A(I) in memory location FOUND; otherwise, write all 0s into this location.
6. Given an array A(I) of 100 16-bit signed numbers that are stored in memory starting at address >A000, write a program to generate two arrays from the given array such that one P(J) consists of only positive numbers and the other N(K) contains only negative numbers. Store the array of positive numbers starting at address >B000 in memory and the array of negative numbers at address >C000.

7. Given a 16-bit binary number in R_0, write a program that converts it to its equivalent BCD number in R_0. If the result is bigger than 16 bits, place all 1s in R_0 and R_1.
8. Given an array A(I) of 100 16-bit signed integer numbers, write a program to generate a new array B(I) as follows:

$$B(I) = A(I) \quad \text{for I} = 1, 2, 99, \text{and } 100$$

and

$$B(I) = \text{median value of } A(I-2), A(I-1), A(I), A(I+1), \text{ and } A(I+2) \quad \text{for all other Is}$$

Section 4.5

9. Write a subroutine that converts a given 16-bit BCD number to its equivalent binary number. The BCD number is to be passed to a subroutine through register R_7 and the routine returns the equivalent binary number in R_7.
10. Given an array A(I) of 100 16-bit signed integer numbers, write a subroutine to generate a new array B(I) such that

$$B(I) = A(I) \quad \text{for I} = 1 \text{ and } 100$$

and

$$B(I) = \frac{1}{4}(A(I-1) + 2A(I) + A(I+1)) \quad \text{for all other Is}$$

The values of A(I−1), A(I), and A(I+1) are to be passed to the subroutine in registers R_5, R_6, and R_7 and the subroutine returns the result B(I) in register R_5.

11. Write a segment of main program and show its subroutine structure to perform the following operations. The program is to check repeatedly the three least significant bits in register R_1 and depending on their setting execute one of three subroutines: SUBA, SUBB, or SUBC. The subroutines are selected as follows:
(1) If bit 15 of R_1 is set, initiate SUBA.
(2) If bit 14 of R_1 is set and bit 15 is not set, initiate SUBB.
(3) If bit 13 of R_1 is set and bits 14 and 15 are not set, initiate SUBC.
If the subroutine is executed, the corresponding bit of R_1 is to be cleared and then control returned to the main program. After returning from the subroutine, the main program is repeated.

5

MEMORY INTERFACE OF THE 99000

5.1 INTRODUCTION

Up to this point in the book, we have introduced the 99000 microprocessor, its signal leads and internal architecture. Moreover, from a software point of view, we have covered its instruction set and how to write programs in assembly language. Now we will begin to examine the hardware interfaces of the 99000. This chapter is devoted to its memory interface and external memory subsystems. For this purpose, we have included the following topics:

1. Memory interface block diagram
2. Address space
3. Data organization
4. Dedicated and general use of memory
5. Memory bus status codes and memory control signals
6. Read cycle
7. Write cycle
8. Slow memory interface
9. Demultiplexing the 99000 system bus
10. EPROM/static RAM memory subsystem
11. Extending the address space of the 99000
12. Memory subsystem with error detection and correction
13. Cache memory for the 99000 system

5.2 MEMORY INTERFACE BLOCK DIAGRAM

The 99000 microprocessor has the ability to address a memory subsystem directly with up to 256K bytes of memory. Figure 5.1 shows the *memory system interface.* It consists of the *multiplexed address/data bus,* which is used to carry address information to the memory subsystem and transfer data between memory and microprocessor. By "multiplexed" we mean that it works as an address bus and data bus at different periods of time during the bus cycle. The ALATCH signal indicates when a valid address is on the bus. Bus status lines $\overline{MEM}$ and BST_1 through BST_3 indicate whether or not a memory bus cycle is in progress and if so, which type of memory cycle it is. The last three signals, $R/\overline{W}$, $\overline{WE}$, and $\overline{RD}$, indicate the direction in which the data are to be carried over the bus, that is, whether a read or write operation is to take place and when valid data are on the bus.

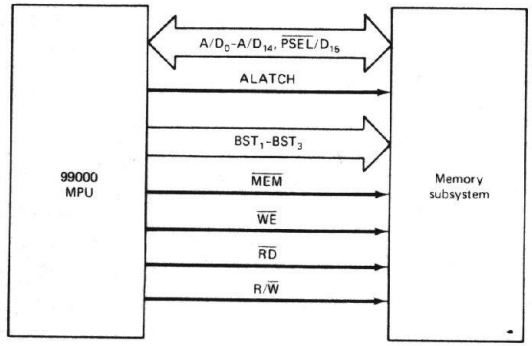


Figure 5.1 99000 memory interface.

5.3 ADDRESS SPACE

As shown in Fig. 5.2(a), the 99000 has a 15-bit external address bus. These lines are labeled A_0 through A_{14} and are used to carry address words from the microprocessor to the memory system. However, the program counter and workspace registers that are internal to the 99000 are 16 bits in length. For this reason, a 16-bit address is shown with A_0 representing the MSB and A_{15} the LSB.

The extra address bit, A_{15}, is not provided on a pin for external use;

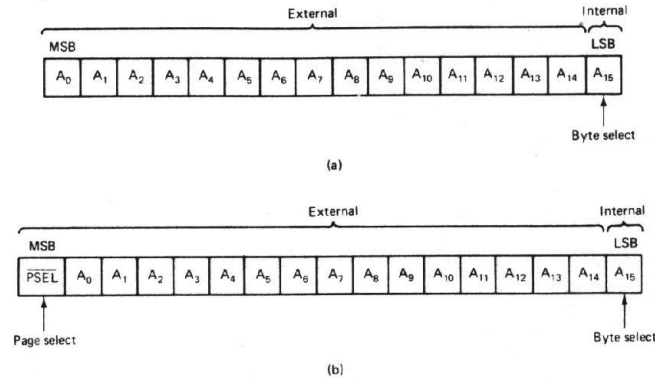


Figure 5.2(a) Address word format; (b) extended address word format.

instead, it is retained within the 99000. For instruction fetches, this bit is always 0. Therefore, instruction words always reside at even addresses. But for data accesses, it is used for byte selection during byte data operations. Logic 0 at internal bit A_{15} selects what is known as the *even byte* and logic 1 selects the *odd byte.*

With just a 16-bit address, the 99000 has the ability to address up to $2^{16} = 65{,}536$ *byte locations* in memory. This is sometimes called the *address reach* of the microprocessor or its *logical address space.* If we consider the address reach of the 99000 from a 16-bit word point of view, the 15-bit external bus provides for access of $2^{15} = 32{,}768$ *word locations.*

The address reach of the 99000 can actually be extended in a paged mode to *two 64K-byte pages.* This gives a total memory system storage of 128K bytes. To do this, bit ST_8 of the status register is employed as a seventeenth address bit and its logic level is multiplexed out at pin 31 ($\overline{PSEL}$) together with the address. This *extended address* is shown in Fig. 5.2(b). In this way, software can be used to make status bit ST_8 logic 0 or logic 1. In turn, the $\overline{PSEL}$ signal can be used to enable one of two 64K-byte pages of memory for operation.

The 99000's address reach can be further expanded to 256K bytes by decoding the memory bus status codes. The codes output on BST_1 through BST_3 can be decoded with external circuitry to produce a *program memory select signal* and a *data memory select signal.* In this way, the memory subsystem can be segmented into a *128K-byte program memory segment* and a *128K-byte data memory segment.*

5.4 DATA ORGANIZATION

Data and instruction transfers between the 99000 and its memory subsystem take place over the 16-bit data bus, D_0 through D_{15}. Actually, four different formats of data transfers can take place over this bus. They are *bit*, *byte*, *word*, and *long-word transfers*. Which type of transfer takes place depends on the instruction that is being executed.

Word data transfers are the most common type. This is because all instruction fetches and many operand transfers are 16 bits in length. For word transfers, the data bus is organized as shown in Fig. 5.3(a). Here D_0 identifies the MSB and D_{15} the LSB. Notice that D_0 can also be used as a *sign bit* for representation of *signed numbers*.

An example of an instruction that initiates word data transfers is the

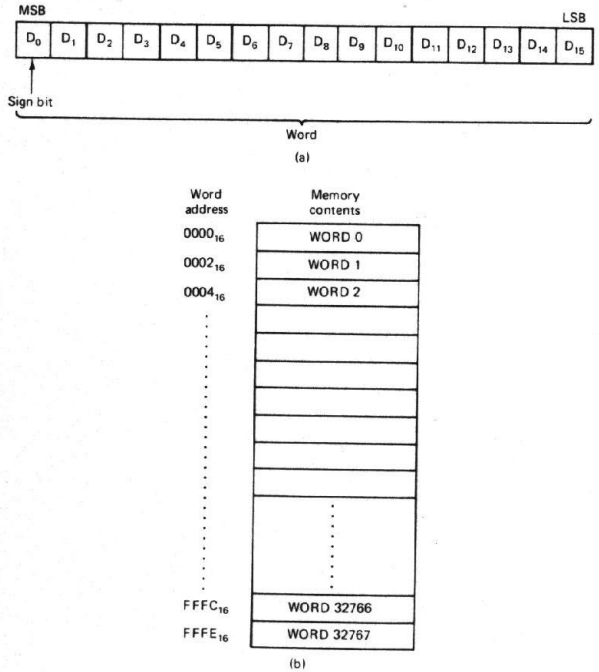


Figure 5.3(a) Word data format; (b) word organization of memory.

add word (A) instruction. It causes 16-bit source and destination operands to be transferred over the bus.

Figure 5.3(b) shows how word data are stored in the memory of a 99000 microcomputer. As far as the microprocessor is concerned, word data are stored at what are called even address boundaries; that is, even-addressed locations such as 0000_{16}, 0002_{16}, 0004_{16}, and up through $FFFE_{16}$. This happens because the LSB of the address, A_{15}, is not used in word accesses of memory and is always set at 0. Therefore, the address internal to the microprocessor is an even binary number.

The 99000 also has the ability to execute many of its instructions with bytes of data as well as words. The byte organization of the data bus is shown in Fig. 5.4(a). Here we see that even bytes are passed over bus lines

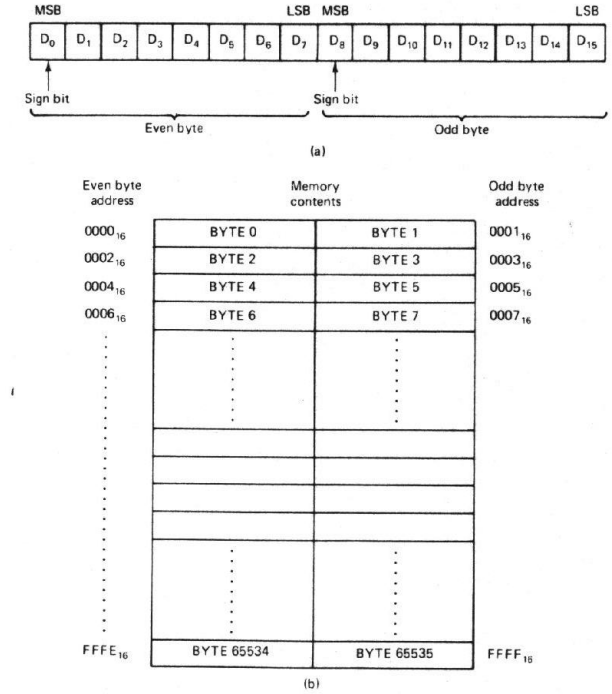


Figure 5.4(a) Byte data format; (b) byte organization of memory.

D_0 through D_7, where D_0 represents the MSB and D_7 the LSB. On the other hand, odd bytes are carried over bus lines D_8 through D_{15}. In this case, D_8 is the MSB and D_{15} the LSB.

The byte organization of data in memory is shown in Fig. 5.4(b). Notice that each 16-bit word location is used to store two bytes, called the odd byte and the even byte. The "add byte" (AB) instruction is an example of an instruction that processes data in byte format. Actually, when executing this instruction, the 99000 always accesses the full word operands. However, it selects either the odd byte or the even byte based on the logic level of internal address bit, A_{15}, and processes it. The other byte is passed through unaffected.

Some of the instructions of the 99000 perform operations on what are called *long words* of data. These 32-bit operands are stored in two consecutive 16-bit memory locations and are passed over the data bus as two consecutive word transfers. An example of an instruction that processes data in the long-word format is the double-precision add (AM) instruction.

Example 5.1

Assume that the sign bit of a data word is set to logic 0 for positive numbers and to 1 for negative numbers. If the data byte -32_{10} is to be written over the bus to address $FF0B_{16}$ in memory, what is the data word in binary form, in hexadecimal form, and over which bus lines is it carried?

Solution: Let us begin by converting the decimal number 32 to binary form. This gives

$$32_{10} = 00100000_2$$

Expressing it as an 8-bit negative number by taking the 2's complement, we get

$$-32_{10} = 11100000_2$$

and in hexadecimal form it is

$$-32_{10} = E0_{16}$$

Address $FF0B_{16}$ is an odd-byte address. Therefore, the data are carried over data bus lines D_8 through D_{15} to the memory system.

5.5 DEDICATED AND GENERAL USE OF MEMORY

In the 99000 microcomputer system, some of the storage locations in memory have dedicated functions. Figure 5.5 is a *memory map* that shows the 99000's logical address space and its dedicated areas. From the map we see that the first 32 word locations of memory are reserved for storage of *interrupt vectors*. This corresponds to the address range from 0000_{16} to $003E_{16}$. These vectors identify the starting points of the interrupt service routines in memory.

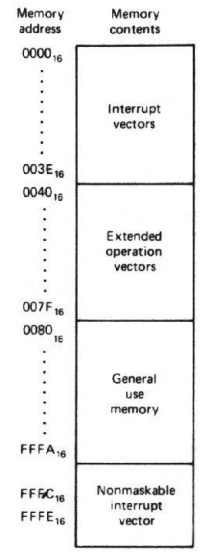


Figure 5.5 Dedicated and general use of memory.

Following this block in memory is another block of 32 reserved word locations. These locations have a similar dedicated function; however, the vectors stored here are for the 99000's *extended operation instructions*. They are located over the address range 0040_{16} to $007E_{16}$.

The last reserved memory area is used to store a vector for the *nonmaskable interrupt service routine*. This vector requires just two word locations of memory and they are located at $FFFC_{16}$ and $FFFE_{16}$.

The address space from 0080_{16} through $FFFB_{16}$ is *general use area* and can be used for program storage or data storage memory. Moreover, any part of the address space of the 99000's memory system can be implemented with ROM or RAM devices.

5.6 MEMORY BUS STATUS CODES AND MEMORY CONTROL SIGNALS

Let us now look at the signals that are used to demultiplex the address/data bus and control the timing of read/write data transfers. Earlier we indicated that the 99000 outputs bus status codes to identify which type of bus cycle

is taking place. During all bus operations to memory, the *memory enable* signal $\overline{MEM}$ is at logic 0. At the same time, a *bus status code* is output on BST_1 through BST_3. It identifies which of the different types of memory accesses is taking place over the address/data bus.

A table of bus status codes is shown in Fig. 5.6. Here the various memory bus status codes are highlighted. For instance, whenever an instruction is being fetched from memory over the data bus, the *instruction acquisition* (IAQ) code is output on the status bus. If the instruction is a two-word instruction or is followed by immediate data, the read of the second word from memory is accompanied by the *immediate operand* (IOP) code on the status bus.

$\overline{MEM}$	BST_1	BST_2	BST_3	Mnemonic	Name
0	0	0	0	SOPL	Source operand with MPILCK
0	0	0	1	SOP	Source operand
0	0	1	0	IOP	Immediate operand
0	0	1	1	IAQ	Instruction acquisition
0	1	0	0	DOP	Destination operand
0	1	0	1	INTA	Interrupt acknowledge
0	1	1	0	WS	Workspace
0	1	1	1	GM	General memory
1	0	0	0	AUMSL	ALU or macrostore with MPILCK
1	0	0	1	AUMS	ALU or macrostore
1	0	1	0	RESET	Reset
1	0	1	1	IO	Input/output
1	1	0	0	WP	Workspace pointer
1	1	0	1	ST	Status register
1	1	1	0	MID	Macroinstruction detect
1	1	1	1	HOLDA	Hold acknowledge

Figure 5.6 Memory bus status codes.

On the other hand, if an instruction requires reading source and destination operands from general memory, the corresponding memory cycles are identified by the *source operand transfer* (SOP) and *destination operand transfer* (DOP) bus status codes, respectively.

Source and destination operands can be read from workspace registers rather than locations in general memory. In this case, the *workspace transfer* (WS) code is output during each read cycle instead of SOP and DOP codes.

The other four signals that control the interface between the 99000 microprocessor and memory are R/$\overline{W}$, ALATCH, $\overline{WE}$, and $\overline{RD}$. ALATCH indicates whether the bus is set up for operation as an address bus or data bus. Logic 0 on ALATCH indicates that the bus is carrying address information and 1 indicates data. During write operations, R/$\overline{W}$ and $\overline{WE}$ switch to logic 0 to indicate to the memory system that the bus is in the output state and valid data are on the bus. R/$\overline{W}$ provides an *early write* signal for use with high-speed memory subsystems. On the other hand, when an instruction or data are read from memory, $\overline{RD}$ switches to logic 0. It tells the memory system that the bus is set up for reading of data and that the memory system should put data from the addressed location onto the bus.

5.7 READ CYCLE

Now that we have introduced the memory interface of the 99000, let us continue by tracing the sequence of events that occur when an instruction or data operand is read from memory.

The sequence of signals that take place at the memory interface during a read operation is shown in Fig. 5.7. Notice that the *read cycle* begins with the 0-to-1 transition of ALATCH. This indicates that a valid address is on the system bus. At the same time, $\overline{MEM}$ switches to the 0 level, indicating that a memory cycle is in progress and an appropriate bus status code (IAQ, IOP, SOP, DOP, or WS) is output on BST_1 through BST_3. On the 1-to-0 transition

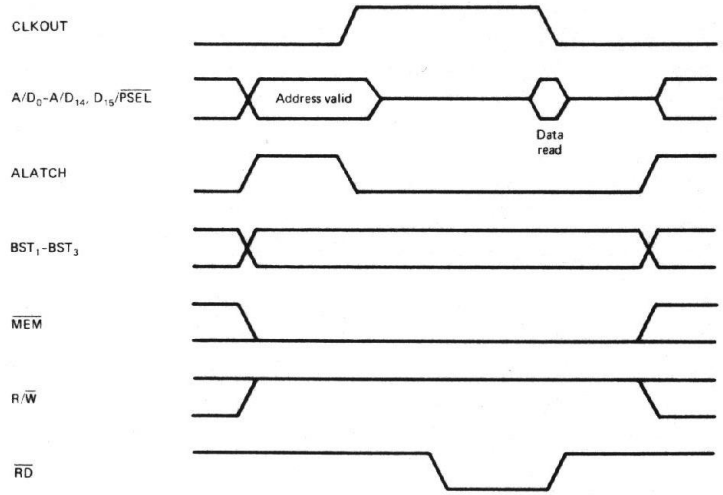


Figure 5.7 Read bus cycle timing.

of ALATCH, the address must be latched into external circuitry and applied to the memory subsystem. Notice that the address actually remains valid past the falling edge of ALATCH. This assures that the correct value is latched up.

As the 99000 switches $\overline{RD}$ to logic 0, it signals the memory subsystem to put data onto the bus. On the 1-to-0 transition of CLKOUT that follows, the 99000 reads the data off the bus.

Notice that the 99000 has the ability to perform a memory operation in just one CLKOUT cycle. Earlier we indicated that CLKOUT was identical to the machine cycle of the 99000. Therefore, at 6 MHz (24 MHz crystal) its memory bus cycle time is 167 ns.

5.8 WRITE CYCLE

The signals and timing involved in the memory *write cycle* of the 99000's microcomputer system are similar to those found in the read cycle. Looking at Fig. 5.8, we find that the write cycle also begins with the 0-to-1 transition of ALATCH. The address and status code are output onto their corresponding buses and $\overline{MEM}$ is pulled to the 0 logic level, indicating that a

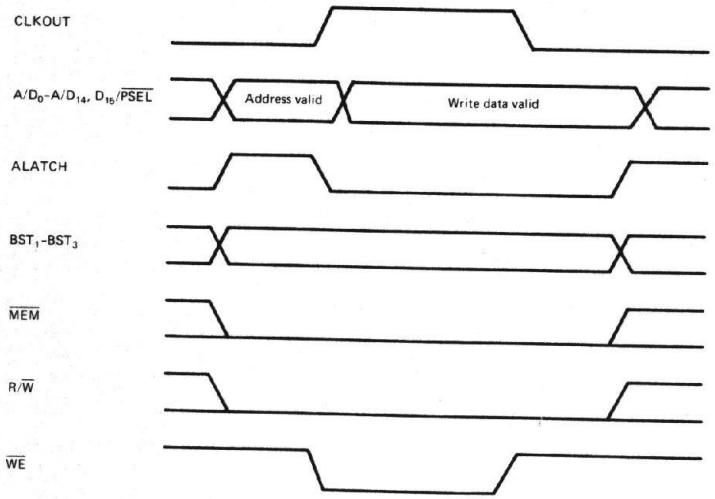


Figure 5.8 Write bus cycle timing.

memory cycle is in progress. However, this time R/$\overline{W}$ is switched to logic 0 to give an early signal that a write cycle has just started. The address must again be latched in external circuitry as ALATCH returns to the 0 level.

The 99000 outputs the data that are to be written into the storage location onto the data bus. This time $\overline{RD}$ remains at logic 1 and $\overline{WE}$ is switched to logic 0. In this way, the memory system is signaled that valid data are on the bus and that it should complete the write operation.

5.9 SLOW MEMORY INTERFACE

We just showed that the standard bus cycle of the 99000 takes just 167 ns to be completed. For the 99000 microcomputer system to operate in this way, its memory subsystem must be designed with very fast static memory devices. However, the 99000's memory interface also has the ability to work with slower memories. This is achieved through the use of the READY signal that has been added to the interface as shown in Fig. 5.9.

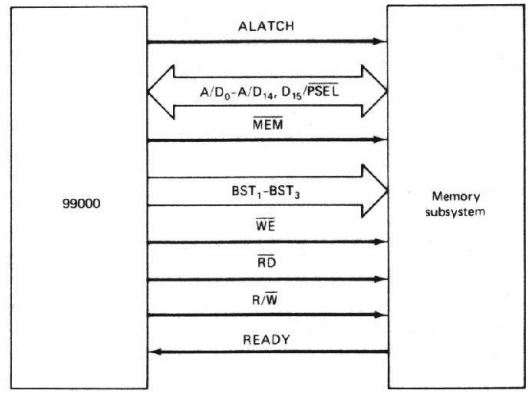


Figure 5.9 Slow memory interface.

READY is used to insert multiple wait states into the memory bus cycle. It gets tested each machine cycle of a memory operation. If the READY input is logic 1 when tested by the 99000, the memory cycle is completed. On the other hand, if it is logic 0, the read or write of data does not occur. Instead, READY is sampled in each machine state that follows and the memory cycle is extended until READY returns to logic 1. This per-

mits creation of a memory cycle in which the memory system provides the READY signal to indicate when read or write cycles are to be completed.

For instance, if the bus cycle is extended by one machine cycle, the new bus cycle duration is

$$t_{CYL1} = 2t_{CYL}$$
$$= 2(167 \text{ ns})$$
$$= 334 \text{ ns}$$

This longer bus cycle permits the 99000 to work with a memory subsystem designed with slower memory devices such as dynamic RAMs.

Example 5.2

What is the memory cycle time of a memory system for the 99000 that has two wait states inserted with the READY signal? Assume that the 99000 is operating at 6 MHz (24 MHz crystal).

Solution: **When two wait states are inserted, the memory bus cycle takes three machine states to complete.**

$$t_{CYL2} = 3t_{CYL} = 3(167 \text{ ns})$$
$$= 501 \text{ ns}$$

5.10 DEMULTIPLEXING THE 99000'S SYSTEM BUS

The system bus of the 99000 is designed to support a very short duration memory bus cycle. Earlier we mentioned that a bus cycle without wait states takes just 167 ns. This short bus cycle permits the use of fast memory devices, such as 35-ns-access-time static RAM, in the 99000's memory subsystem. This is typically done only for applications that require very high performance operation. The result of this short bus cycle is a much higher *bus bandwidth* than for the other 16-bit microprocessors that are available.

Remember that the address and data buses of the 99000 are multiplexed and must be demultiplexed with external circuitry. Because of the high-speed bus operation, the multiplexed address/data bus signals cannot be distributed throughout the microcomputer system. The delays experienced in distributing them would result in incorrect bus operation. Instead, they must be demultiplexed with circuitry placed immediately next to the address/data pins.

Figure 5.10 shows how the bus in a 99000 microcomputer system can be demultiplexed. Here we find that two 74ALS573 *transparent octal latches* are used to latch addresses output on the A_0 through A_{14} lines. When ALATCH switches to logic 1, the latches are enabled. Since they are transparent latches, a valid address is available at their outputs A_0L through $A_{14}L$ 12 ns after the address on A_0 through A_{14} becomes valid. This address is latched into the outputs as ALATCH returns to logic 0.

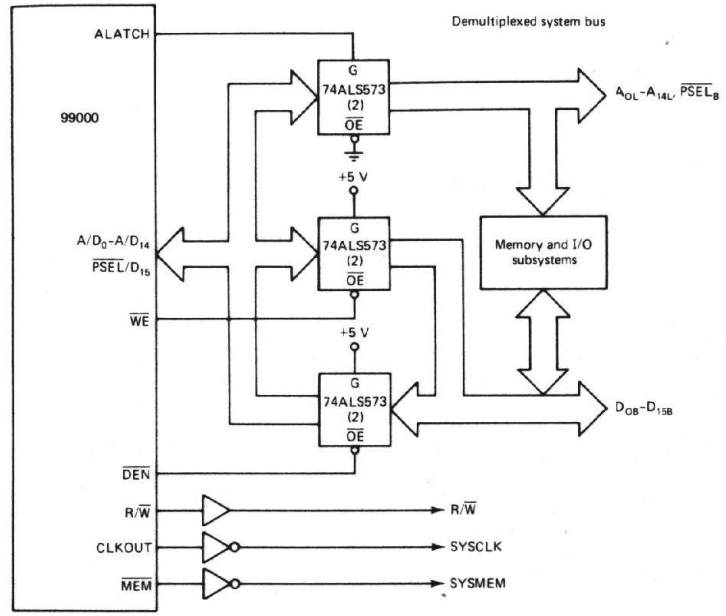


Figure 5.10 Demultiplexing the system address/data bus.

The *data bus buffer* part of the system bus demultiplexing circuit must also be located physically close to the 99000 device. Notice in Fig. 5.10 that two 74ALS573 devices are also used here. The upper two devices provide a data path for writing to memory. When the 99000 switches $\overline{WE}$ to logic 0 during the data-transfer part of the write bus cycle, the data that it has output on D_0 through D_{15} are passed to the buffered system data bus lines D_{0B} through D_{15B} and the memory subsystem.

On the other hand, during read bus cycles, the 99000 accesses data stored in memory through the lower two 74ALS573 devices. The outputs of these devices are enabled by $\overline{DEN}$ to put the data at the outputs of memory onto data bus lines D_0 through D_{15}.

Three buffered control signals complete the demultiplexed system bus. They are buffered read/write ($R/\overline{W}$), system clock (SYSCLK), and system memory (SYSMEM).

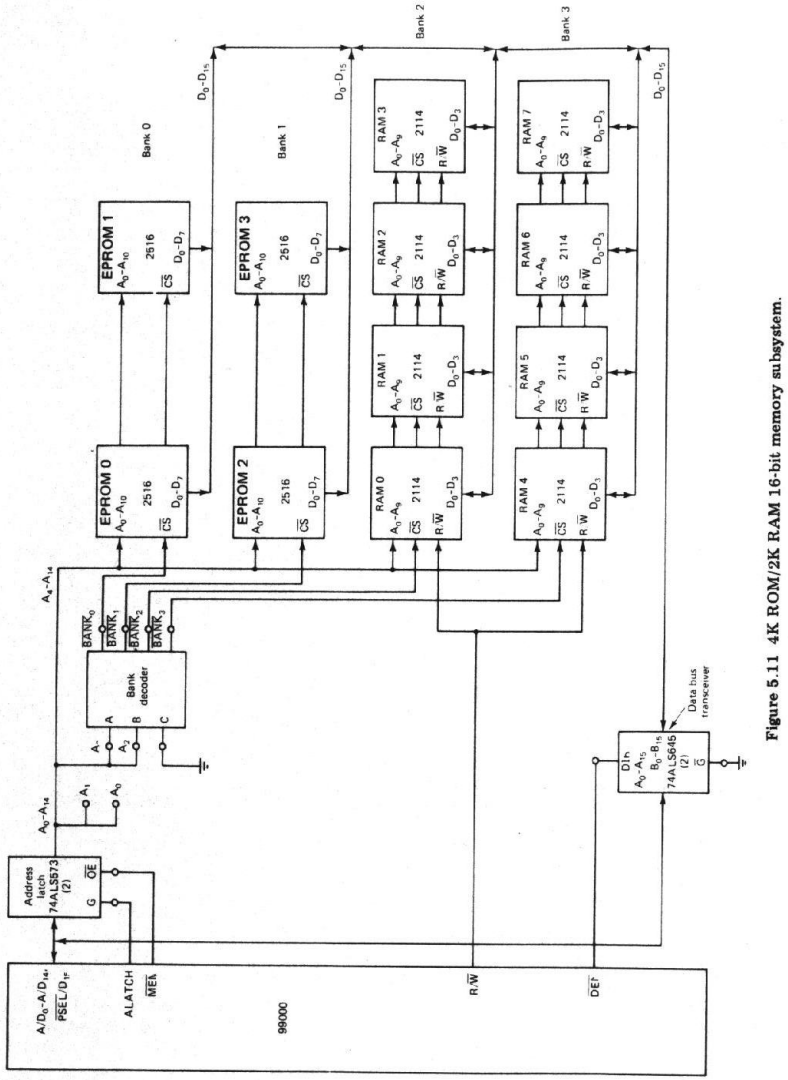


Figure 5.11 4K ROM/2K RAM 16-bit memory subsystem.

5.11 EPROM/STATIC RAM MEMORY SUBSYSTEM

The circuit illustrated in Fig. 5.11 shows a *static memory subsystem* that can be used for program and data storage in a 99000 microcomputer system. Looking at the memory array of the subsystem, we find that it contains two banks of TMS2516 EPROMs. These banks are labeled BANK 0 and BANK 1 in the circuit diagram. Each bank is formed with two 16K-bit EPROMs. They are organized 2K by 8 and connect together to give a total bank capacity of 2K 16-bit words. In this way, we get a total of 4K 16-bit words of EPROM for program storage memory.

The other two banks of the memory array contain static RAMs. Each bank contains four 1K by 4-bit RAM ICs connected to give 16-bit storage locations. This gives a total bank storage capacity of 1K 16-bit words and a total data storage memory capacity of 2K 16-bit words.

The address input on lines A_0 through A_{14} determines whether program or data memory is selected and the actual location that is to be accessed. Notice that bits A_4 through A_{14} of the address are directly applied to all memory devices in parallel. It is this part of the address that selects the actual storage location to be accessed. At the same time, bits A_2 and A_3 of the address are applied to inputs of a 74LS138 2-line-to-4-line decoder. Based on the binary code at these lines, one of the bank outputs, $\overline{BANK}_0$ through $\overline{BANK}_3$, is switched to logic 0. This 0 logic level is applied to the $\overline{CS}$ or $\overline{S}$ input of one of the banks of EPROMs or RAMs and enables it for operation.

When data are to be read from EPROM or RAM, the address selects the appropriate bank and storage location. In turn, the enabled memory devices output a word of data at D_0 through D_{15}.

When accessing the RAM part of the memory subsystem, $\overline{WE}$ signals whether data are to be read from or written into the addressed storage location. For a write operation, data are applied to the data input of the TMS2114 RAMs. When $\overline{WE}$ switches to logic 0, the data on the bus are written into the addressed locations of the enabled bank of RAMs.

5.12 EXTENDING THE ADDRESS SPACE OF THE 99000

Today, microcomputer systems are being employed in more complex applications and with increased use of high-level languages such as BASIC and PASCAL. Both of these conditions provide requirements for support of a large memory subsystem. At the same time, the dramatic decrease in the cost of semiconductor memory has made the use of large memory subsystems more practical.

The 99000 with its 15-bit address bus is limited to direct addressing of a 32K-word (64K-byte) memory subsystem. This is called its *logical address*

space. As indicated earlier, the $\overline{PSEL}$ signal can be used as an additional address bit to extend the logical address space of the 99000 in a paged mode to two 64K-byte pages, or 128K bytes.

Figure 5.12 shows how the extended address feature of the 99000 can be used to implement a 64K-word paged memory subsystem. Notice that the memory subsystem is formed from two separate 32K-word memory banks. These sections are called PAGE 0 and PAGE 1.

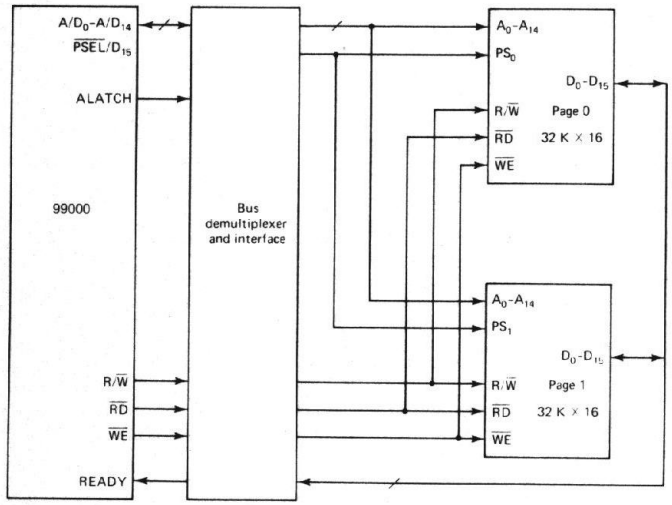


Figure 5.12 64K-word paged memory subsystem.

Remember that status bit ST_8 can be used as a sixteenth address bit. The logic level of ST_8 is complemented and multiplexed out at pin 31 of the 99000's package. This signal, $\overline{PSEL}$, together with A_0 through A_{14}, gives a 16-bit external address. This extended address permits the 99000 to access two independent 32K-word pages of memory, for a total of 64K words.

In the circuit diagram, $\overline{PSEL}$ is passed to the memory system together with A_0 through A_{14}. However, it is applied to page select inputs PS_0 and PS_1. For instance, when status bit ST_8 is set to logic 1, the $\overline{PSEL}$ output is

logic 0 and PAGE 0 of the memory system is selected. Therefore, only the memory devices in PAGE 0 of the memory subsystem are enabled for operation and data are read from or written into the addressed storage location.

Through software the value of ST_8 can be changed to logic 0. Now $\overline{PSEL}$ is logic 1 and storage locations on PAGE 1 of the memory subsystem are accessed instead of those on PAGE 0.

A number of different methods can be used to extend further the size of the 99000's address space. For instance, additional external decoding circuitry could be included to identify memory bus status codes that represent instruction acquisitions from those for data acquisitions. The outputs of this decoder are signals that indicate when accesses of the program storage memory and data storage memory occur. These signals can be used to enable an independent *128K-byte program storage memory segment* and an independent *128K-byte data storage memory segment*. In this way, the logical address space has been increased in a *segmented* fashion to 256K bytes.

Another way to expand the memory system of the 99000 is to use output ports as enable signals for 64K-byte pages of memory. If this is done, an output instruction must be executed to select one of a number of 64K-byte pages of memory for operation. This is another way of extending memory in a paged mode.

A technique known as *overlaying* is a fourth method of extending memory. Using this technique, the memory system is extended by use of a mass storage device such as a floppy disk. That is, the 64K-byte address space of the system is divided into segments called *overlays*. These overlay areas are not dedicated to one function; instead, they are loaded with different programs and data, depending on which task the microcomputer is performing. If the overlay contains data that are modified during program execution, it must be updated on the mass storage media at completion of the task. The disadvantage of this method lies in its degradation of system performance due to the overhead time required for loading from disk and returning modified data back to the disk.

A fifth choice is to extend the address reach of the 99000 microprocessor in a *mapped fashion*. This can be done by adding a *memory mapper* device to the memory subsystem interface.

Figure 5.13 illustrates the concept behind memory mapping. Notice that 11 of the 99000's 15 address lines are passed directly to the memory subsystem. The other four address lines are used as inputs to the mapper circuit. Here the 4-bit code gets translated into a 12-bit address extension. This address extension is output by the memory mapper and combined with the other 11 address bits to give an extended 23-bit address at the memory subsystem. The extended address is known as a *physical address* and it permits a *physical memory system* address space as large as 8 megabytes.

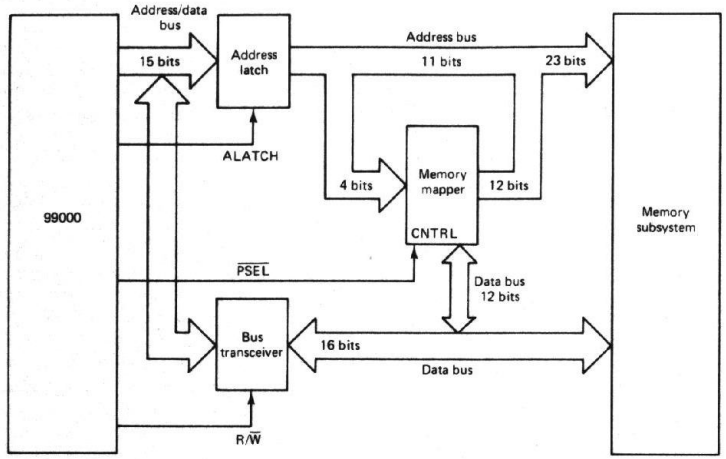


Figure 5.13 Memory mapped address reach expansion.

5.13 99000 MEMORY SUBSYSTEM WITH ERROR DETECTION AND CORRECTION

With the use of 64K-byte and larger memory subsystems becoming popular in microcomputer systems, the need has arisen for automatic error detection and correction capabilities. For instance, the occurrence of a *single-bit* hardware error in a large dynamic RAM subsystem could lock up the microcomputer system in an inoperable state.

Both hard and soft single-bit errors can be detected and corrected by adding a small amount of circuitry to the memory subsystem. Figure 5.14 shows such a configuration. Notice that an *error detector* and *corrector* (EDAC) *section* has been included in the memory interface. This results in improved accuracy for memory data transfers and increased overall system reliability.

The ability to perform error detection and correction has become more important with the widespread use of high-capacity dynamic RAM ICs such as 64K-bit devices. This is because a small number of *soft errors* are being experienced due to *alpha particles*. These soft errors are typically of the single-bit type.

Looking at Fig. 5.14, we see that data read from or written to memory

are also applied to the EDAC section. During a write operation, the data word output by the 99000 on the data bus is first processed by the EDAC. The EDAC generates and outputs a *check code*. This check word is stored together with the data word in memory. In the circuit diagram, we have shown that the 16-bit data bus of the 99000 has been extended to 22 bits. This corresponds to the addition of a 6-bit check word.

In this way, we see that the use of EDAC does require some memory overhead. In fact, if 64K dynamic RAMs are used, six more ICs must be added to the 64K-word subsystem to extend its word length from 16 bits to 22 bits.

When data are read back from memory, the data words are again checked by the EDAC circuit. The new check word that is generated is compared to the check word read from memory. If a match occurs, the EDAC operation is transparent to the microprocessor. That is, the read cycle is continued through to completion.

However, if a single- or double-bit error is detected, the EDAC unit signals the microprocessor of this condition. For a single-bit error, the micro-

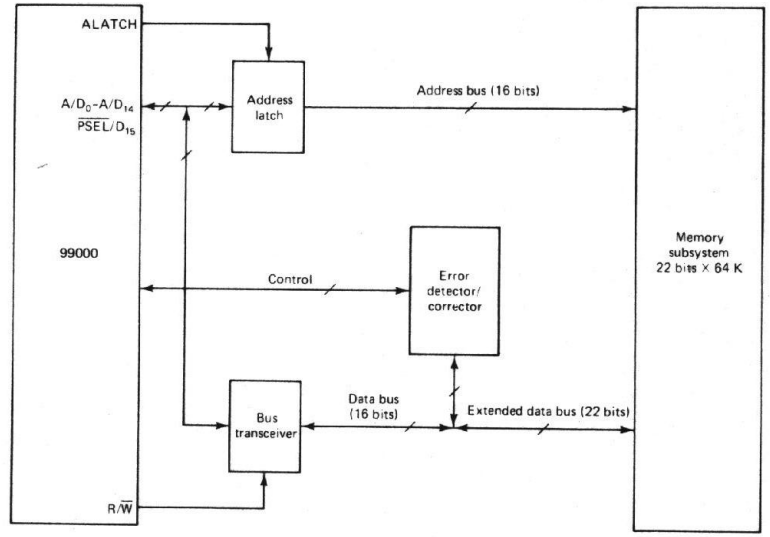


Figure 5.14 Memory subsystem with error detection and correction.

processor can set control signals such that the EDAC circuit automatically corrects the error. Then the corrected data are put on the data bus and read by the microprocessor.

Single-bit errors can be automatically corrected by the EDAC, but not *multibit errors*. Instead, the EDAC sends another flag signal to the microprocessor to identify the occurrence of the multibit error. In this way, the system can provide some form of system response to the incorrectly read data.

5.14 CACHE MEMORY FOR THE 99000 SYSTEM

Memory subsystems that are made with high capacity but relatively slow dynamic RAMs, such as the TMS4164, degrade the performance of the 99000 microcomputer system. Even though these dynamic RAMs are available with access times as short as 120 ns, they are still too slow to work in a 99000 system that is running without wait states. If wait states are introduced, the microprocessor is slowed down to work with the memory, but overall system performance is decreased.

A system with *cache memory* such as that shown in Fig. 5.15 provides a means for improving overall microcomputer system performance even when slow memory devices are in use. In a system with cache, a second, smaller but very fast memory section is added for use together with the large, slower main memory section. This small memory section, known as the *cache*, typically contains several thousand words of RAM and is made with high-speed static RAMs rather than dynamic RAMs. For this reason, cache can be accessed without wait states, whereas accesses of main memory require wait states.

The concept behind cache is that it can store data used frequently. When an address of a storage location to be read is put on the system bus, the *cache tag* circuit determines whether or not the data to be accessed reside in both main memory and the cache memory. If they do, the memory cycle is considered a *"hit"* condition. In turn, the cache tag signals the *cache controller* that a hit has occurred. The cache controller inhibits access of the main memory and initiates access of the corresponding data in cache.

In this way, we see that the use of cache reduces the number of accesses made from the slower main memory. This results in a level of system performance that approaches that of a system operating with main memory that requires no wait states.

If the address output to the main memory subsystem does not correspond to data that are already cached, the cache tag signals this condition to the controller. This represents a *"miss"* and the cache controller causes the data to be read from main system memory instead of from cache. For a read

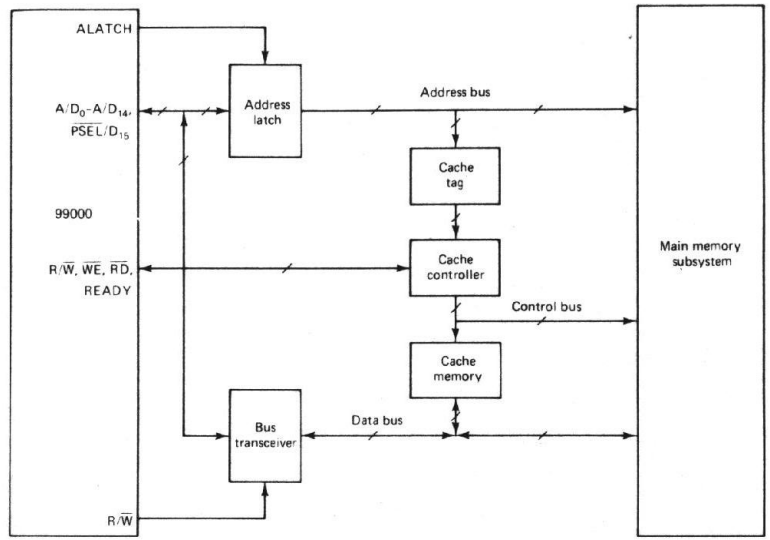


Figure 5.15 99000 memory subsystem with cache.

operation of an address that represents data not in cache, the data are first read from main memory and then written into a corresponding location in cache. Notice that all data that reside in cache also reside in main memory.

Two types of write operation are also performed. If the address to be written into represents data that are already cached, the write is performed to storage locations in both cache and main memory. This feature of a cache is known as *automatic write-through*. On the other hand, if the write is to a location that is not yet cached, a write is initiated such that just main memory is updated.

ASSIGNMENT

Section 5.2

1. What is the function of the address bus relative to memory operation?
2. What is the function of the data bus relative to memory operation?

Section 5.3

3. Which bit of the 99000's address is maintained internal for byte selection?
4. How many unique byte addresses can be accessed by the 99000's extended address bus?

Section 5.4

5. Which data formats can be directly processed by the 99000?
6. How would the data >12345678 be stored starting at address >A000 in memory?
7. Over which data bus lines would data byte >FF be carried if it is written to address >B001 in memory?

Section 5.5

8. What part of the first page of the 99000's memory address space should be used for program storage?
9. How many bytes from the first page of the 99000's memory are reserved for storage of interrupt and extended operation vectors?

Section 5.6

10. What bus status code is output by the 99000 when it reads an interrupt vector from memory?
11. Give an overview of the function of each of the signals that follow relative to memory operation: ALATCH, $\overline{WE}$, and $\overline{RD}$.

Section 5.7

12. Give five examples of operations that would require the 99000 to perform a read operation from memory.
13. Describe the sequence of bus activity that would occur as the 99000 reads a long-word operand from memory.

Section 5.8

14. Describe the sequence of bus activity that would occur as the 99000 writes a byte operand into memory.

Section 5.9

15. What function does the READY signal serve in the 99000 microcomputer system?
16. If the 99000 is run by a 20-MHz crystal, what is the duration of each bus cycle if READY is fixed at the 1 logic level?

Section 5.10

17. Explain the operation of the circuit shown in Fig. 5.10 as a word-wide write cycle is performed to the memory subsystem.

Section 5.11

18. If the address on the bus shown in Fig. 5.11 is >10FE, which bank of memory devices is enabled and which address in the bank is accessed?

Section 5.12

19. Assume that the 99000 system of Fig. 5.12 is currently accessing PAGE 0 of memory. Write a sequence of instructions that will modify the status such that PAGE 1 can be accessed.
20. Name two methods of expanding the address reach of the 99000 beyond 128K bytes.
21. Draw a simple logic circuit that can decode the bus status to produce a select signal for enabling program or data memory.
22. In the circuit of Fig. 5.13, the address at the outputs of the address latch is >ABCD. If the 12-bit output produced by the memory mapper is >F00, what location is accessed in the memory subsystem?

Section 5.13

23. What is the need for EDAC in a microcomputer system?
24. Give an overview of how an EDAC supports single-bit error detection and correction.

Section 5.14

25. Why would cache memory be used in a 99000 microcomputer system?
26. List all the differences between cache and main memory.
27. What is meant by "hit condition" when discussing the operation of a cache?

6

INPUT/OUTPUT INTERFACE OF THE 99000

6.1 INTRODUCTION

In Chapter 5 we studied the memory interface of the 99000. Here we continue with another important interface, the *input/output interface*. The following topics are covered in the chapter:

1. Communications register unit
2. I/O instructions
3. The base address and I/O address space
4. Bit-serial I/O operation and bus cycle
5. External serial I/O interface circuitry
6. Parallel I/O operation and bus cycle
7. External parallel I/O interface circuitry
8. Wait states in the I/O bus cycle

6.2 COMMUNICATIONS REGISTER UNIT

The *communications register unit* (CRU) is the main I/O interface of the 99000 microprocessor. Other popular 16-bit microprocessors do not have an independent I/O interface such as that of the 99000. Because of this CRU, the 99000 provides a very flexible I/O mechanism that has the ability

to do single-bit or multibit I/O operations through a bit-serial I/O interface and byte or word transfers through a parallel I/O interface.

Serial Interface Block Diagram

The *serial I/O interface* of the 99000 is illustrated in Fig. 6.1. Here we see that the address bus A_0 through A_{14}, memory control signals ALATCH, $R/\overline{W}$, $\overline{RD}$, and $\overline{MEM}$, and bus status lines BST_1 through BST_3 are common to both the memory and serial I/O interface. The address output on these lines are used to select the I/O port for which data are input or output. Notice that the I/O interface also contains three new signals: IN, OUT, and $\overline{IOCLK}$. Data are output in *bit-serial form* on the OUT line synchronously with clock pulses at the $\overline{IOCLK}$ output. On the other hand, bit-serial data are input on the IN line. This interface provides for easy control of *single-bit I/O ports*.

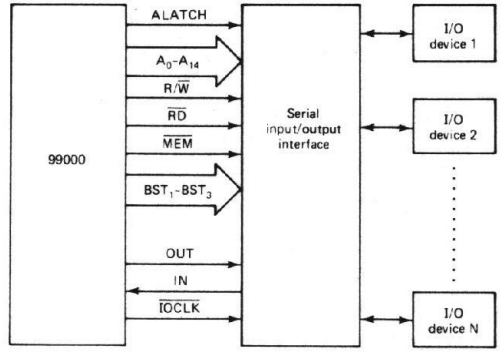


Figure 6.1 Serial input/output interface block diagram.

Parallel Interface Block Diagram

The *parallel I/O interface*, which is shown in Fig. 6.2, is similar to the serial interface except that it uses the data bus line, D_0 through D_{15}, for input and output of data instead of IN and OUT. Byte transfers take place over the most significant eight bus lines. D_0 through D_7, and words are carried over the complete bus, D_0 through D_{15}. The parallel I/O capability provides for easy interface to *LSI peripherals* that operate over an 8-bit parallel bus.

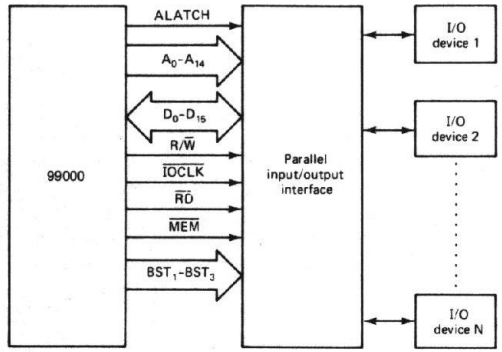


Figure 6.2 Parallel input/output interface block diagram.

6.3 INPUT/OUTPUT INSTRUCTIONS

Five instructions are provided in the instruction set of the 99000 specifically to perform input/output operations through the serial I/O interface. They are listed together with a summary of their operation in Fig. 6.3.

Of these instructions, three are for *single-bit I/O operations:* the *set bit one* (SBO), the *set bit zero* (SBZ), and *test bit* (TB). The first two, SBO and SBZ, perform output operations and are used to set the logic level at a specific output port to the 1 and 0 logic levels, respectively. The third instruction, TB, is used to input the logic level from an input port. The 0 or 1 level that is input is stored in the equal bit, ST_2, of the status register.

Multibit serial I/O operations are performed using the *load communications register* (LDCR) *instruction* and *store communications register* (STCR) *instruction.* LDCR is used for multibit output operations and STCR for multibit input operations. With these instructions, we can input from or output to from 2 to 15 I/O ports at consecutive addresses. The number of bits to be input or output are specified as a 4-bit count which is part of the instruction.

As shown in Fig. 6.3, only the LDCR and STCR instructions are applicable to the parallel I/O interface. In this case, the count specified with the instruction determines whether a word or byte transfer is to take place over the data bus instead of the number of bits to be input or output.

There are just four counts that are allowed for use with parallel I/O: 0010, 0011, 1010, and 1011. If the count specified with the instruction is either 1010 or 1011, a word transfer occurs. On the other hand, for a byte

Instruction	Meaning	Format	Serial I/O explanation	Parallel I/O explanation
SBO	Set bit one	SBO I	Sets the I/O bit at the specified displacement from the base address to logic 1	None
SBZ	Set bit zero	SBZ I	Sets the I/O bit at the specified displacement from the base address to logic 0	None
TB	Test bit	TB I	Sets the equal bit of the status register equal to the logic level at the I/O bit at the specified displacement from the base address	None
LDCR	Load communication register	LDCR S,I	Loads the logic levels specified in the instruction into the 2 to 16 consecutive I/O bits starting with the bit located at the base address	Outputs a word or byte of data
STCR	Store communication register	STCR S,I	Stores the logic levels at 2 to 16 consecutive I/O bits starting with the bit located at the base address into the storage location specified in the instruction	Inputs a word or byte of data

Figure 6.3 Input/output instructions.

transfer to take place, the count must be either 0010 or 0011. In this way, we see that logic 1 in the MSB location selects word transfers and 0 in this position selects byte transfers.

Moreover, an autoincrement mode is selected for the I/O address when 0011 or 1011 is selected as the count. It is the logic 1 in the LSB of the count that selects *autoincrement mode* of operation. When autoincrement operation is selected, the address that points to the I/O port is automatically incremented by 2 during execution of the instruction. In this way, it will point to the next I/O port at completion of the current I/O operation.

6.4 THE BASE ADDRESS AND I/O ADDRESS SPACE

The port accessed for input or output of data is selected by an *I/O address* that is output on the system address bus A_0 through A_{14}. This 15-bit address provides $2^{15} = 32{,}768$ (32K) independent I/O addresses. The I/O ports associated with these addresses can be bit wide, byte wide, or word wide.

Figure 6.4 shows the I/O address space of the 99000. This I/O address space is totally independent of the system's memory address space. Independence is achieved through the use of the bus status code. During the complete I/O bus cycle, the $\overline{MEM}$ output of the 99000 is logic 1. This indicates

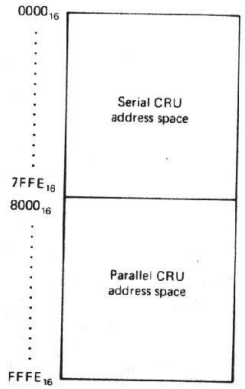


Figure 6.4 I/O address space.

that a memory operation is not in progress. The rest of the bus status lines are $BST_1BST_2BST_3 = 011$ and indicate that a I/O operation is in progress. These signals must be used to enable external circuitry to decode the address output on the system bus.

Looking at Fig. 6.4, we see that the MSB of the I/O address indicates whether a serial or parallel operation is in progress. Serial I/O operations occur when A_0 is logic 0 and parallel operations occur when A_0 is logic 1. In this way, we find that the lower 16,384 (16K) of addresses, 0000_{16} through $7FFE_{16}$, in the I/O address space are dedicated to *serial accessible I/O ports* and the higher 16K of addresses, 8000_{16} through $FFFE_{16}$, to *parallel accessible I/O ports.*

I/O Base Address

During execution of an I/O instruction, register R_{12} of the current workspace must contain an *I/O address pointer.* This is known as the *base address* and must be loaded under software control prior to execution of the I/O instruction. The 16-bit value loaded into R_{12} is called the *I/O software address.* It differs in value from the address output on A_0 through A_{14} during an I/O bus cycle. This is because the LSB A_{15} is not output or used to determine which I/O port is to be accessed. The value output on the bus is known as the *I/O hardware address.* It is the actual address of the I/O port to be accessed.

Figure 6.5(a) shows the relationship between the value of the software and hardware addresses. Here we see that the software address is equal to the hardware address shifted left by 1 bit. For this reason, the value of the software address is twice that of the hardware address. For example, if the hardware address is to be $1000_{16} = 0010000000000000_2$, R_{12} must be loaded with $2000_{16} = 0010000000000000_2$.

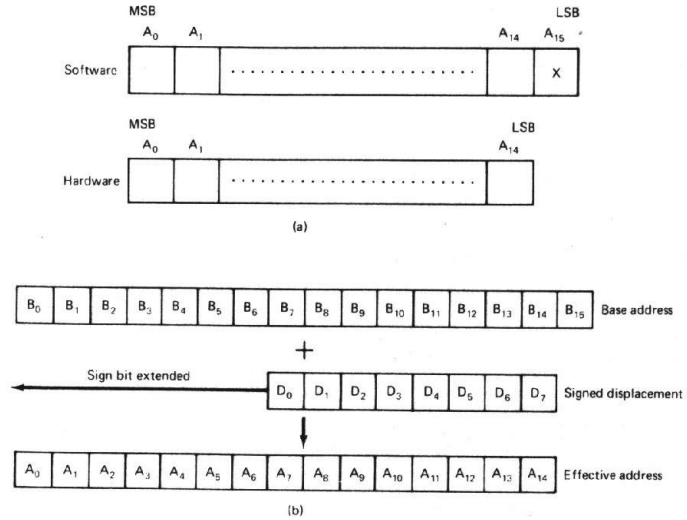


Figure 6.5(a) Software and hardware address words; **(b)** generating the software address for a single-bit I/O operation.

Base Address for a Single-Bit I/O Instruction

In the case of the single-bit serial I/O instructions, the base address identifies the I/O port for which the input or output operation is to occur. As shown in Fig. 6.3, an 8-bit signed displacement labeled I can be defined as part of the SBO, SBZ, or TB instruction. This allows the port to which the I/O operation takes place to be offset by +127 to −128 from the port pointed to by the base address. The effective address of the I/O port to be accessed is obtained by adding the displacement to the base address in R_{12}. This is done as shown in Fig. 6.5(b).

For example, let the displacement equal $01111111_2 = 7F_{16}$ (+127) and the base address equal $2000_{16} = 0010000000000000_2$. Adding, we obtain an effective software address of $0010000011111110_2 = 20FE_{16}$. The corresponding hardware address is $001000001111111_2 = 107F_{16}$.

Example 6.1

What is the displaced software and hardware address if the displacement specified with a TB instruction is to be −127 and register R_{12} holds 2000_{16}? The TB instruction always contains the 2's complement of the displacement.

Solution: Expressing the displacement in binary form and converting to a signed number gives

$$-127 = 11111110_2$$

The sign bit is extended for the displacement to give the 16-bit word

$$1111111111111110_2$$

The base address in binary form is

$$2000_{16} = 0010000000000000_2$$

Adding the displacement to the base address, we get

$$0010000000000000_2 + 1111111111111110_2 = 0001111111111110_2$$
$$= 1FFE_{16}$$

This is the software address and its equivalent hardware address is

$$0001111111111111_2 = 0FFF_{16}$$

Base Address for a Multibit I/O Instruction

For multibit I/O instructions, the base address in R_{12} acts differently depending on whether a serial or parallel data transfer is taking place. For parallel transfers, its equivalent hardware address points to the byte-wide or word-wide I/O port.

On the other hand, for multibit serial transfers, the base address points to the port for which the first bit of data will be input or output. In this case, the address is automatically incremented after each bit is input or output such that it always points to the next consecutive I/O port.

6.5 BIT-SERIAL I/O OPERATION AND BUS CYCLE

Now that we have introduced the communications register unit interface, I/O instructions, and I/O address space, let us continue by tracing the sequence of events that occur during the *single-bit* and *multibit I/O bus cycles.*

Single-Bit I/O Operation

Figure 6.6 shows the timing sequence for a single-bit output bus cycle such as those performed by the SBO or SBZ instructions. This diagram represents an I/O bus cycle with no wait states and has a duration of two CLKOUT

cycles. The sequence begins with ALATCH switching to logic 1, $\overline{MEM}$ switching to logic 1, and a I/O bus status code (011) output on the BST lines. Next, the address of the I/O port that is to be accessed is put on bus

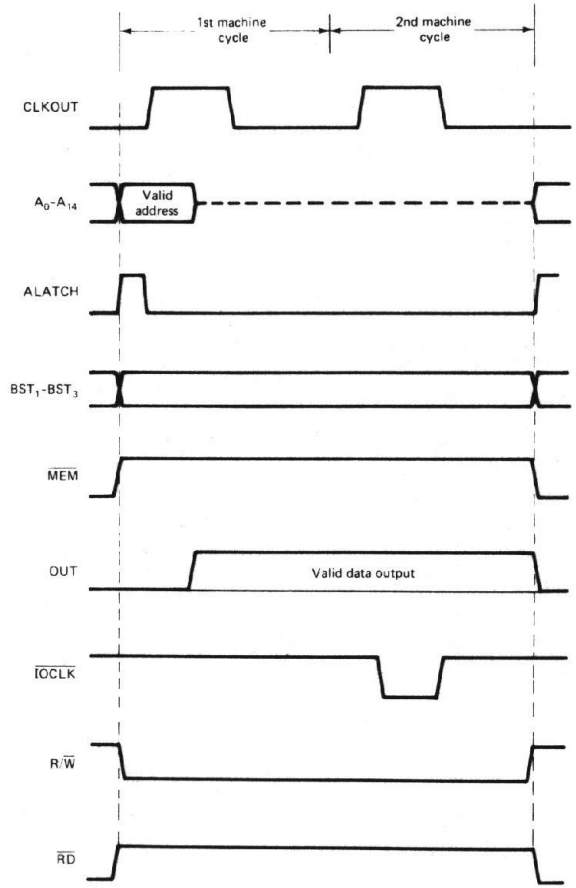


Figure 6.6 Single-bit serial output bus cycle timing.

lines A_0 through A_{14}. As ALATCH switches back to logic 0, this address must be latched into the external system bus demultiplexing circuitry.

After this, the 99000 sets OUT to the logic level that is to be output. This output remains valid throughout the rest of this and the next CLKOUT cycle. $\overline{IOCLK}$ remains at the 1 logic level until the second CLKOUT cycle of

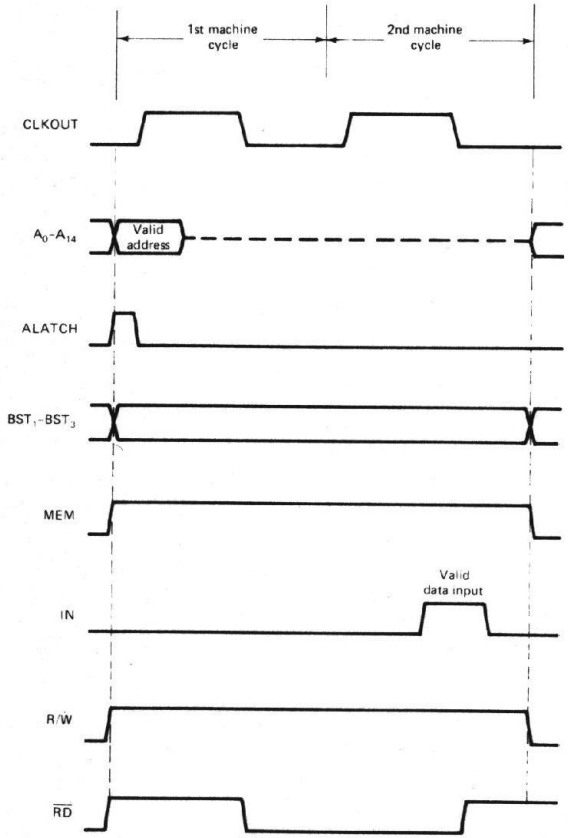


Figure 6.7 Single-bit serial input bus cycle timing.

the output bus cycle. Then it is switched to logic 0. This pulse should be used to load the data at OUT synchronously into the output port.

The single-bit serial input cycle that occurs during the execution of the TB instruction is similar to that just described for the SBO or SBZ instruction. Figure 6.7 is a timing diagram for this type of I/O bus cycle. Notice that this time, data are read in from the input port on IN. The 99000 samples the logic level at IN synchronously with the 1-to-0 edge of the second CLKOUT pulse. The $\overline{RD}$ signal is logic 0 at this time and can be used together with the IO bus status code to enable the 0 or 1 logic level from the port selected by the address onto the IN line. The logic level read at IN is loaded into the equal bit of the status register.

Multibit Serial I/O Operation

The timing diagrams of the multibit serial input and output operations performed by the STCR and LDCR instructions, respectively, are similar to those given in Figs. 6.6 and 6.7. However, the bus cycle repeats for each bit of data that is to be input or output. During the execution of these instructions, one bit of data is output every other clock cycle.

6.6 EXTERNAL SERIAL I/O INTERFACE CIRCUITRY

Let us now look at the circuitry that is required external to the 99000 to implement input and output ports when using the serial I/O interface.

64 Input Serial I/O Interface

The diagram in Fig. 6.8 shows how eight 74LS251 8-line-to-1-line multiplexer ICs can be used to implement a *64-input serial I/O interface*. Notice that each device forms an 8-bit input port. These ports are labeled PORT 0 through PORT 7. The eight devices provide 64 independent input lines. They are labeled I_0 through I_{63}. On the other hand, the OUT lines of the eight ports are all tied in parallel and supplied to the IN input of the 99000.

The 99000 selects data from the appropriate input line with the I/O address it outputs on A_{0L} through A_{14L}. This is the demultiplexed system address bus. Bit A_{0L} of this address is logic 0 during all serial I/O operations. For this reason, it is applied to the G_{2A} and G_{2B} inputs of the 74LS138 I/O address decoder. At the same time, the signal $\overline{IO}$ is inverted and applied to the G_1 input of the decoder. This input is always 0 when an I/O address is on the bus. These signal levels enable the decoder for operation.

Address lines A_{1L}, A_{2L}, and A_{3L} apply a 3-bit code to the CBA inputs of the address decoder. When the decoder is enabled, the output, 0 through 7, corresponding to this 3-bit input code is switched to logic 0. Outputs 0

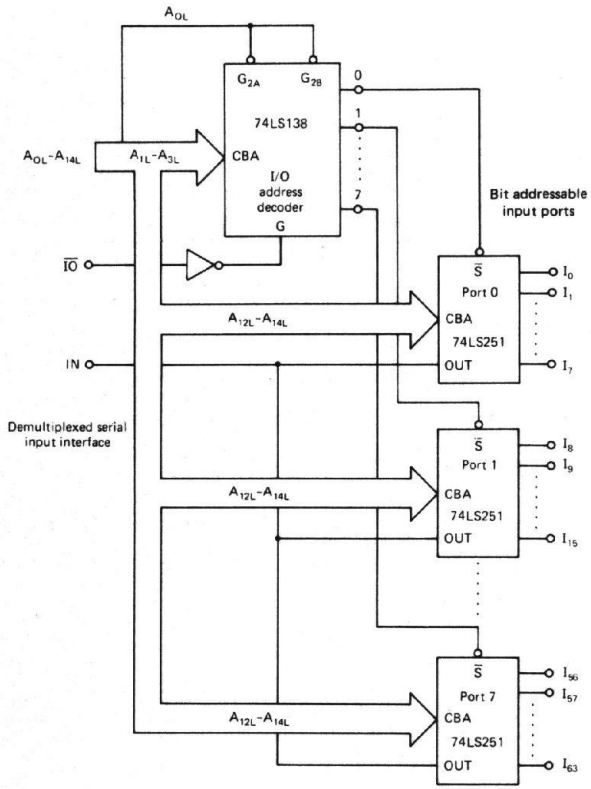


Figure 6.8 64 bit-addressable input ports.

through 7 are used as strobe inputs for the 74LS251 devices for input ports 0 through 7, respectively. In this way, one of the eight input multiplexers is selected for operation.

For example, let us assume that address bits A_{1L} through A_{3L} are equal to 001. This makes output 1 of the decoder switch to logic 0. It is applied to the $\overline{S}$ input of the PORT 1 multiplexer and enables it for operation.

Three other address lines, $A_{12}L$, $A_{13}L$, and $A_{14}L$, are applied in parallel to the C, B, and A inputs, respectively, of all 74LS251 port multiplexers. The code on these lines selects data from one of the eight inputs of the enabled multiplexer and passes it out onto the IN line. For instance, assuming that the PORT 1 multiplexer has been selected and that $A_{12L}A_{13L}A_{14L}$ = 111, the logic level at input I_{15} is passed to IN for input to the 99000.

Example 6.2

The I/O address output by the 99000 to the circuit shown in Fig. 6.8 is 7000_{16}. From which input line is data multiplexed to IN?

Solution: Let us begin by expressing the address in binary form. This gives

$$7000_{16} = 0111000000000000_2$$

The MSB of the address A_0L is logic 0. This identifies that a serial I/O operation is in progress and together with $\overline{IO}$ equal to logic 0 enables the 74LS138 address decoder.

$A_0L = 0$ enables serial I/O address decoder

$IO = 0$

The next 3 bits, $A_1LA_2LA_3L$, equal 111 and are applied to the CBA inputs of the decoder. This code causes output 7 to switch to logic 0 and enables the PORT 7 multiplexer for operation.

$A_1LA_2LA_3L = 111$

$7 = 0$ enables PORT 7 multiplexer

The three LSBs of the address, $A_{12L}A_{13L}A_{14L}$, equal 000. This is applied to all multiplexers in parallel; however, just the PORT 7 multiplexer is enabled. Therefore, the logic level at input I_{56} is multiplexed onto IN for return to the 99000.

64 Output Serial I/O Interface

A circuit that provides *64 bit-addressable output ports* from the serial I/O interface is shown in Fig. 6.9. Here we find that eight 74LS259 8-bit addressable latch ICs are used to produce eight output ports. These ports are labeled PORT 0 through PORT 7. The individual output lines of these ports are labeled O_0 through O_{63}.

Data output on OUT are passed to the D_{IN} input of all ports in parallel. As for the input circuit of Fig. 6.8, the demultiplexed system address output on lines A_0L through $A_{14}L$ selects the port to which data are to be output. During serial I/O operations, A_0L provides logic 0 at G_{2A} and G_{2B} of the 74LS138 I/O address decoder. Also, $\overline{IO}$ and $\overline{IOCLK}$ are gated together to supply logic 1 at the G_1 input of the decoder. These signals enable it for operation.

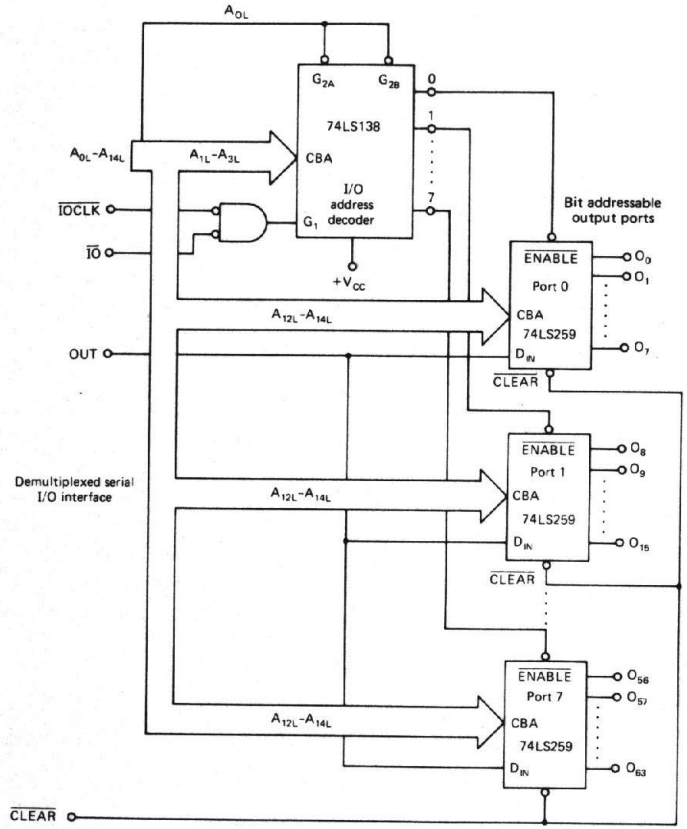


Figure 6.9 64 bit-addressable output ports.

The code at address lines A_{1L} through A_{3L} identifies the port to be enabled. But this time $\overline{IOCLK}$ is used to generate the G_1 enable input of the decoder. Therefore, the output selected by the code $A_{1L}A_{2L}A_{3L}$ switches to logic 0 during the pulse at $\overline{IOCLK}$.

At the same time, bits A_{12L} through A_{14L} of the address are input to

all 74LS259 devices in parallel. This code selects the output line on the enabled output port at which data are to be latched. In this way, the logic level output on OUT is latched at the output selected by $A_{12L}A_{13L}A_{14L}$.

Example 6.3

The address output by the 99000 to the circuit shown in Fig. 6.9 during execution of a SBO instruction is 0007_{16}. Which output is set to logic 1?

Solution: We begin by converting the address to binary form. This gives

$$0007_{16} = 0000000000000111_2$$

The logic 0 at A_{0L} is one enable input to the address decoder and when both $\overline{IO}$ and $\overline{IOCLK}$ are equal to 0, the other enable input is supplied.

$A_{0L} = 0$ enable inputs to address decoder

$\overline{IO} = 0$

$\overline{IOCLK} = 0$

$A_{1L}A_{2L}A_{3L} = 000$ identifies output PORT 0 and when $\overline{IOCLK}$ switches to its active logic level, the 0 output of the decoder switches to logic 0. This enables PORT 0.

$A_{1L}A_{2L}A_{3L} = 000$, $\overline{IOCLK} = 0$ enables PORT 0

$A_{12L}A_{13L}A_{14L} = 111$ selects output O_7 at PORT 0.

$A_{12L}A_{13L}A_{14L} = 111$ selects output O_7

Logic 1 is output on OUT by the 99000; therefore, output O_7 is set to logic 1 synchronously with the pulse at $\overline{IOCLK}$.

6.7 PARALLEL I/O OPERATION AND BUS CYCLE

Looking at Fig. 6.10, we find that the sequence of events that take place during a *parallel output bus cycle* are similar to those in a single-bit-serial output bus cycle. Just like for the single-bit output cycle, at the beginning of the first clock cycle, ALATCH switches from 0 to 1, $\overline{MEM}$ switches to logic 1, R/$\overline{W}$ switches to logic 0, an IO bus status code is output on BST_1 through BST_3, and the address of the I/O port is put on the address bus. The address must be latched in external hardware synchronous with the 1-to-0 transition of ALATCH. However, in this case, the 1 at A_0 selects the parallel mode of I/O operation and sets up the 99000 internally for a parallel data transfer over the data bus instead of a serial transfer at OUT.

During the second clock cycle, the data output operation is performed as a parallel word transfer over bus lines D_0 through D_{15} or byte transfer over D_0 through D_7. Remember that the count specified with the LDCR or STCR instruction determines whether a word or byte transfer takes place.

Note that the output data become available after the address is removed in machine cycle 1 and remains valid for the rest of the output bus cycle. Later in the second machine cycle, data can be written into the output port synchronously with a pulse at $\overline{\text{IOCLK}}$.

If the autoincrement mode is selected with the count, the base address

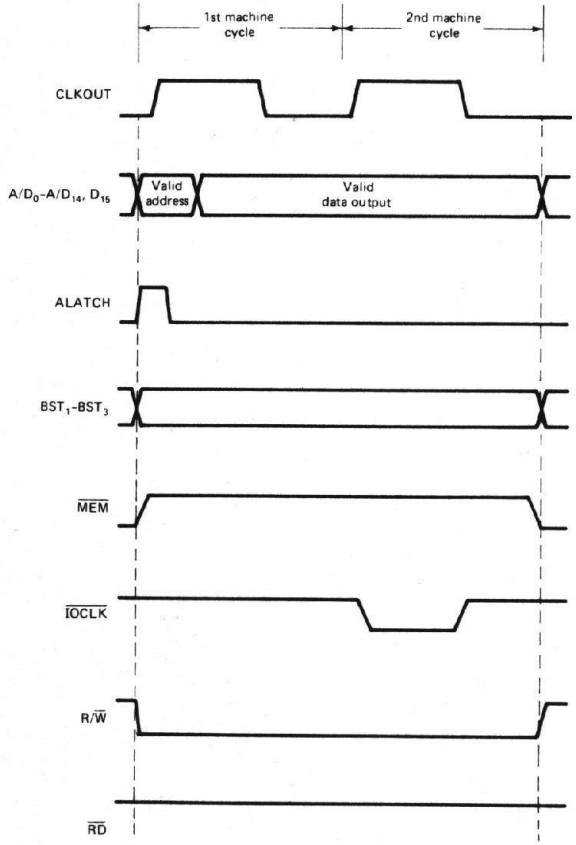


Figure 6.10 Parallel output bus cycle timing.

held in register R_{12} is automatically incremented by 2 at the end of execution of the instruction. Looping on this instruction provides a simple means for output of data to successive I/O ports at word addresses. An example is the byte-wide registers within an LSI peripheral.

6.8 EXTERNAL PARALLEL I/O INTERFACE CIRCUITRY

Now that we have described the parallel I/O bus cycle, let us demonstrate with circuitry how its signals can be used to implement parallel input and output ports.

Eight Byte-Wide Parallel Input Ports

Figure 6.11 shows how *eight byte-wide input ports* can be implemented in the 99000 system using the parallel I/O interface. This circuit contains one 74LS244 three-state buffer for each port. If an I/O operation is not in progress, the outputs of these buffers are all in the high-Z state and isolated from the data bus.

During a parallel STCR instruction, which is specified for a byte transfer, the I/O address that is latched onto the system bus is applied to the inputs of the 74LS138 I/O address decoder. For all parallel I/O operations, A_0 equals 1. This supplies the G_1 enable input to the address decoder. Moreover, during the complete parallel input bus cycle, the $\overline{\text{IO}}$ output of the bus status decoder is logic 0 and the $\overline{\text{RD}}$ output of the 99000 is switched to logic 0. This signals the input interface to put onto the bus the data that are to be input. In our circuit, these two signals are applied to the G_{2A} and G_{2B} inputs of the address decoder. Logic 0 at these two inputs together with logic 1 at G_1 enables the decoder for operation.

Since the decoder is enabled, its input code, $A_{12L}A_{13L}A_{14L}$, causes the corresponding output to switch to logic 0. For example, if the input code is 000, the 0 output of the decoder switches to logic 0. These outputs are applied to the $\overline{G}_1$ and $\overline{G}_2$ inputs of the 74LS244 input buffers. The buffer that corresponds to the decoder output that is logic 0 is enabled and the data from its input lines are passed onto data bus lines D_0 through D_7. The 99000 reads this byte of data off the bus and stores it in memory.

Example 6.4

If the I/O address output on the bus during a STCR instruction to the circuit shown in Fig. 6.11 is 4007_{16}, which port is enabled and which data byte is input?

Solution: Let us begin by expressing the address in binary form. This gives

$$4007_{16} = 100000000000111_2$$

The MSB of the address is $A_0 = 1$. Applying it to the G_1 input of the address decoder together with $\overline{\text{IO}} = 0$ at G_{2A} and $\overline{\text{RD}} = 0$ at G_{2B} enables the decoder for operation.

$A_0 = G_1 = 1$ enables address decoder

$\overline{IO} = G_{2A} = 0$

$\overline{RD} = G_{2B} = 0$

The three LSBs of the address are $A_{12L}A_{13L}A_{14L} = 111$ and cause output 7 of the decoder to switch to logic 0.

$A_{12L}A_{13L}A_{14L} = 111$ output 7 = 0

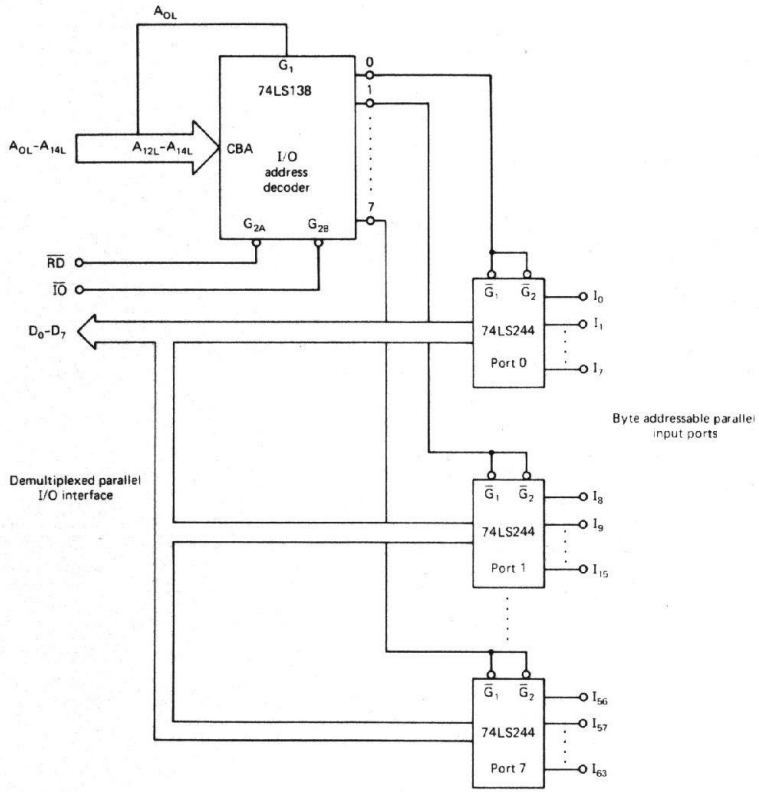


Figure 6.11 Eight byte-wide parallel input ports.

This enables PORT 7 and data bytes I_{56} through I_{63} are put on data bus lines D_0 through D_7, respectively, for input to the 99000.

Eight Byte-Wide Parallel Output Ports

The circuit illustrated in Fig. 6.12 shows how *eight byte-wide parallel output ports* can be constructed in the 99000 microcomputer system. Note that

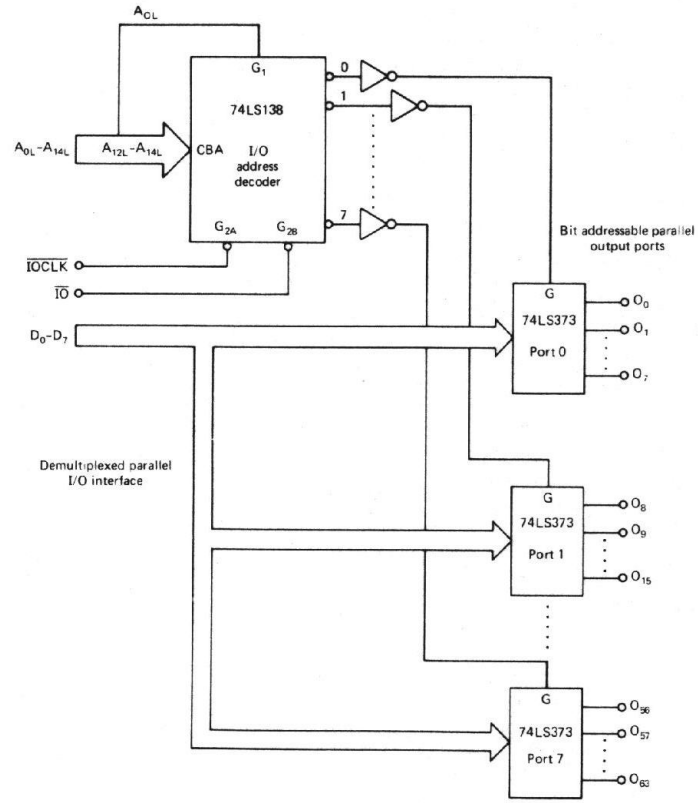


Figure 6.12 Eight byte-wide parallel output ports.

each port is implemented with a 74LS373 octal latch. The individual output lines are labeled O_0 through O_{63}.

For this circuit to operate, the 99000 outputs an I/O address on A_{0L} through A_{14L} due to the execution of a byte parallel LDCR instruction. It is this address that selects the output port. Next the 99000 outputs the byte of data on D_0 through D_7. These data represent the logic levels to which the lines of the output port are to be set and are applied to the inputs of all port latches in parallel. When the data are valid, a pulse is output on $\overline{IOCLK}$. This pulse is used to enable the IO address decoder and causes the decoder output selected by $A_{12L}A_{13L}A_{14L}$ to be pulsed to the 0 logic level. The pulse output by the decoder is inverted and then used to load the corresponding port latch from the data bus.

For example, the address $100000000000111_2 = 4007_{16}$ causes PORT 7 to be loaded from D_0 through D_7 and the byte of data appears at outputs O_{56} through O_{63}.

6.9 WAIT STATES IN THE I/O BUS CYCLE

For many of the slower LSI peripherals that are available today, the high-speed data transfer rates of the 99000's I/O interfaces may be too fast. For this reason, the memory bus cycle wait-state logic we discussed earlier also applies to its I/O interfaces. It can be used to introduce wait states into the serial or parallel input/output bus cycle. In this way, the effective data transfer rate can be slowed down.

The READY input is tested at a point in the I/O bus cycle just before the input or output data transfer takes place. For instance, if READY is at logic 0, wait states are inserted into the I/O bus cycle. Assuming that READY is switched back to 1 before the next clock cycle is completed, the current I/O bus cycle is extended by just one wait state. This increases the time that it takes for a data transfer to be performed from the duration of two clock cycles to three clock cycles.

ASSIGNMENT

Section 6.2

1. What is the function of the address bus relative to I/O operation?
2. Which line carries data during input operations performed through the serial I/O interface?
3. Which line carries data during output operations performed through the serial I/O interface?

Section 6.3

4. Write instructions that will do the following:
   (a) Load the base pointer register with the address >6000.
   (b) Set the seventh bit displaced in the positive direction relative to the current base pointer to logic 0.
   (c) Set the fifth bit displaced in the negative direction relative to the current base pointer to logic 1.
5. Write a sequence of instructions that first tests the logic level at the input port at I/O address >7000. The test should be repeated until the input line is read as logic 1. When this happens, a branch is to be initiated to a subroutine called INPUT.
6. Write a subroutine called INPUT that will input the contents of the eight consecutive input ports starting at I/O address >7002 and stores the byte at memory location >A000. Then it acknowledges the input of data by generating a short-duration pulse at the output port at I/O address >7100.

Section 6.4

7. Does the instruction sequence

```
LI    R12,>A000
LDCR  >2000,3
```

   perform a 3-bit serial output operation, byte-wide parallel output operation, or word-wide parallel output operation?
8. What are the software and hardware addresses for each of the output instructions in problem 4?

Section 6.5

9. What status code is output during the bus cycle of a TB instruction?

Section 6.6

10. A test bit instruction is executed to access the input port at address >1002 in Fig. 6.8. Which input is read, and where is its logic level stored?
11. Write an instruction sequence that when executed will generate a square-wave output signal at output O_8 of the circuit shown in Fig. 6.9.

Section 6.7

12. What is the advantage of using parallel I/O operations over serial I/O operations when the data to be input or output are either byte-wide or word-wide?

Section 6.8

13. What base address must be used with a STCR instruction to input the contents of PORT 1 in the circuit of Fig. 6.11?
14. Write a sequence of instructions that would input bytes of data from PORTS 0 through 7 and store them in consecutive memory locations starting at memory address >B000.

7

INTERRUPT INTERFACE OF THE 99000

7.1 INTRODUCTION

In Chapter 6 we examined in detail the I/O interface of the 99000. Here we will conclude our study with its interrupt interface. The topics that we cover are as follows:

1. Interrupts
2. External interrupt interface
3. Interrupt priority levels
4. External interrupt request
5. Priority encoder
6. Context switch sequence
7. Interrupt vectors
8. Interrupt mask
9. Reset and nonmaskable interrupt
10. External interrupt interface circuitry
11. Extended operation instructions
12. Internal interrupt functions
13. Illegal opcode detection and macroinstruction detection
14. Macrostore memory
15. Macrostore modes of operation
16. Entry and exit of macrostore

17. Privileged mode
18. Arithmetic fault detection

7.2 INTERRUPTS

Interrupts are used to initiate a change in program environment based on the occurrence of an event internal to the microprocessor or in external hardware. For instance, when an interrupt signal occurs, indicating that an external device, such as a printer, requires service, the 99000 must suspend what it is doing in the main part of the program and pass control to a special routine that performs the function required by the device. This segment of program is known as the interrupt service routine. At completion of this routine, the processor must return control to the main program and execution picks up where it left off earlier.

The 99000 provides capability for 17 external interrupts, 16 software interrupts, and three internal interrupts. Two of the external interrupts are dedicated to special functions known as reset and the nonmaskable interrupt. But the functions of the other 15 external interrupts can be defined by the user. Moreover, the functions of all 16 software interrupts are user definable. On the other hand, the three internal interrupts all have dedicated functions. These functions are *illegal opcode detection*, *privileged mode violation detection*, and *arithmetic overflow detection*.

7.3 EXTERNAL INTERRUPT INTERFACE

Figure 7.1 shows the *external interrupt interface* of the 99000. The 15 user-definable interrupts are input to the microprocessor as a 4-bit code on *interrupt code lines* IC_0 through IC_3. *Interrupt request* ($\overline{INTREQ}$) signals the 99000 that a code is available on the IC lines. On the other hand, the reset and nonmaskable interrupts have dedicated input leads. These inputs are labeled $\overline{RESET}$ and $\overline{NMI}$, respectively.

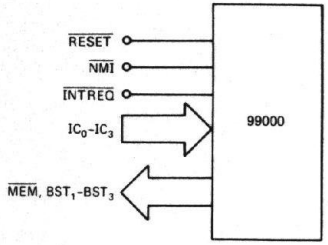


Figure 7.1 Interrupt interface of the 99000.

An *interrupt acknowledge code* is output on the status bus, $\overline{MEM}BST_1$-BST_2BST_3, during the interrupt control transfer sequence. This code is equal to 0101_2. It can be decoded in external circuitry to produce an *interrupt acknowledge* (INTA) signal. INTA indicates to external circuitry that the request for service has been accepted.

7.4 INTERRUPT PRIORITY LEVELS

The external interrupts of the 99000 are assigned priority levels. Figure 7.2 shows that the priority levels are called 0 through 15 and that they correspond to external user-definable interrupts INT_0 through INT_{15}. Level 0 represents the highest priority and level 15 the lowest priority. Moreover, we see that INT_0 through INT_{15} correspond to interrupt request codes IC_0IC_1-IC_2IC_3 equal 0000_2 through 1111_2, respectively.

Interrupt input	Priority level	Interrupt code			
		IC_0	IC_1	IC_2	IC_3
INT_0	0	0	0	0	0
INT_1	1	0	0	0	1
INT_2	2	0	0	1	0
INT_3	3	0	0	1	1
INT_4	4	0	1	0	0
INT_5	5	0	1	0	1
INT_6	6	0	1	1	0
INT_7	7	0	1	1	1
INT_8	8	1	0	0	0
INT_9	9	1	0	0	1
INT_{10}	10	1	0	1	0
INT_{11}	11	1	0	1	1
INT_{12}	12	1	1	0	0
INT_{13}	13	1	1	0	1
INT_{14}	14	1	1	1	0
INT_{15}	15	1	1	1	1

Figure 7.2 External interrupt priority levels.

The importance of priority lies in the fact that if a interrupt service routine has been initiated to perform a function at a specific priority level, only devices with higher priority can interrupt the active service routine.

Lower-priority-level devices will have to wait until the routine is completed before their request for service can be acknowledged. For this reason, the user normally assigns tasks that must not be interrupted frequently to higher-priority levels and those that can be interrupted to lower-priority levels.

An example of a high-priority service routine that should not be interrupted is that for a power failure. For this reason, it is typically assigned to priority level 1.

Example 7.1

If a level 3 interrupt is presently being serviced, which other level interrupt request could interrupt its service routine?

Solution: Only interrupts with higher-priority levels can cause the level 3 service routine to be interrupted. That is, just levels 0, 1, and 2 could be initiated.

7.5 EXTERNAL INTERRUPT REQUEST

The request for service by a device that uses one of the external user-definable interrupts is initiated through external circuitry. This circuitry must put the request code corresponding to the priority level, 0000_2 through 1111_2, of the highest-priority active interrupt onto lines $IC_0IC_1IC_2IC_3$ and then switch $\overline{INTREQ}$ to logic 0. For instance, if a level 5 interrupt is to be requested, the code $IC_0IC_1IC_2IC_3 = 0101$ is input together with $\overline{INTREQ} = 0$.

The 99000 tests the logic level of $\overline{INTREQ}$ each machine cycle. If it is found to be logic 0, execution of the current instruction is first completed and then the code at IC_0 through IC_3 is read into an internal interrupt code register.

7.6 PRIORITY ENCODER

Priority levels are assigned to external devices that request service with interrupts by a circuit known as a *priority encoder*. Figure 7.3 shows how it interfaces to the 99000. This circuit can have one interrupt input for each of the 99000 priority levels. These inputs are labeled $\overline{INT}_1$ through $\overline{INT}_{15}$. The signal that indicates that service is required by an external device is just wired to the input of the encoder that corresponds to the priority level that is to be assigned to the device. When an input is at its active 0 logic level, the encoder produces the 4-bit interrput code at its IC_0 through IC_3 outputs and switches the interrupt request signal, $\overline{INTREQ}$, to its active level.

The priority encoder section also includes circuits that synchronize the application of the interrupt signals to the input of the encoder section.

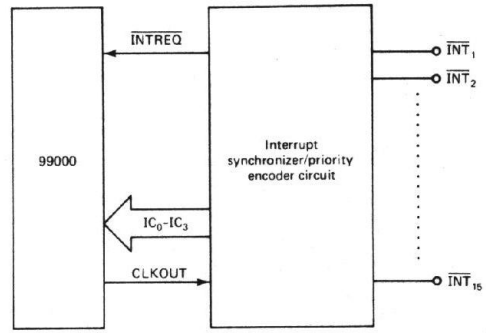


Figure 7.3 Interfacing the priority encoder to the 99000.

This is done by providing latches that sample the interrupt signals supplied by the external devices during each machine cycle of the 99000. The sampling of $\overline{INT}_1$ through $\overline{INT}_{15}$ is synchronized with CLKOUT.

7.7 CONTEXT SWITCH SEQUENCE

When an interrupt request is acknowledged, a transfer of control is initiated to a new program environment. As indicated earlier, the new segment of program to which control is passed is called the *interrupt service routine*. The transition to the service routine occurs through a mechanism known as a *context switch*.

The *context switch sequence* of an interrupt is illustrated in Fig. 7.4. Notice that before the interrupt occurs, the 99000 is executing instructions in the program environment identified as A. For this reason, its internal registers contain PC_A, WP_A, and ST_A. When set in this way, the instructions of program A are being fetched one after the other from memory and executed. The 16 registers in RAM identified as workspace A are being used as working data registers.

If an external interrupt occurs while the 99000 is executing program A, the instruction that is presently being executed is first completed. Then a context switch is initiated to the new program environment corresponding to the interrupt's service routine. For instance, if an interrupt is activated that has program B for its service routine, a context switch will be initiated to pass control to the first instruction in program B and workspace B is brought in for data operations.

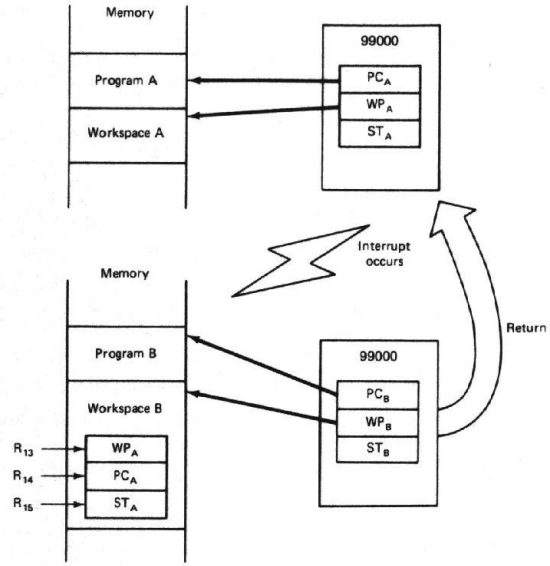


Figure 7.4 Context switch mechanism (Courtesy of *Texas Instruments, Incorporated.*)

The transfer of control to program B begins as the 99000 automatically brings in the service routine vector, WP_B and PC_B, from memory and puts them into the workspace pointer and program counter, respectively. This vector defines a new program environment. PC_B is the starting address of the service routine in program memory, and WP_B allocates a new workspace in data memory.

During the memory read cycles that are performed to bring in the values of WP_B and PC_B, the interrupt acknowledge (INTA = 1101) bus status code is output on lines $MEMBST_1BST_2BST_3$. As pointed out earlier, this code can be decoded with external circuits to produce a signal that acknowledges to the requesting device that service has been granted.

Next, the 99000 automatically saves the old internal register values, WP_A, PC_A, and ST_A, in registers R_{13}, R_{14}, and R_{15}, respectively, of the new workspace. To do this, it must perform three memory write cycles. By saving these values, linkage for return to the old program environment is preserved.

The *4-bit mask* in bits ST_{12} through ST_{15} of the status register are next

set to a value equal to one less than the priority level of the active interrupt. In this way, lower-priority interrupts are masked out. This completes the transition sequence and the 99000 is ready to start executing program B, thereby servicing the external device.

A *return workspace* (RTWP) instruction must be included at the end of the service routine, program B. When this instruction is executed, it initiates a return sequence that passes program control to the original program environment. In our example, this is program A.

As shown in Fig. 7.4, the return sequence begins with the 99000 fetching the old internal register contents, PC_A, WP_A, and ST_A, from registers R_{13}, R_{14}, and R_{15} of workspace B, and returning them to its internal registers. Then the 99000 resumes instruction execution. In this way, program control picks up in program A at a point where it left off due to the occurrence of the interrupt request.

One of the major benefits of the architecture of the 99000 lies in its very fast context switch mechanisms. In fact, the context switch sequence we just described takes only 2.3 μs to pass control to the service routine. This is due to the fact that the 99000 has just three internal registers instead of a large register file that must be saved before initiating a service routine. Fast context switching is important in interrupt intensive system environments, large multiuser systems, and for implementation of high-level languages.

7.8 INTERRUPT VECTORS AND THE INTERRUPT VECTOR TABLE

As just mentioned, the entry point for the service routine of an interrupt is defined by a *vector* consisting of a 16-bit program counter (PC) address and a 16-bit workspace pointer (WP) address. In the 99000 system, these vectors must be stored in a dedicated area of memory.

Figure 7.5 is a memory map showing in detail the locations of the interrupt vectors. Notice that the vector for the 99000's reset function is labeled WP_{RESET} and PC_{RESET}. They are stored at address 0000_{16} and 0002_{16}, respectively, in memory. The next vector is for the level 1 external interrupt vector WP_{INT1} and PC_{INT1} at 0004_{16} and 0006_{16}. It is followed by the vector for INT_2 through INT_{15} at word addresses up through $003C_{16}$ and $003E_{16}$.

In address range 0040_{16} through $007E_{16}$ of the vector table, we find 16 more two-word vectors. These vectors are labeled XOP_0 through XOP_{15} and are for use by the extended operation instructions (XOPs) of the 99000. XOPs are the software interrupts of the 99000 microcomputer.

A last vector is stored at addresses $FFFC_{16}$ and $FFFE_{16}$. It is the PC and WP for the nonmaskable interrupt service routine.

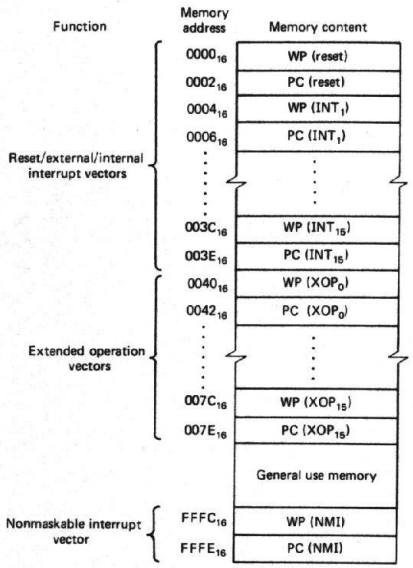


Figure 7.5 Interrupt vectors. (Courtesy of *Texas Instruments, Incorporated.*)

The address values that are stored in these locations are automatically loaded into the workspace pointer register and program counter of the 99000 as part of the context switch mechanism.

Example 7.2

What are the addresses of WP and PC for the level 3 interrupt vector?

Solution: Looking at Fig. 7.5, we find that the level 3 interrupt vector is stored at

$$WP_{INT3} = 000C_{16}$$

and

$$PC_{INT3} = 000E_{16}$$

7.9 INTERRUPT MASK

In our description of the context switch mechanism, we mentioned that with the initiation of an interrupt, the *interrupt mask* in the status register is set to a level equal to one less than that of the initiated interrupt. An example is

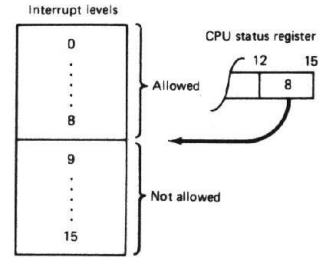


Figure 7.6 Interrupt mask for active level 9 interrupt. (Courtesy of *Texas Instruments, Incorporated.*)

shown in Fig. 7.6. Here we see that a level 9 interrupt has been initiated. Therefore, the mask is set to $1000_2 = 8_{16}$, respresenting level 8.

Actually, prior to acknowledging an interrupt request, the 99000 first compares the interrupt code at IC_0 through IC_3 to the current value held in the interrupt mask. If the code is *lower priority* (higher-level number) than the value currently in ST_{12} through ST_{15}, the interrupt is masked out and will not be acknowledged until the active service routine has run to completion. For the example in Fig. 7.6, interrupt levels 9 through 15 are masked out. In order to be acknowledged, they must first wait for the level 9 service routine to be completed.

On the other hand, if the new request is for yet a *higher-priority* (lower-level number) interrupt, the present routine is suspended and a context switch initiated to the service routine for the new higher-priority interrupt. From Fig. 7.6 we see that levels 0 through 8 can be initiated.

The interrupt mask can be set to any value through software. This is done with the load immediate interrupt mask (LIMI) instruction. As discussed in Chapter 3, the format of this instruction is

```
LIMI  I
```

Here I is the immediate operand that gets loaded into the mask.

Using the LIMI instruction, program control can be used to enable/disable certain ranges of interrupts. For instance,

```
LIMI  7
```

sets the mask to 0111. This disables external user-defined interrupts for priority levels from 7 through 15 while leaving those for levels 0 through 6 enabled.

The load status (LST) instruction can also be used to manipulate the interrupt mask. It differs from LIMI in that all bits of the status register are affected and the operand resides in a workspace register.

Example 7.3

To what value is the interrupt mask set when a level 3 interrupt has been initiated?

Solution: The interrupt mask is set to 2 when the level 3 interrupt is initiated.

$$ST_{12}ST_{13}ST_{14}ST_{15} = 0010_2$$

7.10 RESET INTERRUPT

The *reset interrupt* is provided to allow *hardware initialization* of the 99000 microcomputer system. Normally, this is done only when power is applied to the system. It can be used to initialize the internal registers of the 99000 and external hardware signals such as those for I/O to their starting logic levels.

The $\overline{\text{RESET}}$ lead is the reset interrupt input of the 99000. It represents the level 0 or highest-priority interrupt and cannot be masked out. The logic level at $\overline{\text{RESET}}$ is sampled at each 1-to-0 transition of CLKOUT. If logic 0 is found at this input for three consecutive samples, the 99000 stops execution of the current instruction and sets signal lines $\overline{\text{WE}}$/$\overline{\text{IOCLK}}$, $\overline{\text{RD}}$, and $\overline{\text{MEM}}$ to their inactive levels. It stays in this state until $\overline{\text{RESET}}$ returns to the 1 logic level.

Two clock cycles after $\overline{\text{RESET}}$ switches back to logic 1, a context switch is initiated to the reset function interrupt service routine. This is done through the vector WP_{RESET} and PC_{RESET} that is stored at addresses 0000_{16} and 0002_{16} in memory. Next, the 99000 clears the status register and error register. After this happens, execution of the *power-up routine* is initiated. This service routine can be used to initialize the system resources such as I/O ports, the interrupt mask, and memory. Then it passes program control to the start of the microcomputer's *application program*.

Figure 7.7 shows a simple circuit that can be used to produce the hardware reset signal. When the switch is depressed, capacitor C_1 is discharged through R_1. This gives the 0 logic level at the input of the buffer with

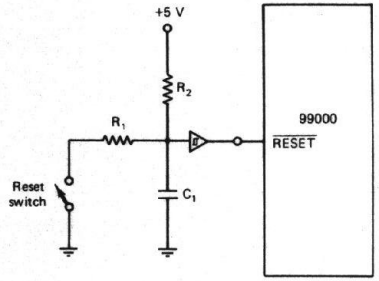


Figure 7.7 Reset circuit.

Schmitt trigger input. The output of this buffer, which is also 0, is applied to $\overline{\text{RESET}}$. Logic 0 at this input is an active reset and causes the 99000 to suspend operation. As the switch is released, C_1 recharges through R_2 back to the 1 logic level and the initialization software routine is initiated.

The values of R_2 and C_1 must be selected to assure that three clock cycles elapse before $\overline{\text{RESET}}$ returns to the one level.

7.11 NONMASKABLE INTERRUPT

The *nonmaskable interrupt* of the 99000 differs from the external interrupts in that it cannot be masked out by the interrupt mask in the status register. That is, if an external user-defined interrupt happens to be in progress when a device uses the nonmaskable interrupt to request service, its service routine is suspended and the nonmaskable interrupt's service routine is initiated. This interrupt input is typically used to implement specialized functions such as a *load function* or *single-step mode* of operation.

A nonmaskable interrupt is requested by switching the $\overline{\text{NMI}}$ input of the 99000 to logic 0. Upon completion of the current instruction, a context switch is initiated to its service routine. The workspace and entry point of the nonmaskable interrupt service routine is identified by the address vector WP_{NMI} and PC_{NMI}. This vector is stored in memory at addresses $FFFC_{16}$ and $FFFE_{16}$. As part of the context switch, the interrupt mask in the status register is automatically cleared to 0000_{16}. This masks out all external interrupts.

7.12 EXTERNAL INTERRUPT INTERFACE CIRCUITRY

Up to this point in the chapter, we have discussed interrupts relative to the response initiated by the 99000 after an interrupt code and interrupt request have been input. However, we did indicate that external circuitry is required to synchronize, priortize, and encode the interrupt signals before they are applied to the interrupt interface. Sometimes, additional circuitry is added to the external interface to enhance further the 99000's interrupt structure. For instance, circuits can be included to permit masking of individual interrupt inputs or for creating edge-triggered interrupt inputs.

Let us now look at some circuits that can be used for these purposes.

Eight-Input Interrupt Interface

A device that can be used to provide the priority encoder function for the 99000 is the 74LS148 priority encoder. One of these devices can be used to construct an *eight-interrupt interface* for the 99000 microcomputer system.

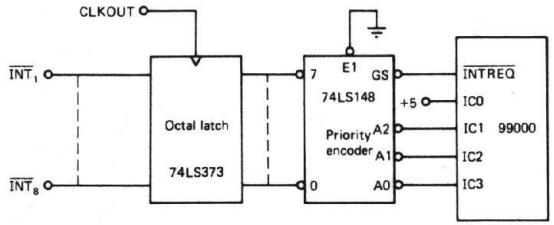


Figure 7.8 Eight-interrupt interface circuit. (Courtesy of *Texas Instruments, Incorporated.*)

This is done by connecting the device as shown in Fig. 7.8. Notice that a 74LS373 octal latch has been added to synchronize the inputs of the encoder. In this way, the logic levels at interrupt inputs $\overline{INT}_1$ through $\overline{INT}_8$ are sampled and latched once each machine cycle. This occurs on the 0-to-1 edge of CLKOUT.

The outputs of the 74LS373 latch are applied to inputs 0 through 7 of the 74LS148 priority encoder. Here they are prioritized and the code corresponding to the highest-priority active input is generated at output $A_2A_1A_0$. This output code is applied to $IC_1IC_2IC_3$ of the 99000. The unused interrupt code input, IC_0, is fixed at the 1 logic level.

Output GS of the priority encoder, which switches to logic 0 whenever an input is active, is connected to the $\overline{INTREQ}$ input of the 99000. In this way, it signals the 99000 that a device is requesting service, and that a valid code is available at the output of the encoder. In response, the 99000 reads the code at $IC_0IC_1IC_2IC_3$ and if acknowledged, a context switch is initiated to the corresponding service routine.

Fifteen-Input Interrupt Interface

Figure 7.9 shows how two 74LS148 devices can be interconnected to provide a *15-interrupt interface* for the 99000. Notice that the interrupt inputs are first synchronized to CLKOUT and latched at the inputs of the encoder. The upper 74LS148 has as its inputs the seven higher-priority interrupt inputs, $\overline{INT}_1$ through $\overline{INT}_7$. The lower device accepts interrupt inputs $\overline{INT}_8$ through $\overline{INT}_{15}$. At the output side of the encoder, we see that the GS, A_2, A_1, and A_0 outputs of the two devices are gated together to generate the signals for the $\overline{INTREQ}$, IC_1, IC_2, and IC_3 inputs of the 99000. Moreover, the EO output of the upper encoder is inverted and supplied to IC_0 to distinguish between the two groups of eight interrupt inputs.

For instance, when an interrupt occurs, one of the interrupts in the group $\overline{INT}_1$ through $\overline{INT}_7$, EO, equals 1. This makes IC_0 switch to the 1

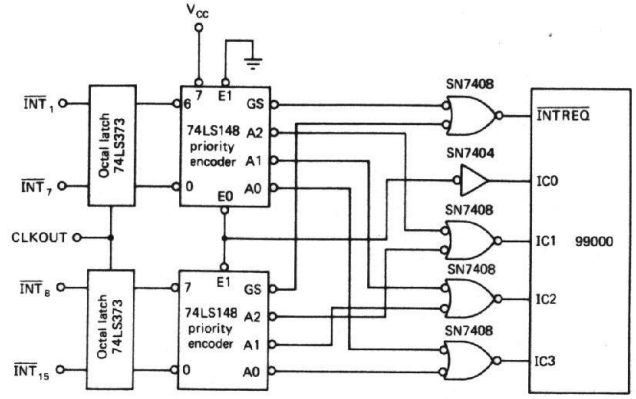


Figure 7.9 Fifteen-interrupt interface circuit. (Courtesy of *Texas Instruments, Incorporated.*)

logic level. At the same time, EO is applied to the EI input of the lower-priority encoder. Logic 1 at this input disables the device and its outputs are forced to the 1 logic level. In turn, the gates are enabled for use by the upper-priority encoder device. Therefore, the IC_1 through IC_3 outputs produced by the gates correspond to the logic levels output at A_2 through A_0, respectively, of the upper-priority encoder device.

Edge-Triggered Interrupt Inputs

The interrupt inputs provided by the 74LS148 in the circuit of Figs. 7.8 and 7.9 are level detecting. Therefore, the external device supplying the request for service must maintain the signal at the interrupt input until it is acknowledged by the 99000. Otherwise, the request for service would be lost.

Some external devices produce a short-duration pulse instead of a fixed logic level for use as an interrupt signal. If the 99000 is busy servicing a higher-priority interrupt when the pulse is produced, the request for service could be completely missed. To overcome this problem, additional circuitry can be included in the interrupt interface to convert the level-sensitive interrupt inputs to positive or negative edge-triggered inputs.

The circuit shown in Fig. 7.10 demonstrates how a D-type flip-flop can be connected to make interrupt 7 negative edge triggered. On the 1-to-0 transition of INT_7, the logic 1 at the D input of the 74LS74 flip-flop causes its outputs to set. This makes $\overline{Q}$ logic 0 and issues a level 7 interrupt request

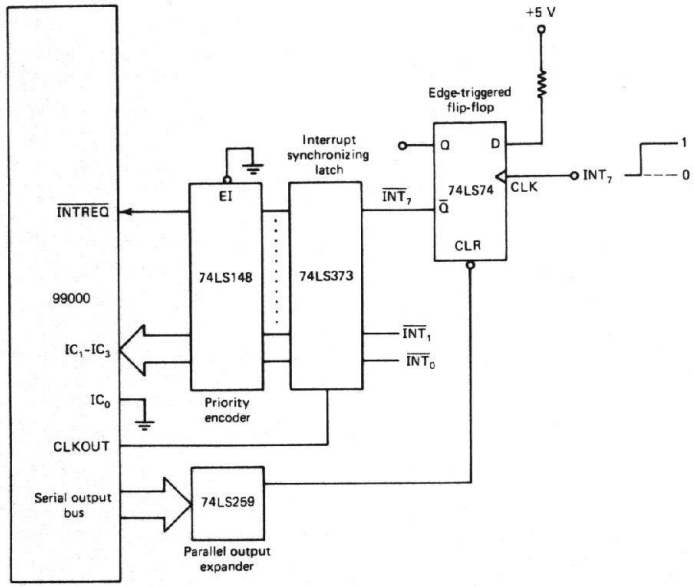


Figure 7.10 Negative edge-triggered interrupt input.

to the 99000. Even if INT_7 returns to logic 1, the $\overline{Q}$ output of the flip-flop stays at logic 0. In this way, the request for service is maintained until it is acknowledged by the 99000.

Notice that the flip-flop is cleared with an output line from the serial output interface. This output must be pulsed to logic 0 as part of the interrupt service routine to remove the interrupt request. Otherwise, the interrupt would be initiated a second time when its service routine runs to completion.

7.13 EXTENDED OPERATION INSTRUCTIONS

The *extended operation* (XOP) instruction of the 99000 provides a mechanism for performing a vectored call of a software routine. Typically, this software routine is an emulation routine for a more complex function. However, they can also be used to perform supervisor calls for an operating

system. The XOP emulation routines are written using assembly language instructions, assembled into machine code, and stored in the main program memory of the 99000 microcomputer system.

The general form of the XOP instruction is

```
XOP  S,n
```

where n is the number assigned to the extended operation and can be any number from 0 through 15. For instance, XOP_0 could define a floating-point addition instruction and XOP_1 a floating-point subtraction instruction.

On the other hand, S is an effective address that can be used as a pointer to the location of a table in memory that is used to pass parameters to the extended operation routine. For example, in the case of a floating-point addition emulation routine, the main program must put the two floating-point numbers into the table such that they can be accessed during the addition.

The starting point and workspace data area used by each XOP instruction is defined by a vector stored in the interrupt vector table in memory. Earlier we identified vector table locations in the range 0040_{16} through $007E_{16}$ as those assigned to vectors for XOP_0 through XOP_{15}. For example, XOP_1 causes a vectored call of the subroutine corresponding to WP_{XOP1} and PC_{XOP1} located at word addresses 0044_{16} and 0046_{16}, respectively.

The XOP instructions act like *software interrupts*. This is because, when executed, they initiate essentially the same context switch sequence that takes place when the 99000 services a hardware interrupt. That is, its vector is fetched from memory and loaded into WP and PC. Then the old contents of WP, PC, and ST are saved in registers R_{13}, R_{14}, and R_{15}, respectively, of the new workspace. The one difference is that the source address S is loaded into R_{11} of the new workspace. Therefore, instructions in the emulation routine can access parameters in the table by simply using the indirect workspace register with displacement addressing mode.

Just like for the BLWP instruction, a RTWP instruction is needed at the end of the extended operation routine to return control to the main part of the program. In this way, execution is resumed with the instruction that follows the XOP instruction.

The XOP instruction operates in this way only when status bit ST_{11} is logic 0.

7.14 INTERNAL INTERRUPT FUNCTIONS

Internal interrupts represent *error conditions* which when they occur are detected automatically by the 99000. As identified earlier, the three internal interrupt functions are *illegal opcode detection, privileged mode viola-*

tion detection, and *arithmetic overflow detection*. These internal functions are dedicated to the *level 2 interrupt*. The privileged mode and arithmetic overflow interrupt functions are maskable and can be interrupted by level 0 and level 1 interrupts. On the other hand, the illegal opcode interrupt is nonmaskable.

Whenever one of these three internal conditions occurs, the 99000 sets the corresponding bit in its internal error register and then initiates a request for a level 2 interrupt. Instructions in the level 2 service routine can examine the bits of the error register to identify which ones are set. Based on these findings, a branch can be initiated to a segment of program that is provided to service the error condition.

If none of the error bits are set, an external level 2 interrupt request has

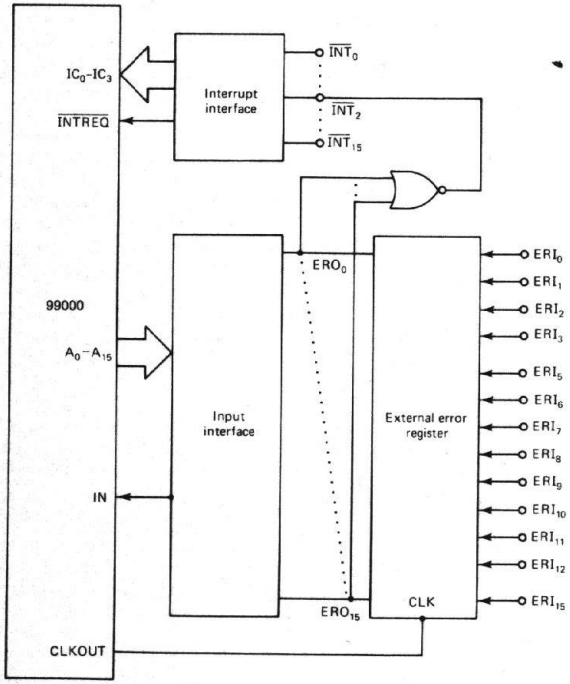


Figure 7.11 External error register.

been issued and its service routine is initiated through the vector WP_{INT2} and PC_{INT2}. This routine could be written to examine bits in an externally implemented error register in which bits have been assigned to additional functions. This *external error register* can provide externally detected system level errors such as *memory parity errors, addresses that are out of range*, or *an attempt to write into write-protected memory*.

Figure 7.11 shows how an external error register can be implemented in the 99000 microcomputer system. Notice that *error inputs* (ERI) are provided for only those error conditions that are not already implemented in the internal register.

Looking at the circuit diagram, we see that the contents of the external error register is updated each CKLOUT cycle. If any of the error conditions have occurred, the corresponding error register outputs become logic 1. This causes the output of the NOR gate to switch to logic 0. This output is supplied to $\overline{INT}_2$ and signals the 99000 of an external error by initiating a level 2 interrupt request. Once initiated, the service routine for the external level 2 interrupt can examine the bits of the external error register through the serial input interface to identify which error condition has occurred. Then a branch can be made to the appropriate service routine.

7.15 ILLEGAL OPCODE DETECTION, MACROINSTRUCTION DETECTION, AND MACROINSTRUCTION EMULATION

Illegal opcode detection (ILLOP) is one of the internal functions dedicated to the level 2 interrupt of the 99000. Using this mechanism, the attempt to execute an opcode that does not correspond to one of the instructions in the instruction set of the 99000 is identified. It is flagged with an error condition and then an interrupt initiated to pass control to an error service routine.

Illegal Opcode Detection Sequence

Figure 7.12 shows the sequence in which the 99000 microprocessor determines how to initiate execution after fetching an opcode from program memory. Here we see that if it is identified as a *standard opcode*, a branch is taken to a microcode routine that performs the function. However, when an illegal opcode is detected during the instruction acquisition cycle, the 99000 does not execute the instruction. Instead, it first outputs the *macroinstruction detect* (MID) *bus status code* (1110) on lines $\overline{MEM}BST_1BST_2BST_3$. Then it checks the other system resources: *attached processors, attached computers, macrostore*, and *main memory software emulation* to determine if they can perform a function for the opcode. If they cannot, it handles the illegal opcode as a violation. In this case, the illegal opcode flag (ILLOP), bit 13 in the error register, is set and a level 2 interrupt initiated.

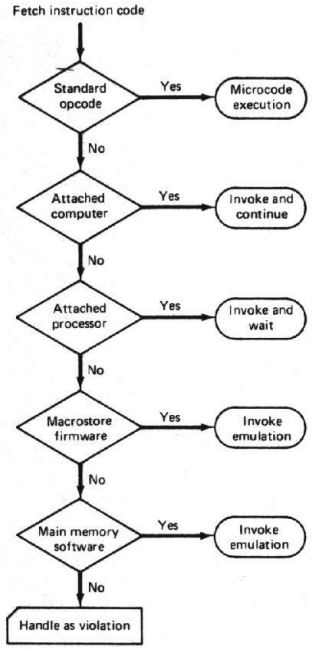


Figure 7.12 Emulation routine invocation mechanism. (*Texas Instruments, Incorporated.*)

The interrupt service routine must test the ILLOP bit of the error register to confirm that the cause of the level 2 interrupt was an attempt to execute an illegal opcode. Since the ILLOP bit of the error register is located at I/O software address $1FD2_{16}$, it can be read with a test bit I/O instruction and then tested with a jump on equal instruction. Once initiated, the level 2 interrupt service routine for detection of an illegal opcode cannot be interrupted. At completion of the service routine, a SBZ instruction must be executed to reset the ILLOP error bit; otherwise, its service routine will be reinitiated.

Macroinstruction Detection

Macroinstruction detection (MID) is an extension of the ILLOP detection mechanism we just described. Through it, the baseline instruction set of the 99105A can be easily enhanced with emulation routines for additional in-

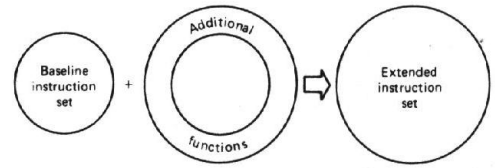


Figure 7.13 Macrostore instruction set extensions. (*Texas Instruments, Incorporated.*)

structions called *macroinstructions*. This concept is illustrated in Fig. 7.13. For example, the additional functions could be emulation routines that implement more complex, specialized operations such as those for *floating-point* and *double-precision arithmetic*. Opcodes that are not used by the instruction set of the 99105A are assigned to these new macroinstructions. The result is a 99000 family component with a more powerful instruction set.

As shown in Fig. 7.12, the MID mechanism has the ability to pass control to a high-performance *firmware emulation routine* or an even higher-performance hardware emulation such as an attached processor or attached computer.

Macrostore memory is a special high-performance memory section of the 99000 microcomputer system whose primary use is for storage of firmware emulation routines of macroinstructions. The emulation routines for macroinstructions are not microcoded into the 99000. They are written and debugged in assembly language, assembled into machine code, and then programmed into the *macrostore program memory area*.

The macrostore memory feature permits creation of a family of 99000 devices with specialized instructions designed for use in common types of applications. An example is a device with standard macrostore instructions for performing floating-point arithmetic instructions. The 99110A floating-point microprocessor is a device that has this type of instruction set. This capability can also be employed by a user to design a custom device with some special-purpose instructions that are important to their application. In this way, we see that macrostore increases the flexibility of the 99000 microcomputer system and improves its overall performance.

Macrostore Memory Address Space

The 99000 microprocessor has the ability to address up to 64K bytes of macrostore memory. This macrostore memory space is totally independent of the 256K-byte main memory address space we introduced in Chapter 5.

Figure 7.14 shows an address map of the macrostore memory space. Notice that the lower 4K of addresses are dedicated for use internal to the

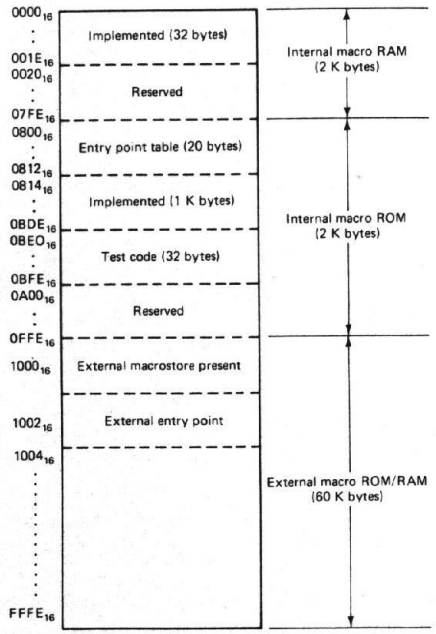


Figure 7.14 Macrostore address space. (*Texas Instruments, Incorporated.*)

99000 device. This *internal macrostore* starts at address 0000_{16} and ends at address $0FFF_{16}$. It contains both a *macro-RAM* area and a *macro-ROM* area.

Macro-RAM is located from address 0000_{16} to $001E_{16}$ and contains 32 bytes (16 words) of RAM. It is provided for use as a high performance workspace by executing macroinstructions. The block of addresses from 0020_{16} up through $07FF_{16}$ are reserved for expansion of macro-RAM to 2K bytes.

The rest of the internal macrostore address space, 0800_{16} to $0FFF_{16}$, is dedicated to *mask-programmable macro-ROM*. This area is mask-programmed during manufacturing with the machine code routines needed to implement the desired macroinstruction functions. The initial 99000 device has 1K bytes (512 words) of macro-ROM. However, enough address space has been reserved for expansion up to 2K bytes (1K words).

Macro-ROM also contains a reserved area identified as the *macrostore entry-point table* and another reserved area labeled *test code*. This entry-point table contains address vectors that identify the starting points of the various macroinstruction routines stored in macro-ROM. Notice that 20 bytes (10 words) can be stored at addresses 0800_{16} through 0812_{16}. The other reserved area, which corresponds to the 32-byte locations at addresses $0BE0_{16}$ through $0BFE_{16}$, are used for storage of code needed to test macrostore memory during manufacturing.

The other 60K bytes of the address space, which lies in the address range from 1000_{16} to $FFFF_{16}$, correspond to what is known as *external macrostore memory*. This section permits external expansion of the macroinstruction storage area and can be implemented with high-speed ROM and RAM devices. Word location 1000_{16} is reserved for storage of a code that identifies whether or not external macrostore is present in the system.

Entry-Point Table

As mentioned earlier, the entry-point table contains the 16-bit address vectors that identify the starting points of macroinstructions in macro-ROM. Figure 7.15 shows the locations of the various *entry-point vectors* and the illegal opcodes that use each vector.

The first eight vector locations are associated with groups of illegal opcodes for single-word instructions. Actually, bits 4 through 7 of the illegal opcode determine which vector is selected to initiate the macroinstruction. Notice in the table that if these 4 bits are $0000_2 = 0_{16}$, control is passed to

Vector	Address	MID opcodes
1	0800	0000-001B, 001E-0028, 002B-007F, 00A0-00AF, 00C0-00FF
2	0802	0100-013F
3	0804	0210-021F, 0230-023F, 0250-025F, 0270-027F, 0290-029F, 02B0-02BF, 02D0-02DF, 02E1-02FF
4	0806	0301-031F, 0320-033F, 0341-035F, 0361-037F, 0381-039F, 03A1-03BF, 03C1-03DF, 03E1-03FF
5	0808	0C00-0C08, 0C0C-0CFF
6	080A	0D00-0DFF
7	080C	0E00-0EFF
8	080E	0F00-0FFF, 0780-07FF
9	0810	AM, SM, SRAM, SLAM, TMB, TCMB, TSMB (illegal second word)
10	0812	XOP (if $ST_{11} = 1$)

Figure 7.15 Entry-point vector locations and corresponding MID opcodes. (*Texas Instruments, Incorporated.*)

the address stored in table location 0800_{16}. On the other hand, if the opcode has $0001_2 = 1_{16}$ in these bit positions, transfer of control is to the address held at location 0802_{16}.

A few of the instructions in the 99000's instruction set are considered two word instructions. If their second word initiates a MID, control is passed to the emulation routine through the vector at address 0810_{16} of the entry-point table. Examples of two-word instructions are: double-precision add (AM), double-precision subtract (SM), and shift right arithmetic double (SRAM).

The extended operation instructions (XOPs) can also be used to initiate instructions in macrostore. However, this capability is optional and can be enabled or disabled with bit 11 of the status register.

Modes of Macrostore Interface Operation

The macrostore interface of the 99000 can be initialized to work in one of three modes. Its simplest mode is obtained by simply tying the attached processor present ($\overline{\text{APP}}$) input to ground and then performing a hardware initialization with the $\overline{\text{RESET}}$ input. When the 99000 is set up and initialized in this way, the macrostore interface is disabled. Therefore, detection of an illegal opcode automatically initiates a level 2 interrupt.

The most common mode of operation is that known as the *standard mode*. The 99000 is put in this state by holding $\overline{\text{APP}}$ at the 1 logic level during hardware reset. When set to work in this way, the internal macro-ROM and macro-RAM as well as external macrostore memory are all functional. Detection of a macroinstruction through the MID mechanism causes a vector to transfer control to the entry point of the routine in macro-ROM. As the instructions in the routine are executed one after the other, the macro-RAM workspace is typically used for data operations.

Read and write bus cycles to the internal macrostore memory are always performed in one machine state and are transparent to the external memory interface. Moreover, the timing for read or write cycles of external macrostore are identical to those for system memory. However, they are accompanied by the macrostore access bus status code, AUMS = 1001, instead of those output for accesses of main system memory. This code can be externally decoded to enable the address decoder for the external macrostore memory system. Control signals $\overline{\text{RD}}$, R/$\overline{\text{W}}$, and $\overline{\text{WE}}$ indicate to external macrostore memory whether a read or write operation is to take place.

The third mode, which is known as *prototyping mode*, is achieved by pulling both $\overline{\text{APP}}$ and $\overline{\text{RESET}}$ to logic 0 together. When the 99000 is initialized in this way, internal macro-ROM is disabled. At the same time, its address range is mapped into the external macrostore address space. On the other hand, internal macro-RAM stays functional just as in the standard mode.

This prototyping mode permits storage of macroinstruction code in external PROM for testing. After debugging is complete, this code can be brought on-chip to the macro-ROM area. This facilitates easy development, testing, and prototyping of custom macrostore functions. Moreover, if the production needs of the application do not warrant the cost of generating masks for internal macro-ROM, the 99000 can be left in this mode when implemented in the final system design.

Internal Macrostore Entry and Exit

When the illegal opcode detection mechanism identifies an illegal opcode that is emulated in macrostore, the MID mechanism passes program control to an emulation routine in macrostore memory. The first step in the macrostore entry sequence is that the illegal opcode is saved in register R_5 of the macro-RAM workspace. Then a context switch is initiated to the macroinstruction in macrostore memory. The entry-point vector associated with the MID opcode is read out of the table and loaded into the 99000's program counter and the workspace pointer is cleared to 0000_{16}. Next, the old WP, PC, and ST are saved in registers R_{13}, R_{14}, and R_{15} of the macro-RAM workspace. Remember that the value of PC saved in register R_{14} points to the instruction following that which invoked the macroinstruction. Therefore, this preserves return linkage to the original program context.

At this point, the 99000 fetches the first instruction of the emulation routine from macro-ROM and executes it. Execution of instructions from macro-ROM continues until the macroinstruction function is complete. During execution of the macroinstruction routine, data stored in main memory can be accessed as operands. At the end of the routine, the old status value that is held in register R_{15} may be updated to represent the status at completion of the macroinstruction.

At entry of the macroinstruction routine, the illegal opcode, which is saved in R_5, can be examined by the emulation routine. In this way, it can be used to initiate a branch to one of a number of different functions related to the opcodes in the group that vectors through the same entry-point table location.

Macroinstruction emulation routines must be terminated with a return workspace (RTWP) instruction. It is this instruction that causes the 99000 to exit execution from macro-ROM and return control to execute instructions in main system memory. When RTWP is executed, the old WP, PC, and updated ST are returned to the internal registers of the 99000. Execution picks up in the main program memory at the instruction following that which is called the macroinstruction.

If a group of illegal opcodes are not to be supported by emulation routines in macrostore, control is still passed to the entry-point table in macrostore memory. A special return instruction, which is initiated with

opcode 0382_{16}, must be included at the entry-point address of the macro-instruction routine in macro-ROM. Execution of this instruction causes exit of macrostore memory and reentry of the ILLOP detection mechanism. At termination, the ILLOP detect-error condition is generated and then a level 2 interrupt initiated.

7.16 PRIVILEGED MODE

To permit creation of a *user/supervisor system environment*, the 99000 has been equipped with a *privileged mode* of operation. Selection of this mode of operation is done through bit 7 of the status register. It can be set to logic 1 under software control to enable the privileged (supervisor) mode. This designates certain I/O and system functions as privileged.

When the 99000 system is set in this way, the operator designated as the *user* (*nonprivileged*) only has access to those system resources not identified as privileged, while the operator designated as the *supervisor* (*privileged*) has access to all system resources. If the user attempts to access a privileged resource, execution is inhibited. Moreover, the attempt is flagged with an error condition by setting the PRIVOP bit in the error register and then a service routine is initiated through the level 2 interrupt.

Privileged Mode Instructions

The instructions that fall into the group known as *privileged mode instructions* are those that affect output, modify the contents of the status register, the external instructions, and instructions that deal with extended system memory. Figure 7.16 lists the mnemonics for these instructions, their names, and identify their functions.

In terms of the I/O interface, output operations initiated by the SBO, SBZ, or LDCR instructions are restricted when the 99000 is in the user mode. This restriction is true only over a specific range of the I/O address space. That is, the user-mode output restriction applies to the address range from $1C00_{16}$ to $7FFE_{16}$ of the serial I/O address space and $9C00_{16}$ to $FFFE_{16}$ of the parallel I/O address space. For this reason, output functions that are to be accessible only when in supervisor mode must reside at one of these privileged addresses.

The privileged instructions that relate to status register operations are LIMI, RSET, LST, and RTWP. The restrictions on these instructions relate to their ability to change bits in the status register. For example, the LIMI and RSET instructions permit loading of the interrupt mask, bits ST_{12} through ST_{15} of the status register, with a new value and clearing of this mask, respectively. Since the contents of the status register cannot be

Mnemonic	Instruction	Function
SBO	Set bit one	Data output
SBZ	Set bit zero	
LDCR	Load communication register unit	
LIMI	Load interrupt mask	Modify internal registers
RSET	Reset	
LST	Load status	
RTWP	Return workspace pointer	
IDLE	Idle	External instruction output
RSET	Reset	
CKON	User defined	
CKOF	User defined	
LREX	User defined	
LDS	Long distance source	Extended memory
LDD	Long distance destination	

Figure 7.16 Privileged mode instructions.

modified while in the user mode, neither of these two instructions can be executed.

The other two instructions, LST and RTWP, provide for loading of the status register from a workspace register and for initiation of the return context switch mechanism, respectively. In the privileged mode, these instructions do execute; however, their affect is limited to modification of bits 0 through 5 and 10 of the status register. The rest of the bits remain unaffected.

In this way, we see that putting the 99000 in the privileged mode eliminates the ability of the user to manipulate the interrupt mask or modify the options selected with status bits.

The next group of instructions in Fig. 7.16 fall into the external instruction category. The IDLE, RSET, CKON, CKOF, and LREX instructions provide the ability to set a specified I/O port in the privileged I/O address space from $1EC4_{16}$ through $1ECE_{16}$ to the 0 logic level. For this reason, these instructions cannot be executed when in the user mode. As indicated earlier, the RSET instruction has a second function which is to reset the status register.

The I/O functions related to the external instructions provide the ability to define additional supervisor functions. For instance, they could be used to enable supervisor accessible memory.

The rest of the instructions in Fig. 7.16 relate to accessing extended memory. They are *long-distance source* (LDS) and *long-distance destination* (LDD). These two instructions are available only on the 99110A. They are also privileged and provide another means of restricting memory use in a user/supervisor environment.

Privileged-Mode Violation Detection

The 99000 has the ability to detect automatically the occurrence of a privileged instruction. When nonprivileged mode is enabled, an attempt to execute one of the privileged instructions represents a violation. The instruction is fetched and decoded but not executed. Instead, bit 14 (PRIVOP) of the error register is set to flag the occurrence of the privileged mode violation; an internal level 2 interrupt is requested; and then a context switch is initiated through the interrupt vector WP_2 and PC_2 to its service routine.

Earlier we pointed out that the level 2 interrupt can also be used by an external interrupt as well as the three internal interrupt conditions. For this reason, the service routine must examine the PRIVOP bit of the error register to determine whether or not the cause was a privileged-mode violation. This bit is located at I/O software address $1FD4_{16}$. If it is set, a branch must be initiated to a service routine for the privileged-mode violation. At the completion of the service routine, PRIVOP (bit 14 of the error register) must be cleared with an output operation.

7.17 ARITHMETIC FAULT DETECTION

The third internal interrupt function is *arithmetic fault detection* capability. That is, the 99000 has the ability to check automatically for the occurrence of an *overflow condition* during all arithmetic operations. The benefit of this feature is that no software overhead is needed to identify that an overflow has occurred.

Arithmetic fault detection is an option and is enabled by setting status bit 10 to logic 1. If an overflow occurs, the 99000 always flags the condition by setting bit 4 (AO) in the error register. However, for the level 2 interrupt to be initiated, the overflow enable bit must be set.

Just like for the other internal functions, the level 2 service routine must examine error bit 4 with an input operation to identify that in fact the cause of the interrupt was an arithmetic overflow. This bit is located at I/O software address $1FC0_{16}$. If it is set, a branch can be made to a routine for servicing the error condition. For instance, a message could be displayed to identify that an overflow has occurred. At the end of the routine, the overflow error bit must be cleared.

ASSIGNMENT

Section 7.2

1. What is the function of an interrupt?
2. Name the three types of interrupts supported by the 99000.

Section 7.3

3. How many external maskable interrupts can be applied to the 99000?

Section 7.4

4. What is meant by "interrupt priority"?
5. What value must be loaded into the interrupt mask to mask out interrupts with priority less than 7?

Section 7.5

6. How does the 99000 receive an interrupt request from a device with priority level 9?

Section 7.6

7. What function is served by a priority encoder circuit?
8. If the interrupt 2 and interrupt 15 inputs of the circuit of Fig. 7.3 are at their active 0 logic levels, what code is output to the 99000 on IC_0 through IC_3?

Section 7.7

9. What is the function of the context switch mechanism relative to interrupt servicing?
10. What information is saved to permit return from an interrupt service routine? Where is it saved?
11. What instruction is used to initiate the return back to the original program environment at completion of an interrupt service routine?

Section 7.8

12. What information is provided in the interrupt vector table?
13. Where does the vector for XOP_2 reside in memory? How is this vector organized?

Section 7.9

14. What happens to the interrupt mask when a level 5 interrupt is acknowledged for servicing?
15. Write an instruction that will initialize the interrupt mask to 15.

Section 7.10

16. When $\overline{\text{RESET}}$ is asserted, what happens to registers WP, PC, ST, and ER?

Section 7.11

17. How does NMI differ from a maskable interrupt and reset?

Section 7.12

18. If inputs $\overline{\text{INT}}_7$ and $\overline{\text{INT}}_8$ in the circuit of Fig. 7.8 become active, what are the outputs of the latch and priority encoder after the next pulse at CLKOUT?

Section 7.13

19. Write an instruction for XOP_8 in which the parameter table resides in memory starting at address >A000.
20. Give an overview of the mechanism by which execution of XOP_5 changes program context.

Section 7.14

21. What are the three internal interrupt functions of the 99000?
22. What priority level is assigned to handle internal interrupts?
23. How can software identify which of the internal interrupts has occurred?
24. Assuming that the external error register is located at I/O address >5000, write a level 2 service routine that tests the error register to determine if an external error condition has occurred. If not, program control is to be returned to the main program. If an error has occurred, the service routine must identify which error input is active and then pass control to its service routine.

Section 7.15

25. Write a level 2 service routine that will test the state of the ILLOP error bit. If ILLOP is set, control is to be passed to a service routine identified by the label CORRECT; otherwise, control is to be returned to the main program.
26. What are the functions of macro-ROM and macro-RAM?
27. If a MID opcode detected by the 99000 is >034A, at what address does its vector reside? What gets loaded into PC and WP?
28. What is the importance of the 99000's capability of implementing macrostore in external memory?

Section 7.16

29. Distinguish between user mode and supervisor mode.
30. Write a program for the level 2 service routine to test the PRIVOP bit of the internal error register and initiate a jump to the routine identified by label VIOLATE. If PRIVOP is not set, control is to be returned to the main program.

Section 7.17

31. Write a routine for the level 2 interrupt that will check to determine if an overflow has occurred. If yes, control is to be passed to a service routine identified by label OVRFLW. If no, control is to be returned to the main program.

BIBLIOGRAPHY

Laffitte, David, "New-Generation 16-Bit μPs—Fast and Function-Oriented," *Electronic Design*, Feb. 19, 1981.

Laffitte, David, and Reed Borie, "Separate Address Space Makes Context Switching a Snap for μPs," *Electronic Design*, June 11, 1981.

Laffitte, David, and Karl Guttag, "Fast On-Chip Memory Extends 16-Bit Family's Reach," *Electronics*, Feb. 24, 1981.

Laffitte, David, Karl Guttag, and Alan Loftus, "Architecture Eases Software's Migration into VLSI Hardware," *Electronic Design*, Feb. 5, 1981.

Texas Instruments Incorporated, *9900 Family Systems Design and Data Book*, 1st ed. Houston: Texas Instruments Incorporated, 1978.

Texas Instruments Incorporated, *TMS99105A and TMS99110A 16-Bit Microprocessors Preliminary Data Manual*. Houston: Texas Instruments Incorporated, Nov. 1982.

Texas Instruments Incorporated, *The TTL Data Book for Design Engineers*, 2nd ed. Dallas: Texas Instruments Incorporated, 1976.

Triebel, Walter A., *Integrated Digital Electronics*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1979.

Triebel, Walter A., and Alfred E. Chu, *Handbook of Semiconductor and Bubble Memories*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1982.

INDEX