COMPUTER ORGANISATION

(PGDCA03)

(PG - DIPLOMA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

UNIT - I

1. INTRODUCTION TO COMPUTERS

Structure

Objectives

At the end of the lesson you will be able to:

- Defination Of computer
- History Of Computers
- Discuss Classification Of Computers
- Discuss about I/P and I/O devices
- Explanation about storage devices
- Types of computer software

1.1 Introduction

In today's information age, computers are being used in every occupation. They are used by people of all age and profession, in their work as well as their leisure. This new social age has changed the basic concept of 'Computing'. Computing, in today's information age, is no more limited to computer programmers and computer engineers. Rather than knowing how to program a computer, most computer users simply need to understand how a computer functions so in this chapter I will be discussing with you about this versatile tool, why is it so powerful and useful, its history and you will also be briefed about the classification of computers.

1.2 What is a Computer?

A computer is an electronic machine that accepts information, stores it until the information is needed, processes the information according to the instructions provided by the user, and finally returns the results to the user. The computer can store and manipulate large quantities of data at very high speed, but a computer cannot think. A computer makes decisions based on simple comparisons such as one number being larger than another. Although the computer can help solve a tremendous variety of problems, it is simply a machine. It cannot solve problems on its own.

1.3 History of Computers

Since civilizations began, many of the advances made by science and technology have depended upon the ability to process large amounts of data and perform complex mathematical calculations. For thousands of years, mathematicians, scientists and businessmen have searched for computing machines that could perform calculations and analyze data quickly and efficiently. One such device was the abacus.

The abacus was an important counting machine in ancient Babylon, China, and throughout Europe where it was used until the late middle ages. It was followed by a series of improvements in mechanical counting machines that led up to the development of accurate mechanical adding machines in the 1930's. These machines used a complicated assortment of gears and levers to perform the calculations but they were far to slow to be of much use to scientists. Also, a machine capable of making simple decisions such as which number is larger was needed. A machine capable of making decisions is called a computer.

The first computer like machine was the Mark I developed by a team from IBM and Harvard University. It used mechanical telephone relays to store information and it processed data entered on punch cards. This machine was not a true computer since it could not make decisions.

In June 1943, work began on the world's first electronic computer. It was built at the University of Pennsylvania as a secret military project during World War II and was to be used to calculate the trajectory of artillery shells. It covered 1500 square feet and weighed 30 tons. The project was not completed until 1946 but the effort was not wasted. In one of its first demonstrations, the computer solved a problem in 20 seconds that took a team of mathematicians three days. This machine was a vast improvement over the mechanical calculating machines of the past because it used vacuum tubes instead of relay switches. It contained over 17,000 of these tubes, which were the same type tubes used in radios at that time.

The invention of the transistor made smaller and less expensive computers possible. Although computers shrank in size, they were still huge by today's standards. Another innovation to computers in the 60's was storing data on tape instead of punched cards. This gave computers the ability to store and retrieve data quickly and reliably.

1.4 Classification of Computers

- Mainframe Computers
- Minicomputers
- Microcomputers
- Supercomputers

1.4.1 Mainframe Computers

Mainframe Computers are very large, often filling an entire room. They can store enormous of information, can perform many tasks at the same time, can communicate with many users at the same time, and are very expensive. . The price of a mainframe computer frequently runs into the millions of dollars. Mainframe computers usually have many terminals connected to them. These terminals look like small computers but they are only devices used to send and receive information from the actual

NOTES

computer using wires. Terminals can be located in the same room with the mainframe computer, but they can also be in different rooms, buildings, or cities. Large businesses, government agencies, and universities usually use this type of computer.

1.4.2 Minicomputers

Minicomputers are much smaller than mainframe computers and they are also much less expensive. The cost of these computers can vary from a few thousand dollars to several hundred thousand dollars. They possess most of the features found on mainframe computers, but on a more limited scale. They can still have many terminals, but not as many as the mainframes. They can store a tremendous amount of information, but again usually not as much as the mainframe. Medium and small businesses typically use these computers

1.4.3 Microcomputers

Microcomputers are usually divided into desktop models and laptop models. They are terribly limited in what they can do when compared to the larger models discussed above because they can only be used by one person at a time, they are much slower than the larger computers, and they cannot store nearly as much information, but they are excellent when used in small businesses, homes, and school classrooms. These computers are inexpensive and easy to use. They have become an indispensable part of modern life.

1.4.4 Supercomputers

Supercomputers are very expensive and are employed for specialized application that requires immense amounts of mathematical calculations. For example, weather forecasting requires a supercomputer. Other uses of supercomputers include animated graphics, fluid dynamic calculations, nuclear energy research, and petroleum exploration. The chief difference between a supercomputer and a mainframe is that a supercomputer channels all its power into executing a few programs as fast as possible, whereas a mainframe uses its power to execute many programs concurrently.

1.5 Computer Tasks

- Input
- Storage
- Processing
- Output

When a computer is asked to do a job, it handles the task in a very special way.

1. It accepts the information from the user. This is called input.

2. It stored the information until it is ready for use. The computer has memory chips, which are designed to hold information until it is needed.
3. It processes the information. The computer has an electronic brain called the Central Processing Unit, which is responsible for processing all data and instructions given to the computer.
4. It then returns the processed information to the user. This is called output.

Every computer has special parts to do each of the jobs listed above. Whether it is a multi-million dollar mainframe or a thousand dollar personal computer, it has the following four components, Input, Memory, Central Processing, and Output. The central processing unit is made up of many components, but two of them are worth mentioning at this point. These are the arithmetic and logic unit and the control unit. The control unit controls the electronic flow of information around the computer. The arithmetic and logic unit, ALU, is responsible for mathematical calculations and logical comparisons.

1.6 Memory

The Internal storage areas in the computer represent as memory. The term memory identifies data storage that comes in the form chips, and the word storage is used for memory that exists on tapes disks. Moreover, the term memory is usually used as shorthand for physical memory, which refers to the actual chips capable of holding data. Some computers also use virtual memory, which expands physical memory onto a hard disks.

Every computer comes with a certain amount of physical memory, usually referred to as main memory or RAM. A computer that has 1 megabyte of memory, therefore, can hold about 1 million bytes (or characters) of information.

There are several different types of memory:

- RAM **(random-access memory):** This is the same as main memory. When used by itself, the term RAM refers to read and write memory; that is, you can both write data into RAM and read data from RAM. This is in contrast to ROM, which permits you only to read data. Most RAM is volatile, which means that it requires a steady flow of electricity to maintain its contents. As soon as the power is turned off, whatever data was in RAM is lost.
- ROM **(read-only memory):** Computers almost always contain a small amount of read-only memory that holds instructions for starting up the computer. Unlike RAM, ROM cannot be written to.
- PROM **(programmable read-only memory):** A PROM is a memory chip on which you can store programs. But once the PROM has been used, you cannot wipe it clean and use it to store something else. Like ROMs, PROMs are non-volatile.

- EPROM **(erasable programmable read-only memory):** An EPROM is a special type of PROM that can be erased by exposing it to ultraviolet light.
- EEPROM **(electrically erasable programmable read-only memory):** An EEPROM is a special type of PROM that can be erased by exposing it to an electrical charge.

1.7 Central Processing Unit (CPU)

The central processing unit is one of the two most important components of your microcomputer. It is the electronic brain of your computer. In addition to processing data, it controls the function of all the other components. The most popular microprocessors in IBM compatible computers are made by Intel. The generations of microprocessors are listed below.

1981	8088
1984	80286
1987	80386
1990	80486
1993	Pentium
1996	P-1
2002	P-4

1.8 Telecommunications

Telecommunications means that you are communicating over long distances usually using phone lines. This enables you to send data to and receive data from another computer that can be located down the street, in another town, or in another country. Telecommunications requires a communication device called a modem, which connects your computer to a standard phone jack. A modem converts the digital signals that your computer uses into analog signals that can be transmitted over the phone lines. To use a modem, you must also have communication software to handle the transmission process.

1.9 Computer Software

Software is a generic term for organized collections of computer data and instructions, often broken into two major categories: system software that provides the basic non-task-specific functions of the computer, and application software which is used by users to accomplish specific tasks.

1.9.1 System Software

System software is responsible for controlling, integrating, and managing the individual hardware components of a computer system so that other software and the users of the system see it as a functional unit without having to be concerned with the low-level details such as transferring data from memory to disk, or rendering text onto a display. Generally, system software consists of an operating system and some fundamental utilities such as disk formatters, file managers, display managers, text editors, user authentication (login) and management tools, and networking and device control software.

1.9.2 Application software

Application software, on the other hand, is used to accomplish specific tasks other than just running the computer system. Application software may consist of a single program, such as an image viewer; a small collection of programs (often called a software package) that work closely together to accomplish a task, such as a spreadsheet or text processing system; a larger collection (often called a software suite) of related but independent programs and packages that have a common user interface or shared data format, such as Microsoft Office, which consists of closely integrated word processor, spreadsheet, database, etc.; or a software system, such as a database management system, which is a collection of fundamental programs that may provide some service to a variety of other independent applications.

1.9.3 Program Software

Program software is software used to write computer programs in specific computer languages. Software is created with programming languages and related utilities, which may come in several of the above forms: single programs like script interpreters, packages containing a compiler, linker, and other tools; and large suites (often called Integrated Development Environments) that include editors, debuggers, and other tools for multiple languages.

1.10 Emerging Trends

The components of a computer are connected by using buses. A bus is a collection of wire that carry electronic signals from one component to another. There are standard buses such as Industry Standard Architecture (ISA), Extended Industry Standard Architecture (EISA), Micro-Channel Architecture (MCA), and so on. The standard bus permits the user to purchase the components from different vendors and connect them easily.

The various input and output devices have a standard way of connecting to the CPU and Memory. These are called interface standards. Some popular interface standards are the RS-232C and Small Computer System Interconnect (SCSI). The places where the standard interfaces are provided are called ports.

NOTES

1.11 Data Representation

1.11.1 Bits and Bytes

Data in Computers are represented using only two symbols '0' & '1'. These are called "Binary digits" (or) "BITS" for short. A set of 8 bits is called a byte and each byte stores one character. 2^n unique strings are represented using n bits only. For example, Using 2 bits we can represent 4 = (2^2) unique strings as 00, 01, 10, 11. ASCII (American Standards Code for Information Interchange) codes are used to represent each character. The ASCII code includes codes for English Letters (Both Capital & Small), decimal digits, 32 special characters and codes for a number of symbols used to control the operation of a computer which are non-printable.

1.11.2 Binary Numbers

Binary numbers are formed using the positional notation. Powers of 2 are used as weights in the binary number system. A binary number 10111 has a decimal value equal to $1X2^4$ $+0X2^3$ $+1X2^1$ $+1X2^0$ =23. A decimal number is converted into an equivalent binary number by dividing the number by 2 and storing the remainder as the least significant bit of the binary number.

1.11.3 Hexadecimal Numbers

High valued binary numbers will be represented by a long sequence of 0's and 1's. A more concise representation is using hexadecimal representation. The base of the hexadecimal system is 16 and the symbols used in this system are 0,1,2,4,5,6,7,8,9,A,B,C,D,E,F. Strings of 4 bits have an equivalent hexadecimal value. For example, 6B is represented by 0110 1011 or 110 1011, 3E1 is represented by 0011 1110 0001 or 11 1110 0001 and 5DBE34 is represented by 101 1101 1011 1110 0011 0100. Decimal fractions can also be converted to binary fractions.

1.11.4 Parity Check Bit

Errors may occur while recording and reading data and when data is transmitted from one unit to another unit in a computer Detection of a single error in the code for a character is possible by introducing an extra bit in its code. This bit, know as the parity check bit, is appended to the code. The user can set the parity bit either as even or odd. the user chooses this bit so that the total number of ones ('1') in the new code is even or odd depending upon the selection. If a single byte is incorrectly read or written or transmitted, then the error can be identified using the parity check bit.

1.12 Input Devices

1.12.1 Key Board

The most common input device is the Keyboard. It is used to input letters, numbers, and commands from the user.

1.12.2 Mouse

Mouse is a small device held in hand and pushed along a flat surface. It can move the cursor in any direction. In a mouse a small ball is kept inside and the ball touches the pad through a hole at the bottom of the mouse. When the mouse is moved, the ball rolls. This movement of the ball is converted into electronic signals and sent to the computer. Mouse is very popular in the modern computers that use Windows and other Graphical User Interface (GUI) applications.

1.12.3 Magnetic Ink Character Recognition (MICR)

In this method, human readable characters are printed on documents such in this method, human readable characters are printed on documents such as cheque using special magnetic ink. The cheque can be read using a special input unit, which can recognize magnetic ink characters. This method eliminates the need to manually enter data from cheques into a floppy. Besides saving time, this method ensures accuracy of data entry and improves security.

1.12.4 Optical Mark Reading and Recognition (OMR)

In this method, special pre-printed forms are designed with boxes which can be marked with a dark pencil or ink. Such a document is read by a document reader, which transcribes the marks into electrical pulses which are transmitted to the computer. These documents are applicable in the areas where responses are one out of a small number of alternatives and the volume of data to be processed is large. For example:

- Objective type answer papers in examinations in which large number of candidates appear.
- Market surveys, population survey etc.,
- Order forms containing a small choice of items.
- Time sheets of factory employees in which start and stop times may be marked.

The advantage of this method is that information is entered at its source and no further transcription is required.

1.12.5 Optical Character Recognition (OCR)

An optical scanner is a device used to read an image, convert it into a set of 0’s and 1’s and store it in the computer’s memory. The image may be hand-written document, a typed or a printed document or a picture.

1.12.6 Bar Coding

In this method, small bars of varying thickness and spacing are printed on packages, books, badges, tags etc., which are read by optical readers and converted to electrical pulses. The patterns of bars are unique an standardized. For example, each grocery product has been given unique 10-digit code and this is represented in bar code form on every container of this product.

1.12.7 Speech Input Unit

A unit, which takes spoken words as its input, and converts them to a form that can be understood by a computer is called a speech input unit. By understanding we mean that the unit can uniquely code (as a sequence of bits) each spoken word, interpret the word and initiate action based on the word.

1.13 Output Devices

1.13.1 Monitor or Video Display Unit (VDU)
Monitor or Video Display Unit (VDU) Monitors provide a visual display of data. It looks like a television. Monitors are of different types and have different display capabilities. These capabilities are determined by a special circuit called the Adapter card. Some popular adapter cards are,

- Color Graphics Adapter (CGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Super Video Graphics Array (SVGA)

1.14 Summary

A computer system may be viewed from the perspective of end users, system and application programmers and hardware designers. In this chapter, we provided a brief historical background for the development of computer systems, starting from the first recorded attempt to build a computer, Mainframe Computers, Minicomputers, Microcomputers, and Supercomputers. We then provided a discussion on the memory and Input and out put devices.

1.15 Key words

Mainframe Computers: mainframe is a high-performance computer used for large-scale computing purposes that require greater availability and security than a smaller-scale machine can offer.

Minicomputers: A mid-sized computer, usually fitting within a single cabinet about the size of a refrigerator that has less memory than a mainframe.

NOTES

Microcomputers: A microcomputer is a computer with a microprocessor as its central processing unit. Another general characteristic of these computers is that they occupy physically small amounts of space when compared to mainframe and minicomputers.

Supercomputers: The fastest type of computer. Supercomputers are very expensive and are employed for specialized applications that require immense amounts of mathematical calculations.

Memory: Memory is an organism's ability to store, retain, and subsequently retrieve information.

RAM: Random-access memory (usually known by its acronym, RAM) is a form of computer data storage. The word *random* thus refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether or not it is related to the previous piece of data.

ROM: Computers almost always contain a small amount of read-only memory that holds instructions for starting up the computer. Unlike RAM, ROM cannot be written to.

EPROM: An EPROM is a special type of PROM that can be erased by exposing it to ultraviolet light.

CPU: It is the electronic brain of computer. In arithmetic processing data, it controls the function of all the other components.

MICR: Magnetic Ink Character Recognition

OMR: Optical Character Recognition

1.16 Exercise:

1. When u switch on your computer which software you see first and what is the utility of that software.

2. Suppose on fine day you are working on your computer and power goes off, again u switch on your computer, what type of booting is done by that computer.

3. Write the essential parts of your computer system without which u cant work and also list that parts which are optional

4. How many types of storage are normally there in storage unit of a computer system? Justify the need for each storage type. Explain them.

5. What are the basic components of the CPU of a computer system? Describe the roles of each of the components in the functioning of a computer system.

NOTES

6. Suppose an entrance exam is held and thousands of students appeared in that exam, which device u will use to evaluate the answer sheets and why?

7. Hardware and software are like two sides of a coin. Do you agree or disagree, Give reasons.

1.17 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation And Architecture By: William Stallings Prentice —Publications

4. Computer Architecture and Organisation By: J.P.Hayes

2. REGISTER TRANSFER LOGIC

Structure

Objectives

At the end of the lesson you will be able to:

- Describe the register transfer language
- Define what is a register transfer
- Define what are bus and memory transfers

2.1 Introduction

A digital system is a sequential logic system constructed with flip-flops and gates. It was shown in previous chapters that a sequential circuit can be specified by means of a state table. To specify a large digital system with a state table would be very difficult, if not impossible, because the number of states would be prohibitively large. To overcome this difficulty, digital system are invariably designed using a modular approach. The system is partitioned into modular subsystems, each of which performs some functions task. The modules are constructed from such digital functions as registers, counters, decoders, multipliers, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system. A typical digital system module would be the processor unit of a digital computer.

The interconnection of digital functions to form a digital system module cannot be described by means of combinational or sequential logic techniques. These techniques were developed to describe a digital system at the gate and flip-flop level and are not suitable for describing the system at the digital function level. To describe a digital system in terms of functions such as adders, decoders, and registers, it is necessary to employ a higher-level mathematical notation. The register-transfer logic method fulfills this requirements. In this method, the registers are selected to be the primitive components in the digital system, rather than the gates and flip-flops as in sequential logic. In this way it is possible to describe, in a concise and precise manner, the information flow and processing tasks among the data stored in the registers. The register-transfer logic method uses a set of expressions and statements which resemble the statements used in programming languages. This notation provides the necessary tools for specifying a prescribed set of interconnections between various digital functions. An important characteristic of the register transfer logic method of presentation is that it is closely related to the way people would prefer to specify the operations of a digital system.

The binary information stored in registers may be binary numbers, binary coded decimal numbers, alphanumeric characters, control information, or any other binary-coded information. The operations that are performed on the data stored in registers depend on the type of data encountered. Numbers are manipulated with arithmetic operations, whereas control information is usually manipulated with logic opera tions such as setting and clearing specified bits in the register. The operations performed on the data stored in registers are called micro operations. The control functions that initiate the sequence of operations consist of timing signals that sequence the operations one at a time. Certain conditions which depend on results of previous operations may also determine the state of control functions. A control function is a binary variable that, when in one binary state, initiates an operation and, when in the other binary state, inhibits the operation. operation that can be performed in parallel during one clock pulse period. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, add, clear, and load. A counter with paralled load is capable of performing the microoperations increment and load. A bidirectional shift register is capable of performing the shift-right and shift-left microoperations. A binary parallel adder is useful for implementing the add microoperation on the contents of two registers that hold binary numbers. A microoperation requires only one clock pulse for execution if the operation is done in parallel. In serial computers, a microoperation requires a number of pulse equal to the word time in the system. This is equal to the number of bits in the shift registers that transfer the information serially while a microoperation is being executed.

2.2 Register Transfer Language

A digital system is an interconnection of digital hardware modules that accomplish a specific information-processing task. Digital systems vary in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called *microoperations*. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. For example, a counter with parallel load is capable of performing the microoperations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The basic components of this method are those that describe a digital system from the operation al level. The operation of a digital system is best described by specifying :

1. The set of registers it contains and their function.
2. The binary-coded information stored in the registers.
3. The sequence of microoperations performed on the binary information stored in the registers.
4. The control that initiates the sequence of microoperations.

It is possible to specify the sequence of microoperations in a computer by explaining every operation in words, but this procedure usually involve a lengthy descriptive explanation. It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers. The use of symbols instead of a narrative explanation provides an organized and concise manner for listing the microoperation sequences in registers and the control functions that initiate them.

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages. A programming language is a procedure for writing symbols to specify a given computational process. Similarly, a natural language such as English is a system for writing symbols and combining them into words and sentences for the purpose of communication between people. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible, so it should not take very long to memorize. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation introduced in this lesson will be utilized in subsequent lessons to specify the register transfers, the microoperations and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

2.3 Register Transfer

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a memory address register and is designated by the name *MAR.* Other designations for registers are *PC* (for program counter), *IR* (for instruction register, and *R* (for processor register). The individual flip-flops in an *n*-bit register are numbered in sequence from 0 through *n* - 1, starting from 0 in the rightmost position and increasing the numbers toward the left. Figure 2-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig. 2-l(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H(for* high byte). The name of the 16-bit register is *PC.* The symbol *PC(0-7)* or *PC(L)* refers to the low-order byte and *PC(8-15)* or *PC[H]* to the high-order byte.

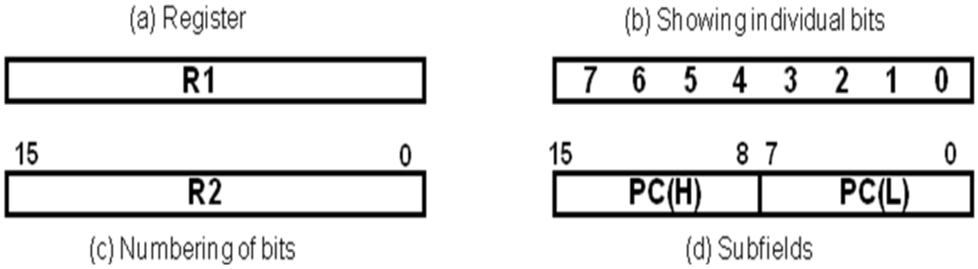


Figure 2-1 Block diagram of register.

Information transfer from one register to another is designated in symbolic form by means of a replacement operator.

$$R2 \leftarrow R1$$

The statement denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an *if-then* statement.

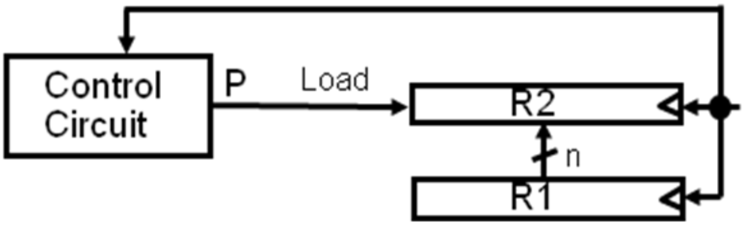
If (P = 1) *then* (R2 ← R1)

where P is a control signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a *control function.* A control function is a Boolean variable equal to 1 or 0. The control function is included in the statement as follows:

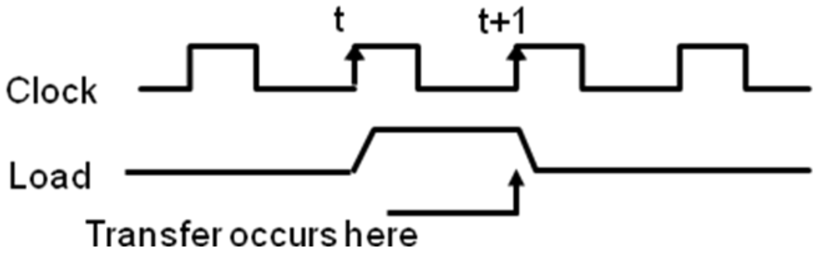
P: R2 ← R1

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if *P=1*

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 2-2 shows the block diagram that depicts the transfer from R1 to R2*.* The *n* outputs of register R1are connected to the *n* inputs of register *R2.* The letter *n* will be used to in any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register R2 has a load input activated by the control variable P*.* It is assumed that the control variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, *P* is activated in the control section by the rising edge of a clock pulse at time *t.* The next positive transition of the clock at time *t* + 1 finds the load input active and the data inputs of *R2* are then loaded into the register in parallel. P may go back to 0 at time *t* + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active.



(a) Block diagram



(b) Timing diagram

Figure 2-2 Transfer from R1 to R2 when P=1

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as *P* becomes active just after time *t,* the actual transfer does not occur until the register is triggered by the next positive transition of the clock at time *t* + 1.

The basic symbols of the register transfer notation are listed in Table 2-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: \ R2 \leftarrow R1, R1 \leftarrow R2$$

denotes an operation exchanges the contents of two registers during one common clock pulse provided that *T* = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

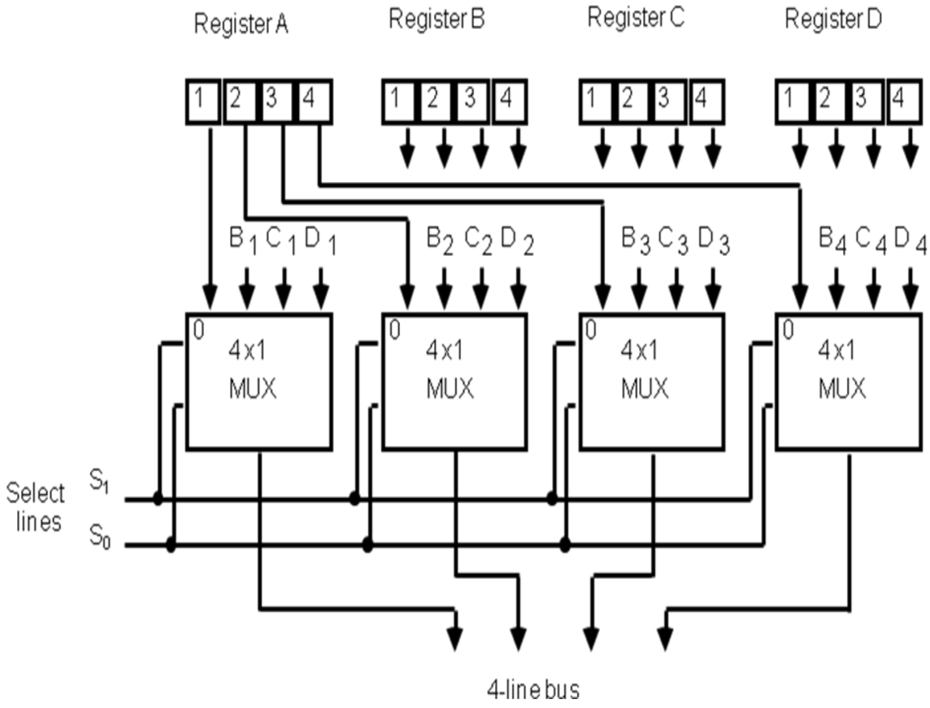
TABLE 2-1 Basic Symbols for Register Transfers

Symbols	**Description**	**Examples**
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses **( )**	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon **:**	Denotes termination of control function	P**:**
Comma **,**	Separates two micro-operations	A ← B**,** B ← A

2-4 Bus and Memory Transfers

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a common bus system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig. 2-3. Each register has four bits, numbered 0 through 3. The bus consists of four 4X1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, .S_1 and S_0. In order not to complicate the diagram with 16 lines crossing each other, we use labels to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register *A* is connected to input 0 of MUX 1 because this input is labeled A_1. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.



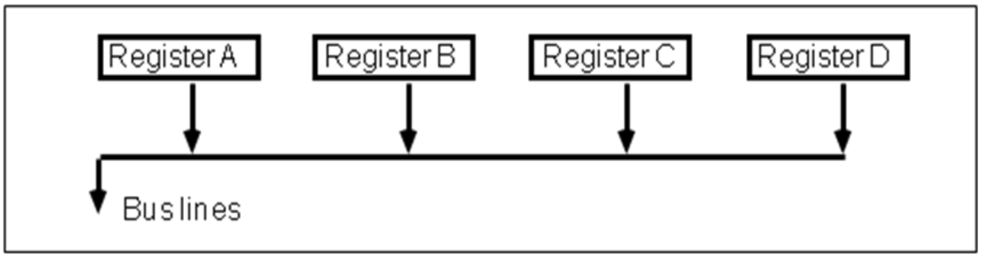


Figure 2-3 Bus system four registers

The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When S_1S_0= 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register *A* since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register *B is* selected if S_1S_0= 01, and so on. Table 2-2 shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

TABLE 2-2 Function Table for Bus of Fig. 2-3

S_1	S_0	**Register selected**
0	O	A
0	1	B
1	0	C
1	1	D

In general, a bus system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus. The number of multiplexers needed to construct the bus is equal to *n,* the number of bits in each register. The size of each multiplexer must be *k X* 1 since it multiplexes *k* data lines. For example, a common bus for eight registers of 16 bits each requires 16 multiplexers, one for each line in the bus. Each multiplexer must have eight data input lines and three selection lines to multiplex one significant bit in the eight registers.

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement.

When the bus is includes in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, R1 \leftarrow BUS$$

The content of register *C is* placed on the bus, and the content of the bus is loaded into register *R1* by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

From this statement the designer knows which control signals must be activated to produce the transfer through the bus.

2-4-1 Three-State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance. Three-state gates may perform any conventional logic, such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in Fig. 2-4. It distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

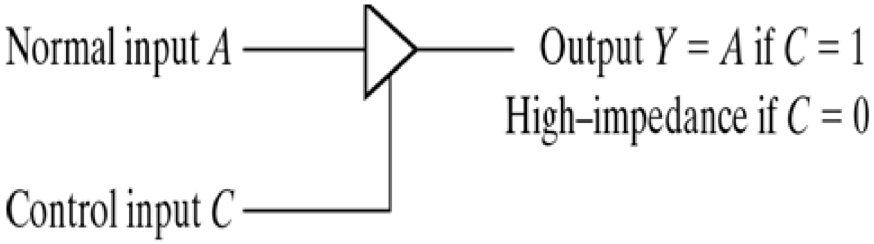


Figure 2-4 Graphic symbol for three-state buffer

The construction of a bus system with three-state buffers is demonstrated in Fig. 2-5. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

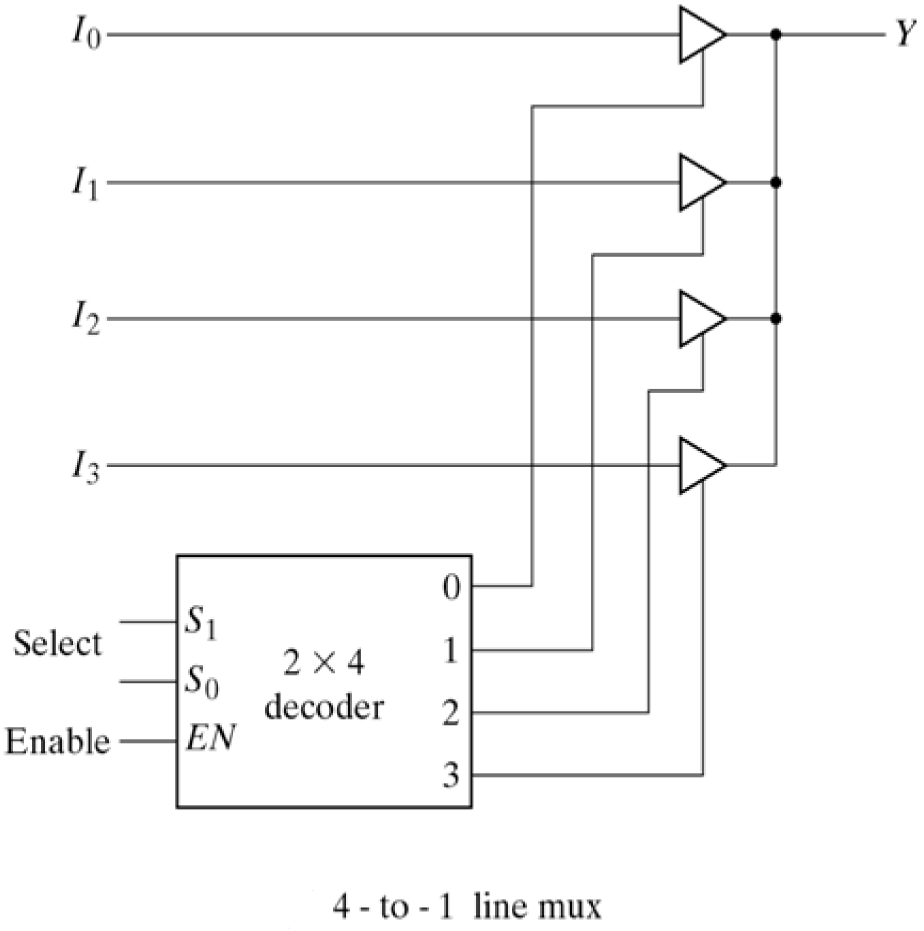


Figure 2-5 Bus line with three state buffers

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Fig. 2-5 is another way of constructing a 4 X 1 multiplexer since the circuit can replace the multiplexer in Fig. 2-3

To construct a common bus for four registers of *n* bits each using three-state buffers, we need *n* circuits with four buffers in each as shown in Fig. 2-5. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of *n* lines. Only one decoder is necessary to select between the four registers.

2-4-2 Memory Transfer

The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter *M.* The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of *M* when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter *M.*

Consider a memory unit that receives the address from a register, called the address register, symbolized by *AR.* The data are transferred to another register, called the data register, symbolized by *DR.* The read operation can be stated as follows:

$$\text{Read: } DR \leftarrow M[AR]$$

This causes a transfer of information into *DR* from the memory word *M* selected by the address in *AR.*

The write operation transfers the content of a data register to a memory word *M* elected by the address. Assume that the input data are in register R1 and the address is in *AR.* The write operation can be stated symbolically as follows:

$$\text{Write: } M[AR] \leftarrow R1$$

2-5 Summary

In this lesson, we have discussed in detail about the register transfer longuage and register transfer of the system.after this we have discussed in detail bus and memory transfer and their implementaion in hardware using simple logic circuites.

2-6 Keywords

Microoperation: In computer central processing units, micro-operations, also known as a micro-ops or μops, are detailed low-level instructions used in some designs to implement complex machine instructions

NOTES

Register: In computer architecture, a processor register is a small amount of storage available on the CPU whose contents can be accessed more quickly than storage available elsewhere.

Register transfer language: In computer system, register transfer language (RTL) is a term used to describe a kind of intermediate representation (IR) that is very close to assembly language, such as that which is used in a compiler.

Control function: In computer system, a control operation (control function) is an operation that affects the recording, processing, transmission, or interpretation of data.

Common bus: Computer *System* for controlling access to a *common bus* in a *computer system.*

High impedance: It means that the signal is neither driven to a logical high nor low level - hence "tri-stated". Such a signal can be seen as an open circuit.

Buffer: In computing, a buffer is a region of memory used to temporarily hold data while it is being moved from one place to another.

2-7 Exercise:

1. Explain the Register transfer logic?

2. Explain Memory transfer?

3. Explain the inter register transfer?

4. An instruction at address 021 in the basic computer has I =0, an operation code of the AND instruction, and an address part equal to 083 (all numbers are hexadecimal). The memory word at address 083 contains the operand B8F2 and the content of AC is A937. Go over the instruction cycle and determine the contents of the following registers at the end of the execute phase: PC, AR, DR, AC and IR. Repeat the problem six more times starting with an operation code of another memory reference instruction.

5. Show the contents in hexadecimal of registers PC, AR, DR, IR and SC of the basic computer when ISZ indirect instruction is fetched from memory and executed. The initial content of PC is 7FF. the content of memory at address 7FF is EA9F. The content of memory at address A9F is 0C35. The content of memory at address C35 is FFFF. Give the answer in a table with five columns, one for each register and a row for each timing signal. Show the contents of the registers after the positive transaction of each clock pulse.

6. A computer uses a memory of 65, 536 words with eight bits in each word. It has the following registers: PC, AR, TR (16 bits each), and AC, DR, IR (eight bits each). A memory reference instruction consists of three

words: an 8-bit operation-code (one word) and a 16-bit address (in the next two words). All operands are eight digits. There is no direct bit.

a) Draw a block diagram of the computer showing the memory and registers (Do not use a common bus).

b) Draw a diagram showing the placement in memory of a typical three-word instruction and corresponding 8 bit operand.

c) List the sequence of micro operations for fetching a memory reference instruction and then placing the operand in DR start from timing signal T0

2-8 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes.

3. MICRO-OPERATIONS

Structure

Objectives

At the end of the lesson you will be able to:

- Discuss about various arithmetic microoperations
- Discuss about various logic microoperations
- Discuss about various shift microoperations
- Discribe various arithmetic logic shift unit

3-1 Arithmetic Microoperations

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computer are classified into four categories:

1. Register transfer microoperations transfer binary information from on register to another.

2. Arithmetic microoperations perform arithmetic operation on numeric data stored in registers.

3. Logic microoperations perform bit manipulation operations on non numeric data stored in registers.

4. Shift microoperations perform shift operations on data stored registers.

The register transfer microoperation was introduced in lesson 2-2. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. In this section we introduce a set of arithmetic microoperation. In the next two sections we present the logic and shift microoperations.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement.

$$R3 \leftarrow R1 + R2$$

specifies an *add* microoperation. It states that the contents of register *R1* added to the contents of register *R2* and the sum transferred to register *R3*. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 3-1. Subtraction is most often implemented through complementation and addition. Instead of using the *operation* minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow R1 + \overline{R2} + 1$$

$\overline{R2}$ is the symbol for the 1's complement of *R2.* Adding 1 to the 1's complement produces the 2's complement. Adding the contents *of R1* to the 2's complement of R2 is equivalent to *R1 - R2.*

TABLE 3-1 Arithemetic Microoperations

Symbol designation	Description
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow \overline{R2}$	Complement the content of R2 (1’s complement)
$R2 \leftarrow \overline{R2} + 1$	2’s complement the contents of R2 (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	R1 plus the 2’s complement of R2 (subtraction)

$R1 \leftarrow \mathbf{R1+1}$	Increment the contents of R1 by one
$R1 \leftarrow \mathbf{R1-1}$	Decrement the contents of R1 by one

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations of multiply and divide are not listed in Table 3-1. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In most computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations.

3-1-1 Binary Adder

To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder. The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 3-1 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of *A* and the addend bits of *B* are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_o and the output carry is C_4. The S outputs of the full-adders generate the required sum bits.

NOTES

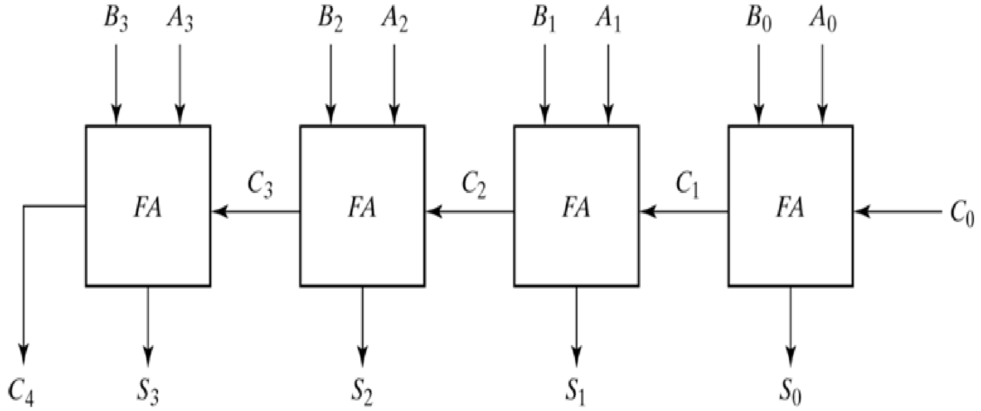


Figure 3-1 4 bit binary adder

An *n*-bit binary adder requires *n* full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder. The *n* data bits for the *A* inputs come from one register (such as R1), and the *n* data bits for the *B* inputs come from another register (such as *R2).* The sum can be transferred to a third register or to one of the source registers (*R1 or R2),* replacing its previous content.

3-1-2 Binary Adder-Subtractor

Remember that the subtraction *A - B* can be done by taking the 2's complement of *B* and adding it to *A.* The 2’s complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits. The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.

The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full-adder. A 4-bit adder-subtractor circuit is shown in Fig. 3-2. The mode input *M* controls the operation. When $M = 0$ the circuit is an adder and when $M = 1$ the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of *B.* When $M = 0$, we have $B \oplus 0 = B$. The full-adders receive the value of *B,* the input carry is 0, and the circuit performs *A* plus *B.* When *M* = we have $B \oplus 1 = B'$ and $C_o = 1$, The *B* inputs are all complemented and a 1 added through the input carry. The circuit performs the operation *A* plus the 2's complement of *B.* For unsigned numbers, this gives *A - B* if $A \geq B$ or the 2's complement of *(B - A)* if $A < B$. For signed numbers, the result is *A - B* provided that there is no overflow.

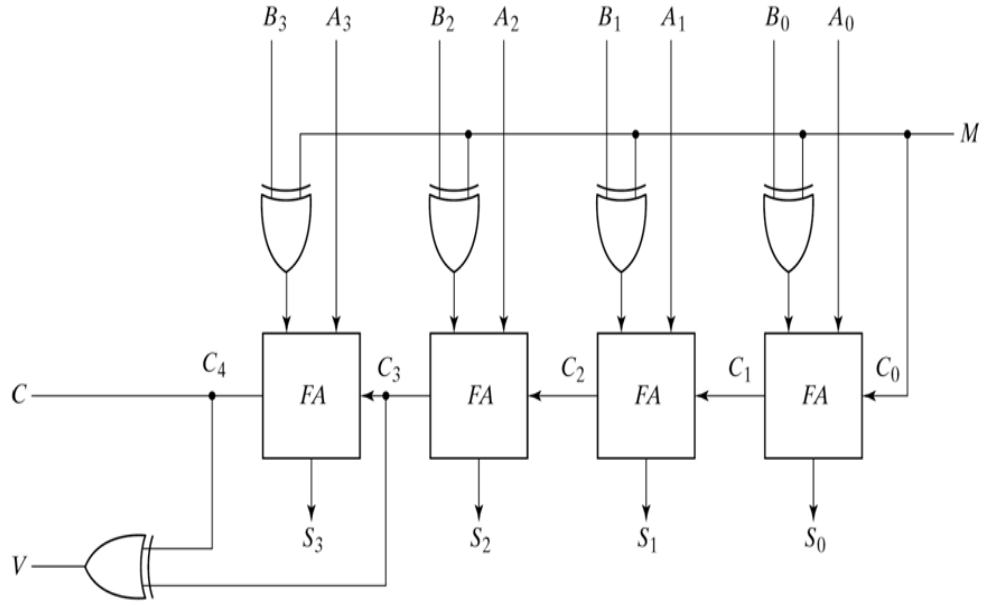


Figure 3-2 4-bit adder-subtractor

3-1-3 Binary Incrementer

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig. 3-3. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from A_0 through A_3, adds one to it, and generates the incremented output in S_o through S_3. The output carry C_4 will be 1 only after incrementing binary 1111. This also causes outputs s_0 through S_3 to go to 0.

The circuit of Fig. 3-3 can be extended to an *n-bit* binary incrementer by extending the diagram to include *n* half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

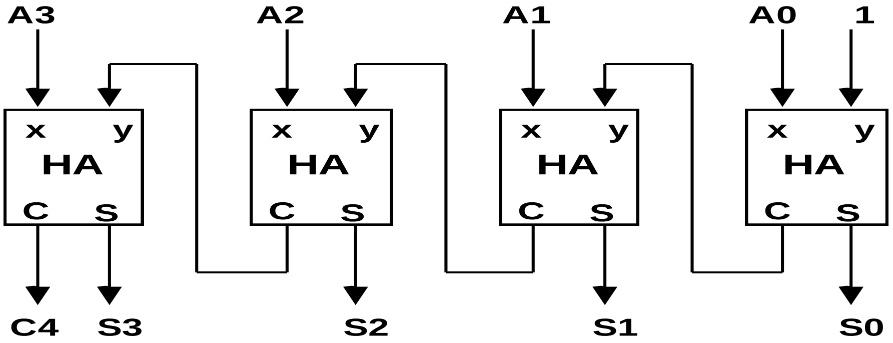


Figure 3-3 4 bit binary incrementer

3-1- 4 Arithmetic Circuit

The arithmetic microoperations listed in Table 3-1 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig. 3-4. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs *A* and *B* and a 4-bit output *D.* The four inputs from *A* go directly to the *X* inputs of the adder. Each of the four inputs from *B* are connected to the data inputs multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, S_1 and S_o. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries a connected from one stage to the next.

The output of the binary adder is calculated from the following arithmetic sum:

$$D = \mathbf{A} + \mathbf{Y} + C_{in}$$

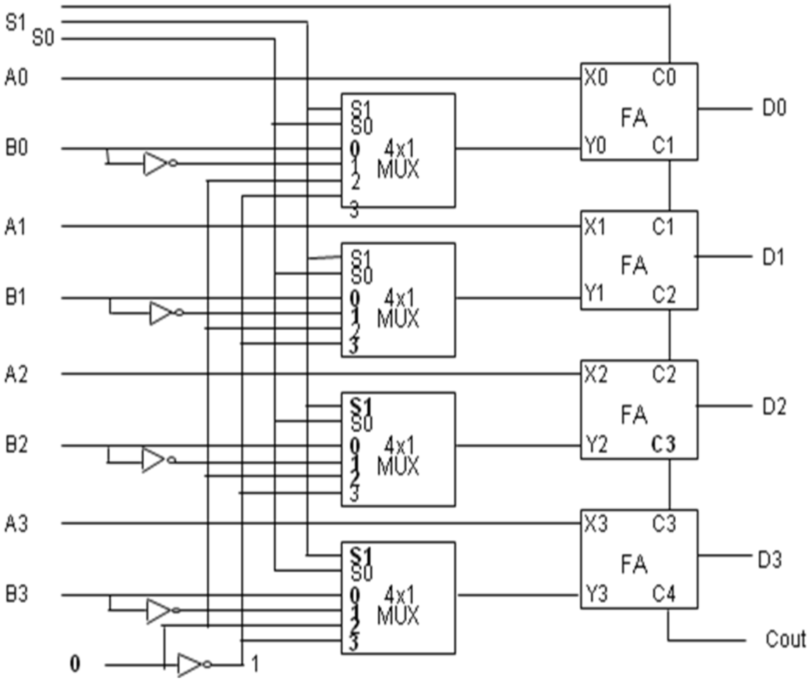


Figure 3-4 4-bit arithmetic circuit

where *A* is the 4-bit binary number at the *X* inputs and *Y* is the 4-bit binary number at the *Y* inputs of the binary adder. C_{in} is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table 3-2.

TABLE 3-2 Arithmetic Circuit Functions Table

Select			Input	Output	
S_0	S_1	C_{in}	Y	D= A + Y + C_{in}	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	$\overline{B}$	D = A + $\overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	D = A + $\overline{B}$ + 1	Subtract
1	0	0	0	D = A	Transfer A

1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

When S_1S_o = 00,the value of *B* is applied to the Y inputs of the adder. If C_{in} = 0, the output *D = A + B.* If C_{in} = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

When S_1S_o = 01, the complement of *B is* applied to the Y inputs of the adder. If C_{in} = 1, then *D = A +* $\overline{B}$ + 1. This produces *A* plus the 2's complement of *B,* which is equivalent to a subtraction of *A - B.* When C_{in} = 0, then *D = A +* $\overline{B}$*.* This is equivalent to a subtract with borrow, that is, *A - B – 1.*

When $S_1S_o{}^{=}$ 10, the inputs from *B* are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes *D = A* + 0 + C_{in}. This gives *D = A* when C_{in} = 0 and *D = A* + 1 when C_{in} = 1. In the first case we have a direct transfer from input *A* to output *D.* In the second case, the value of A is incremented by 1.

When S_1S_o =11, all 1's are inserted into the Y inputs of the adder produce the decrement operation *D = A* - 1 when C_{in} = 0. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number *A* to the 2's complement of 1 produces *F* = *A* + 2's complement of 1 = *A* - 1. When C_{in} = 1, then *D* = *A* -1 + 1= *A,* which causes a direct transfer from input *A* to output *D.* Note that the microoperation *D = A is* generated twice, so there are only seven microoperations in the arithmetic circuit.

3-2 Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers *R1* and *R2 is* symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable *P* = 1. As a numerical example, assume that each register has four bits. Let the content of *R1* be 1010 and the content of *R2* be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

1010 Content of *R1*
$\underline{1100}$ Content of *R2*
0110 Content of *R1* after *P* = 1

The content of *R1,* after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in *R2* and previous values of R1. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol ∨ be used to denote an OR microoperation and the symbol ∧ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation. For example, in the statement

$$P + Q: R1 \leftarrow R2 + R3\ ,\ R4 \leftarrow R5 \vee R6$$

the + between P and Q is an OR operation between two binary variables of a control function. The + between *R2* and -R3 specifies an add microoperation. The OR microoperation is designated by the symbol V between registers *R5* and R6.

3-2-1 List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 3-3. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the two variables *x* and *y.* Note that the functions are determined from the 16 binary combinations that can be assigned to *F.*

TABLE 3-3 Truth Table for 16 Function of Two Variables

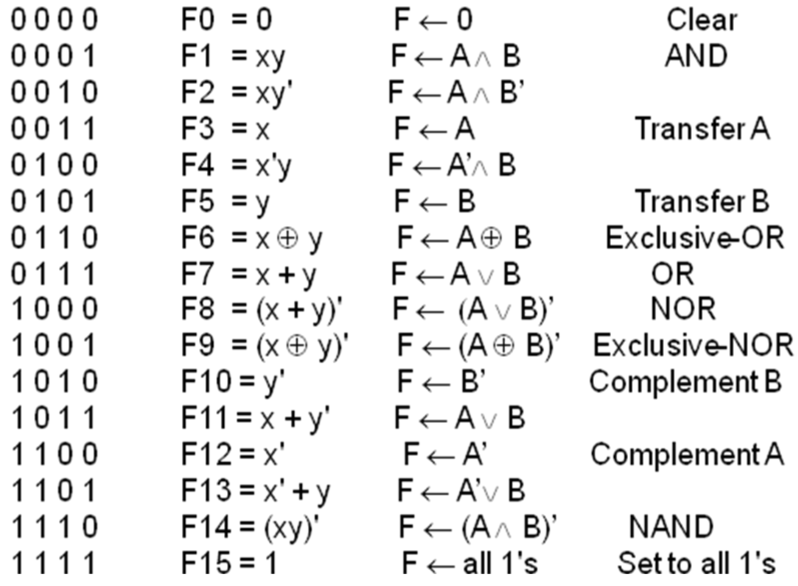
X	Y	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15

NOTES

0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 Boolean functions of two variables *x* and *y* are expressed in algebraic form in the first column of Table 3-4. The 16 logic microoperations are derived from these functions by replacing variable *x* by the binary content of register *A* and variable *y* by the binary content of register *B.* It is important to realize that the Boolean functions listed in the first column of Table 3-4 represent a relationship between two binary variables *x* and *y.* The logic microoperations listed in the second column represent a relationship between the binary content of two registers *A* and *B.* Each bit of the register is treated as a binary variable and the microoperation is performed on the string of bits stored in the registers.

TABLE 3-4 Sxteen Logic Microoperations



Binary value	Boolean Function	Microoperations	Name
0 0 0 0	F0 = 0	F ← 0	Clear
0 0 0 1	F1 = xy	F ← A ∧ B	AND
0 0 1 0	F2 = xy'	F ← A ∧ B'	
0 0 1 1	F3 = x	F ← A	Transfer A
0 1 0 0	F4 = x'y	F ← A'∧ B	
0 1 0 1	F5 = y	F ← B	Transfer B
0 1 1 0	F6 = x ⊕ y	F ← A⊕ B	Exclusive-OR
0 1 1 1	F7 = x + y	F ← A ∨ B	OR
1 0 0 0	F8 = (x + y)'	F ← (A ∨ B)'	NOR
1 0 0 1	F9 = (x ⊕ y)'	F ← (A⊕ B)'	Exclusive-NOR
1 0 1 0	F10 = y'	F ← B'	Complement B
1 0 1 1	F11 = x + y'	F ← A ∨ B	
1 1 0 0	F12 = x'	F ← A'	Complement A
1 1 0 1	F13 = x' + y	F ← A'∨ B	
1 1 1 0	F14 = (xy)'	F ← (A ∧ B)'	NAND
1 1 1 1	F15 = 1	F ← all 1's	Set to all 1's

3-2-2 Hardware Implementation

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four-AND, OR, XOR (exclusive-OR), and complement-from which all others can be derived.

Figure 3-5 shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_o choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript *i.* For a logic circuit with *n* bits, the diagram must be repeated *n* times for *i*= 0, 1, 2,..., *n* — 1. The selection variables are applied to all stages. The function table in Fig. 3-5(b) lists the logic microoperations obtained for each combination of the selection variables.

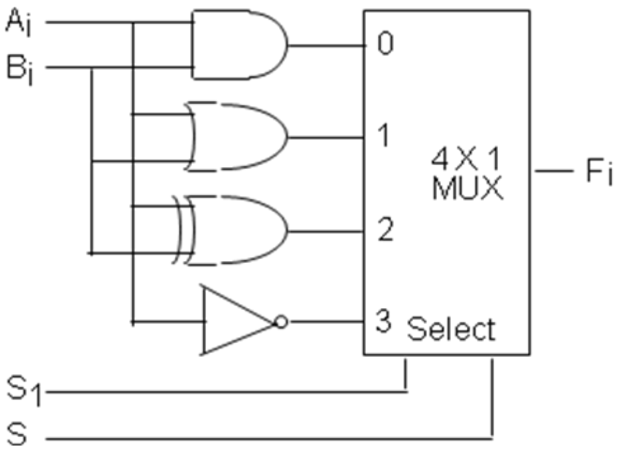


Figure 3-5 (a) One storage of logic circuit

S1	S0	Output	Operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

Figure 3-5 (b) Function table

3-2-3 Some Applications

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by *A)* are manipulated by logic microoperations as a function of the bits of another register (designated by *B).* In a typical application, register *A is* a processor register and the bits of register *B* constitute a logic operand extracted from memory and placed in register *B.*

The *selective-set* operation sets to 1 the bits in register *A* where there are corresponding 1's in register *B.* It does not affect bit positions that have 0's in *B.* The following numerical example clarifies this operation:

The *selective-set* operation sets to 1 the bits in register *A* where there are corresponding 1's in register *B.* It does not affect bit positions that have 0's in *B.* The following numerical example clarifies this operation:

1010	*A* before
$\underline{1100}$	*B* (logic operand)
1110	*A after*

The two leftmost bits of *B* are 1's, so the corresponding bits *of A* are set to 1. One of these two bits was already set and the other has been changed from 0 to 1. The two bits of *A* with corresponding 0's in *B* remain unchanged. The example above serves as a truth table since it has all four possible combinations of two binary variables. From the truth table we note that the bits *of A* after the operation are obtained from the logic-OR operation of bits in *B* and previous values of *A.* Therefore, the OR microoperation can be used to selectively set bits of a register.

The *selective-complement* operation complements bits in *A* where there are corresponding 1's in *B.* It does not affect bit positions that have 0's in *B.* For example:

1010	*A* before
$\underline{1100}$	*B* (logic operand)
0110	*A* after

Again the two leftmost bits of *B* are 1's, so the corresponding bits of *A* are complemented. This example again can serve as a truth table from which one can deduce that the selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The *selective-clear* operation clears to 0 the bits in *A* only where there are corresponding 1's in *B.* For example:

1010	*A* before
$\underline{1100}$	*B* (logic operand)
0010	*A* after

Again the two leftmost bits of *B* are 1's, so the corresponding bits of *A* are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is *AS.* The corresponding logic microoperation is

$$A \leftarrow A \wedge \overline{\mathbf{B}}$$

The *mask* operation is similar to the selective-clear operation except that the bits of *A* are cleared only where there are corresponding 0's in *B.*

The mask operation is an AND micro operation as seen from the following numerical example:

1010	*A* before
$\underline{1100}$	*B* (logic operand)
1000	*A* after masking

The two rightmost bits of *A* are cleared because the corresponding bits of *B* are 0's. The two leftmost bits are left unchanged because the corresponding bits of *B* are 1's. The mask operation is more convenient to use than the selective-clear operation because most computers provide an AND instruction, and few provide an instruction that executes the microoperation for selective-clear.

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an *A* register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

0110 1010	*A* before
$\underline{0000\ 1010}$	*B* (mask)
0000 1010	*A* after masking

and then insert the new value:

0000 1010	*A* before
$\underline{1001\ 0000}$	*B* (insert)
1001 1010	*A* after insert

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words in *A* and *B* and produces an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

1010	A
$\underline{1010}$	B
0000	$A \leftarrow A \oplus B$

When *A* and *B* are equal, the two corresponding bits are either both 0 or both 1. In either case the exclusive-OR operation produces a 0. The all-0's result is then checked to determine if the two numbers were equal.

3-3 Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right.

At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic.

A *logical* shift is one that transfers 0 through the serial input. We will adopt, the symbols *shl* and *shr* for logical shift-left and shift-right microoperations. For example:

$$R1 \leftarrow \textit{shl}\ R1$$
$$R2 \leftarrow \textit{shr}\ R2$$

are two microoperations that specify a 1-bit shift to the left of the content of register *Rl* and a 1-bit shift to the right of the content of register *R2.* The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The *circular* shift (also known as a *rotate* operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table 3-5

TABLE 3-5 Shift Microoperation

Symbolic designation	Description
R ← *shl* R	Shift-left register R
R← *shr* R	Shift-right register R
R← *cil* R	Circular Shift-left register R
R← *cir* R	Circular Shift-right register R
R← *ashl* R	Arithmetic Shift-left register R
R← *ashr* R	Arithmetic Shift-right register R

An *arithmetic* shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form. Figure 3-6 shows a typical register of *n* bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_{n-1} is the most significant bit of the number and R_o is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1}, and so on for the other bits in the register. The bit in R_0 is lost.

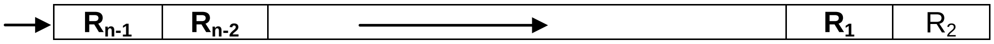


Figure 3-6 Arithmetic shift right

The arithmetic shift-left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2} . A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to $R_{n-2.}$ An overflow flip-flop V_s can be used to detect an arithmetic shift-left overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

If V_s = 0, there is no overflow, but if V_s = 1, there is an overflow and a sign reversal after the shift. *V,* must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

3-3-1 Hardware Implementation

A possible choice for a shift unit would be a bidirectional shift register with parallel load. Information can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the data into the register, and another pulse is needed to initiate the shift. In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit. In this way the content of a register that has to be shifted is first placed onto a common bus whose output is connected to the combinational shifter, and the shifted number is then loaded back into the register. This requires only one clock pulse for loading the shifted value into the register.

A combinational circuit shifter can be constructed with multiplexers as shown in Fig. 3-7. The 4-bit shifter has four data inputs. *Ay* through *Ay,* and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (I_L) and the other for shift right (I_L). When the selection input *S = 0,* the input data are shifted right (down in the diagram). When S = 1, the input data are shifted left (up in the diagram). The function table in Fig. 3-7 shows which input goes to each output after the shift. A shifter with *n* data inputs and outputs requires *n* multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.

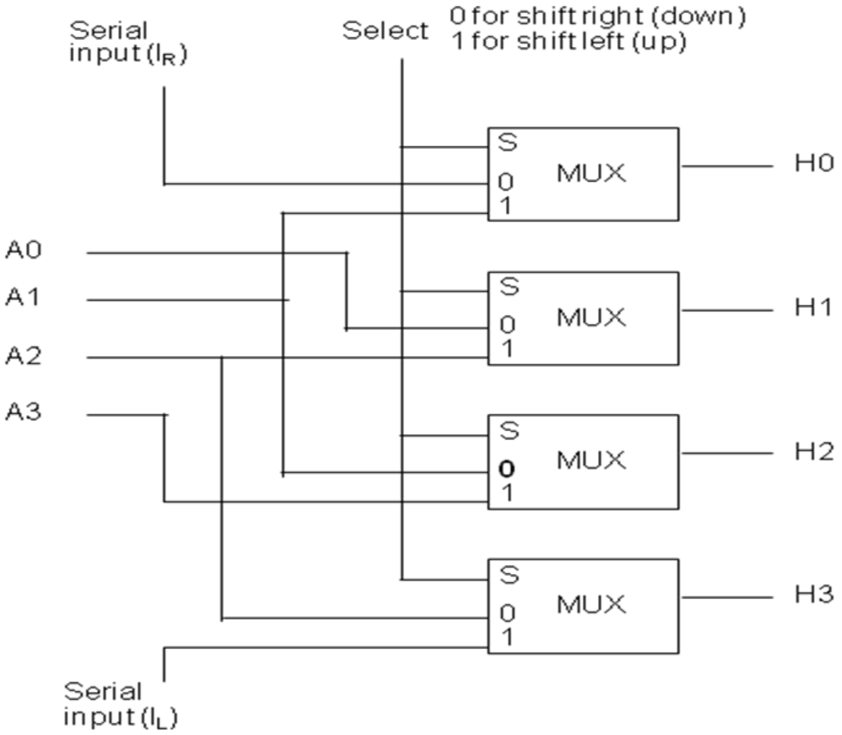


Figure 3-7 4-bit combinational circuit shifter

3-4 Arithmetic Logic Shift Unit

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits introduced in previous sections can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig. 3-8. The subscript *i* designates a typical stage. Inputs A_i and B_i, are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs S_1 and S_0. *A* 4 X 1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i. The data in the multiplexer are selected with inputs S_3 and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig. 3-8 must be repeated *n* times for an *n*-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i, of the next stage in sequence. The input carry to the first stage is the input carry C_{in}, which provides a selection variable for the arithmetic operations.

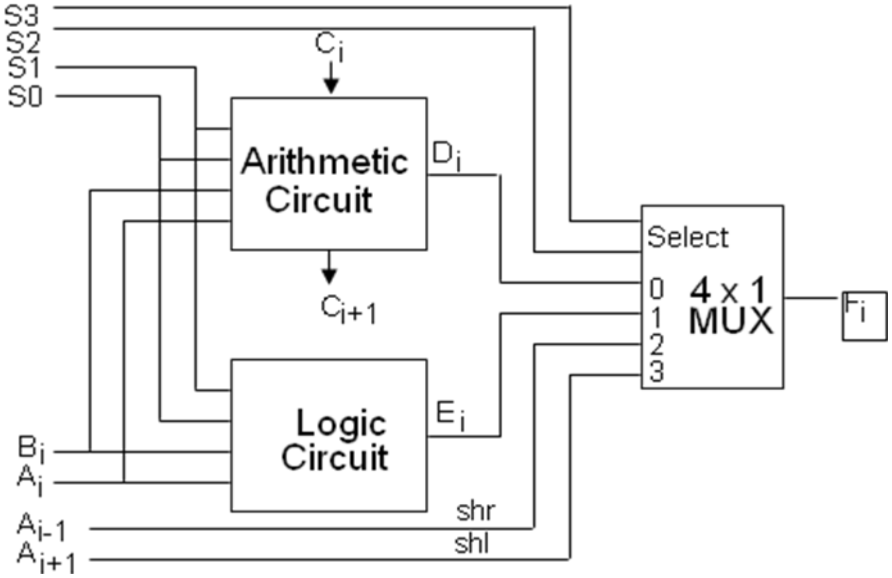
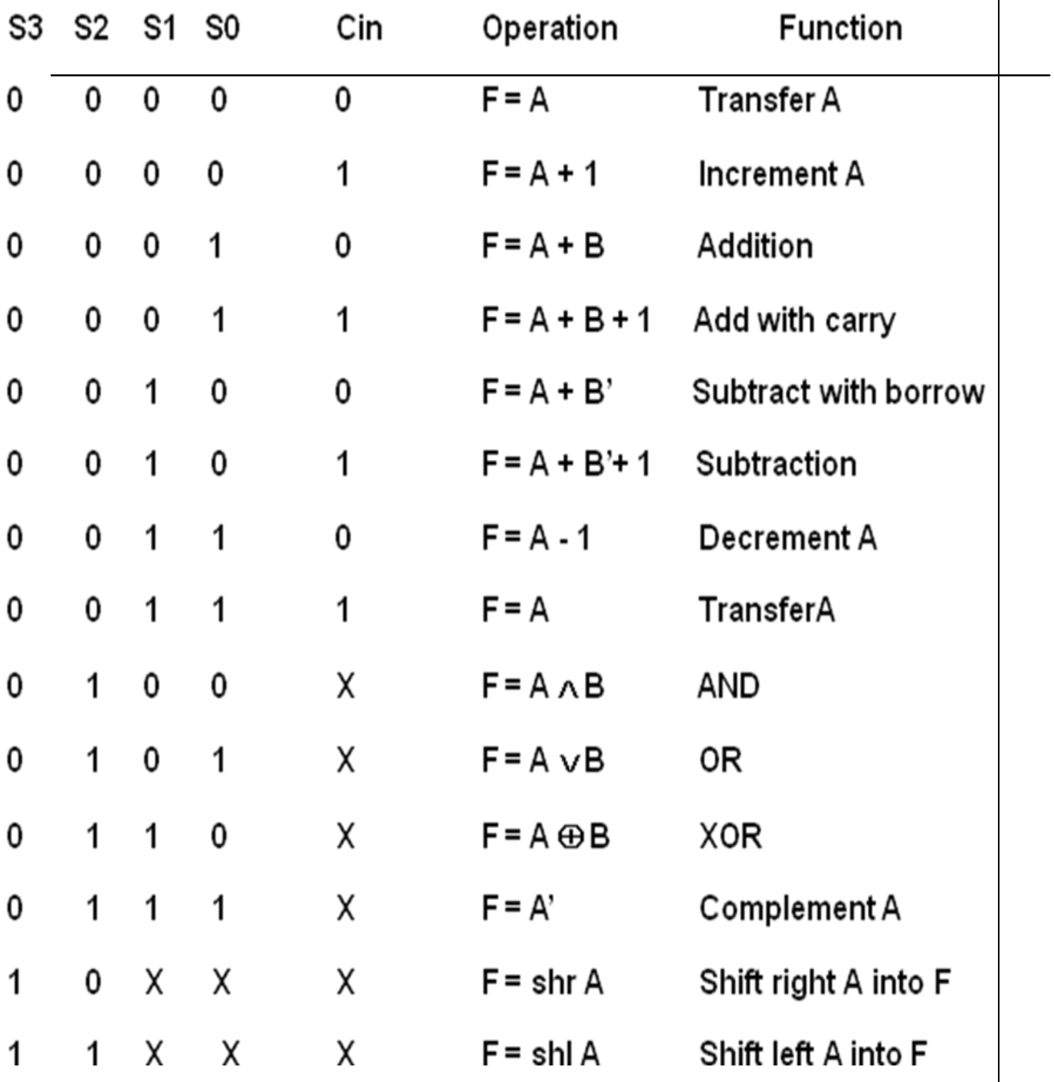


Figure 3-8 One stage of arithmetic logic shift unit

The circuit whose one stage is specified in Fig. 3-8 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables S_0, S_1, S_2, S_3, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

Table 3-6 lists the 14 operations of the ALU. The first eight are arithmetic operations (see Table 3-2) and are selected with S_3S_2 = 00. The next four are logic operations (see Fig. 3-5) and are selected with S_3S_2 = 01. The input carry has no effect during the logic operations and is marked with don't-care X's. The last two operations are shift operations and are selected with S_3S_2 = 10 and 11. The other three selection inputs have no effect on the shift.

Table 3-6 Function Table for Arithmetic Logic Shift Unit



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	F = A ∧B	AND
0	1	0	1	X	F = A ∨B	OR
0	1	1	0	X	F = A ⊕B	XOR
0	1	1	1	X	F = A’	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F

3-5 Summary

In this lesson we have discuss in details about the microoperations and their implementation in hardware using simple logic circuits. While discussing about micro-operations our main emphasis was on simple arithmetic, logic and shift micro-operations, in addition to register transfer and memory transfer.

3-6 Keywords

Add-Microoperation: The arithmetic microoperation defined by the statement specifies an add microoperation

Full-Adder: The full-adder circuit adds three one-bit binary numbers (C A B) and outputs two one-bit binary numbers, a sum (S) and a carry (C).

Shifters: Shift are used for serial transfer of data.

Microoperations: In computer central processing units, micro-operations, also known as a micro-ops or µops, are detailed low-level instructions used in some designs to implement complex machine instructions.

3-7 Exercise

1. Show the block diagram of the hardware that implements the following register transfer statement:

$$yT_2\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

2. Represent the following conditional control statement by two register transfer statements with control functions.

$$\text{If } (P = 1) \text{ then } (R1 \leftarrow R2) \text{ else if } (Q= 1) \text{ then } (R1 \leftarrow R3)$$

3. A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers.

a. How many selection inputs are there in each multiplexer?

b. What size of multiplexers is needed?

c. How many multiplexers are there in the bus?

4. Design a 4-bit combinational circuit decrementer using four full-adder circuits.

5. Assume that the 4-bit arithmetic circuit of Fig. 3-4 is enclosed in one 1C package. Show the connections among two such ICs to form an 8-bit arithmetic circuit.

6. Design a digital circuit that performs the four logic operations of exclusive-OR, exclusive-NOR, NOR, and NAND. Use two selection variables. Show the logic diagram of one typical stage.

7. Register-A holds the 8-bit binary 11011001. Determine the *B* operand and the logic microoperation to be performed in order to change the value in *A* to:

a. 01101101　　　　b. 11111101

8. An 8-bit register contains the binary value 10011100. What is the register value after arithmetic shift right? Starting from the initial number 10011100, determine the register value after an arithmetic shift left, and state whether there is an overflow.

9. Starting from an initial value of *R* = 11011101, determine the sequence of binary values in *R* after a logical shift-left, followed by a circular shift-right, followed by a logical shift-right and a circular shift-left.

3-8 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

NOTES

4. BASIC COMPUTER ORGANIZATION AND DESIGN

Structure

Objectives

At the end of the lesson you will be able to:

- Discuss about various instruction codes
- Distinguish various types of computer instructions
- Discuss about timing and control

- Define Instruction Cycle
- Define terms of memory-reference I, Input-Output and Interrupt

4-1 Instruction Codes

In this lesson we introduce a basic computer and show how its operation can be specified with register transfer statements. The organization of the computer is defined by its internal registers, the riming and control structure, and the set of instructions that it uses. The design of the computer is then carried out in detail. Although the basic computer presented in this lesson is very small compared to commercial computers, it has the advantage of being simple enough so we can demonstrate the design process without too many complications.

The internal organization of a digital system is denned by the sequence of microoperations it performs on data stored in its registers. The general-purpose digital computer is capable of executing various microoperations and, in addition, can be instructed as to what specific sequence of operations it must perform. The user of a computer can control the process by means of a program. A program is a set of instructions that specify the operations, operands, and the sequence by which processing has to occur. The data-processing task may be altered by specifying a new program with different instructions or specifying the same instructions with different data.

A computer instruction is a binary code that specifies a sequence of microoperations for the computer. Instruction codes together with data are stored in memory. The computer reads each instruction from memory and places it in a control register. The control then interprets the binary code of the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set. The ability to store and execute instructions, the stored program concept, is the most important property of a general-purpose computer.

An instruction code is a group of bits that instruct the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement. The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer. The operation code must consist of at least n bits for a given 2^n (or less) distinct operations. As an illustration, consider a computer with 64 distinct operations, one of them being an ADD operation. The operation code consists of six bits, with a bit configuration 110010 assigned to the ADD operation. When this operation code is decoded in the control unit, the computer issues control signals to read an operand from memory and add the operand to a processor register.

At this point we must recognize the relationship between a computer operation and a microoperation. An operation is part of an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit receives the instruction from memory and j interprets the operation code bits. It then issues a sequence of control signals to initiate microoperations in internal computer registers. For every operation | code, the control issues a sequence of microoperations needed for the hardware implementation of the specified operation. For this reason, an operation code is sometimes called a microoperation because it specifies a set of micro-operations.

The operation part of an instruction code specifies the operation to be performed. This operation must be performed on some data stored in processor registers or in memory. An instruction code must therefore specify not only the operation but also the registers or the memory words where the operands are to be found, as well as the register or memory word where the result is to be stored. Memory words can be specified in instruction codes by their address. Processor registers can be specified by assigning to the instruction another binary code of k bits that specifies one of 2^k registers. There are many variations for arranging the binary code of instructions, and each computer has its own particular instruction code format. Instruction code formats are conceived by computer designers who specify the architecture of the computer. In this chapter we choose a particular instruction code to explain the basic organization and design of digital computers.

4-1-1 Stored Program Organization

The simplest way to organize a computer is to have one processor register and an instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

Figure 4-1 depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since 2^{12} = 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code. Computers that have a single-processor register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

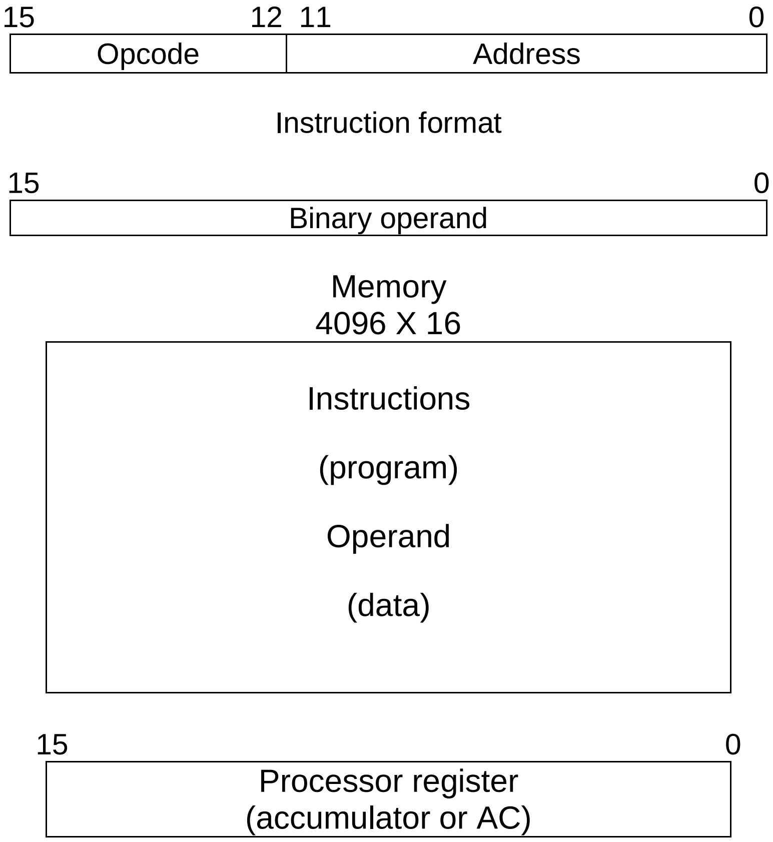


Figure 4-1 Stored program organization

If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes. For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the A C register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

4-1-2 Indirect Address

It is sometimes convenient to use the address bits of an instruction code not as an address but as the actual operand. When the second part of an instruction code specifies an operand, the instruction is said to have an immediate operand. When the second part specifies the address of an operand, the instruction is said to have a direct address. This is in contrast to a third possibility called indirect address, where the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found. One bit of the instruction code can be used to distinguish between a direct and an indirect address.

As an illustration of this configuration, consider the instruction code format shown in Fig. 4-2 (a). It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I. The mode bit is 0 for a direct address and 1 for an indirect address. A direct address instruction is shown in Fig. 4-2 (b). It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC. The instruction in address 35 shown in Fig. 4-2(c) has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC. The indirect address instruction needs two references to memory to fetch an operand. The first reference is needed to read the address of the operand; the second is for the operand itself. We define the effective address to be the address of the operand in a computation-type instruction or the target address in a branch-type instruction. Thus the effective address in the instruction of Fig. 4-2(b) is 457 and in the instruction of Fig 4-2(c) is 1350.

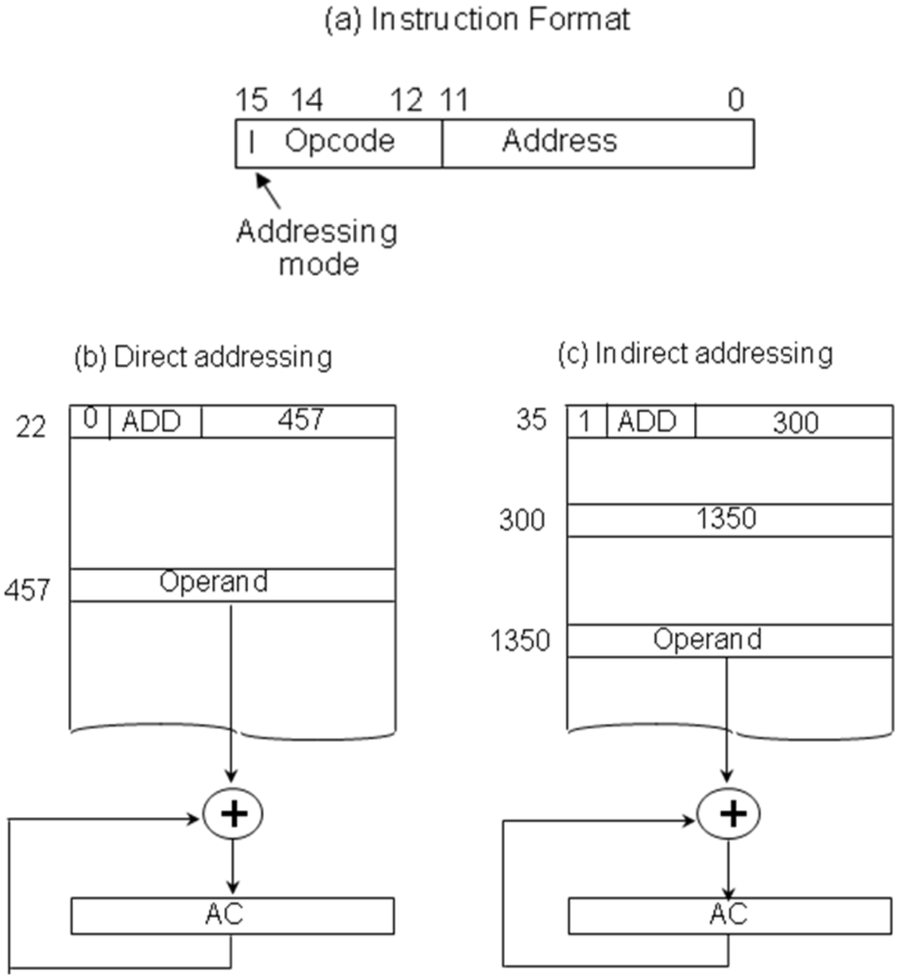


Figure 4-2 Demonstration of direct and indirect address

The direct and indirect addressing modes are used in the computer presented in this lesson. The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data. The pointer could be placed in a processor register instead of memory as done in commercial computers

4-2 Computer Registers

Computer instructions are normally stored in consecutive memory locations and are executed sequentially one at a time. The control reads an instruction from a specific address in memory and executes it. It then continues by reading the next instruction in sequence and executes it, and so on. This type of instruction sequencing needs a counter to calculate the address of the next instruction after execution of the current instruction is completed. It is also necessary to provide a register in the control unit for storing the instruction code after it is read from memory. The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in Fig. 4-3. The registers are also listed in Table 4-1 together with a brief description of their function and the number of bits that they contain.

The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand. This leaves three bits for the operation, part of the instruction and a bit to specify a direct or indirect address. The data register (DR) holds the operand read from memory. The accumulator (AC) register is a general-purpose processing register. The instruction read from memory is placed in the instruction register (IR). The temporary register (TR) is used for holding temporary data during the processing.

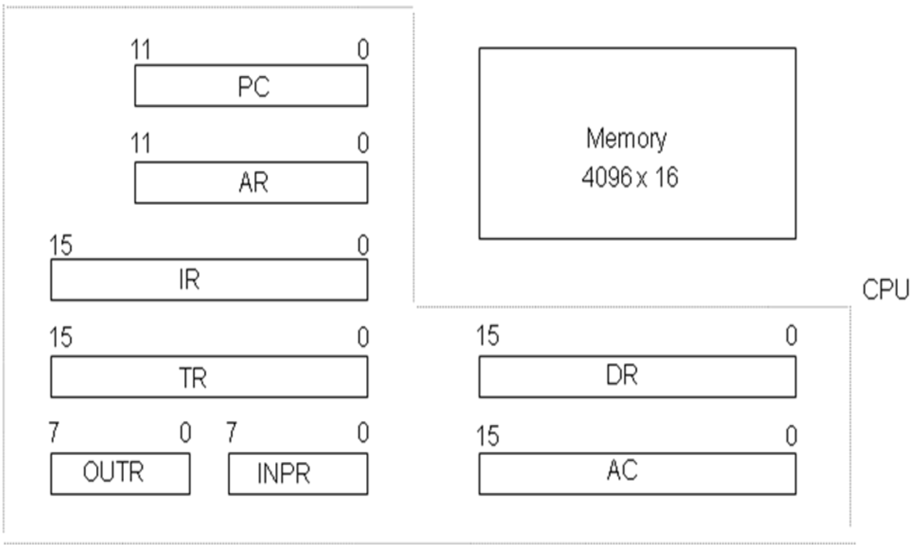


Figure 4-3 Basic computer registers and memory

TABLE 4-1 List of Registers for the Basic Computer

Register Symbol	Number of bits	Register name	Function
DR	16	Data Register	Holds memory operand
AR	12	Address Register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction Register	Holds instruction code
PC	12	Program Counter	Holds address of instruction
TR	16	Temporary Register	Holds temporary data
INTR	8	Input Register	Holds input character
OUTR	8	Output Register	Holds output character

The memory address register (AR) has 12 bits since this is the width of a memory address. The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed. The PC goes through a counting sequence and causes the computer to read sequential instructions previously stored in memory. Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program. The address part of a branch instruction is transferred to PC to become the address of the next instruction. To read an instruction, the content of PC is taken as the address for memory and a memory read cycle is initiated. PC is then incremented by one, so it holds the address of the next instruction in sequence.

Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

4-2-1 Common Bus System

The basic computer has eight registers, a memory unit, and a control unit (to be presented in Sec. 4-4). Paths must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the outputs of each register and the inputs of the other registers. A more efficient scheme for transferring information in a system with many registers is to use a common bus. The connection of the registers and memory of the basic computer to a common bus system is shown in Fig. 4-4.

The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S_2, S_1, and S_o. The number along each output shows the decimal equivalent of the required binary selection. For example, the number along the output of DR is 3. The 16-bit outputs of DR are placed on the bus lines when S_2 S_1S_o =011 since this is the binary value of decimal 3. The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S_2 S_1S_o = 111.

Four registers, DR, AC, IR, and TR, have 16 bits each. Two registers, AR and PC, have 12 bits each since they hold a memory address. When the contents of AR or PC axe applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. This is because INPR receives a character from an input device which is then transferred to AC. OUTR receives a character from AC and delivers it to an output device. There is no transfer from OUTR to any of the other registers.

The 16 lines of the common bus receive information from six registers and the memory unit. The bus lines are connected to the inputs of six registers and the memory. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). The increment operation is achieved by enabling the count input of the counter. Two registers have only a LD input.

The input data and output data of the memory are connected to the common bus, but the memory address is connected to AR. Therefore, AR must always be used to specify a memory address. By using a single register for the address, we eliminate the need for an address bus that would have been needed otherwise. The content of any register can be specified for the memory data input during a write operation. Similarly, any register can receive the data from memory after a read operation except AC

NOTES

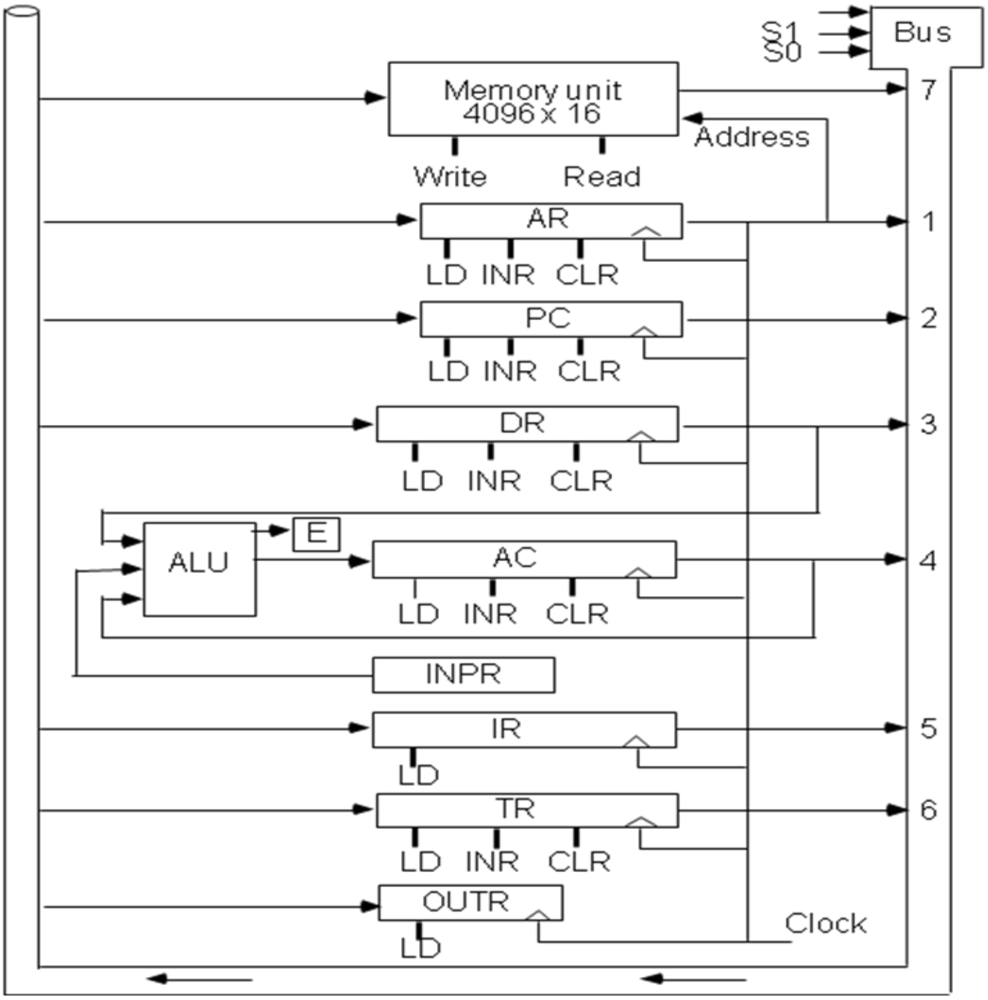


Figure 4-4 Common Bus System

The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs. One set of 16-bit inputs come from the outputs of AC. They are used to implement register microoperations such as complement AC and shift AC. Another set of 16-bit inputs come from the data register DR. The inputs from DR and AC are used for arithmetic and logic microoperations, such as add DR to AC or AND DR to AC. The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit). A third set of 8-bit inputs come from the input register INPR. The operation of INPR and OUTR is explained in Sec. 4-7.

Note that the content of any register can be applied onto the bus and an operation can be performed in the adder and logic circuit during the same clock cycle. The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC. For example, the two microoperations

DR ← AC and AC ← DR

can be executed at the same time. This can be done by placing the content of AC on the bus (with S_2 S_1S_o = 100), enabling the LD (load) input of DR, transferring the content of DR through the adder and logic circuit into AC, and enabling the LD (load) input of AC, all during the same clock cycle. The two transfers occur upon the arrival of the clock pulse transition at the end of the clock cycle.

4-3 Computer Instructions

The basic computer has three instruction code formats, as shown in Fig. 4-5. Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered. A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address (see Fig. 4-2). The register-reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the .4 C register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed. Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

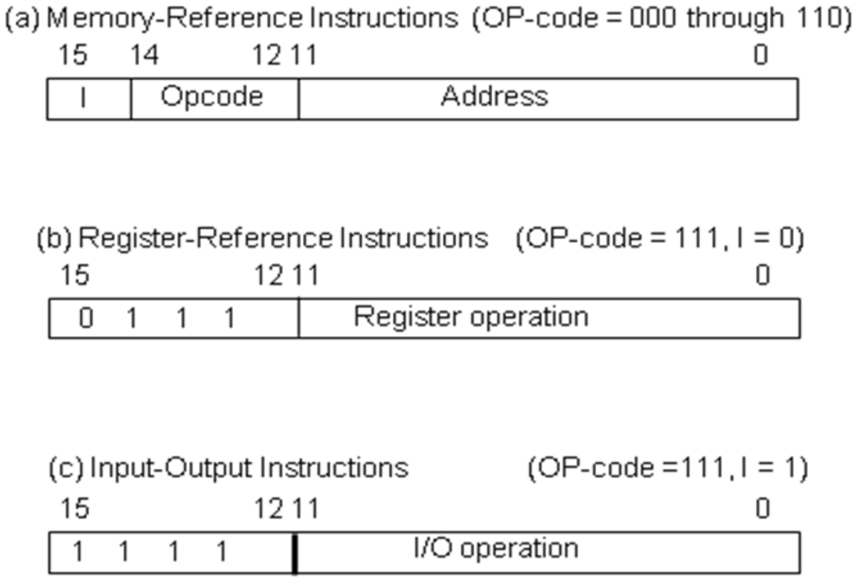
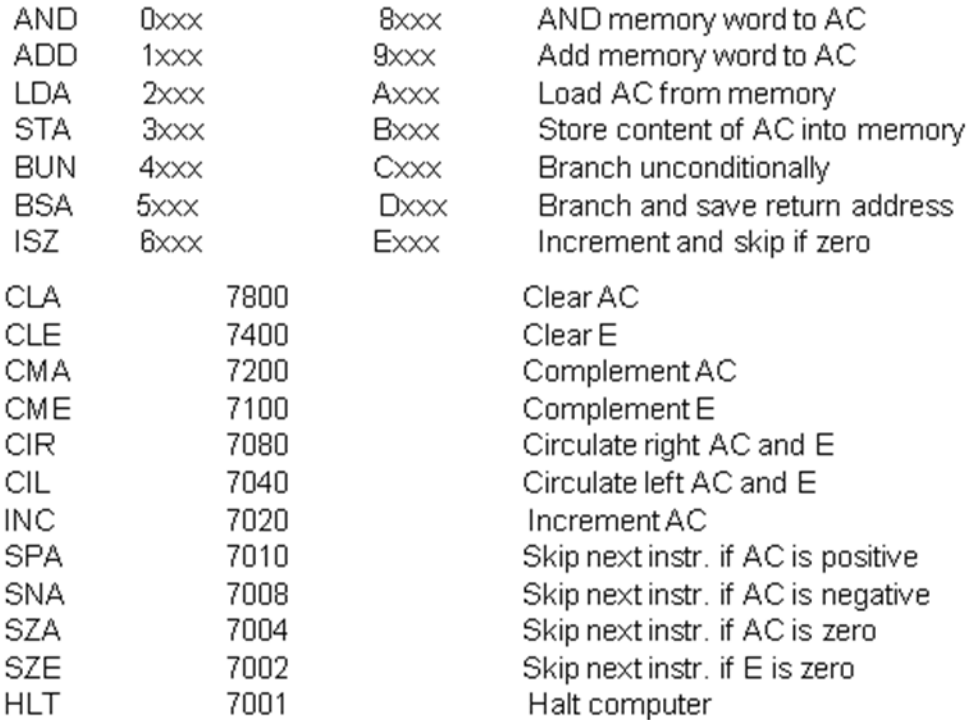

(a) Memory-Reference Instructions (OP-code = 000 through 110)
15
14
12
11
0
I
Opcode
Address
(b) Register-Reference Instructions (OP-code = 111, I = 0)
15
12
11
0
0 1 1 1
Register operation
(c) Input-Output Instructions (OP-code =111, I = 1)
15
12
11
0
1 1 1 1
I/O operation

Figure 4-5 Basic Computer Instruction Format

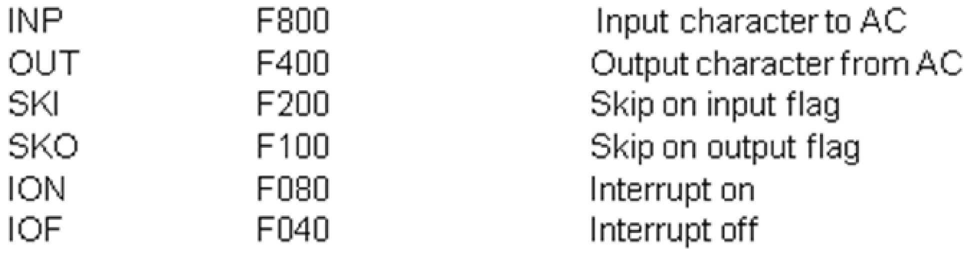
The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three opcode bits in positions 12 though 14 are not equal to 111, the instruction is a memory-reference type and the bit in position 15 is taken as the addressing mode I. If the 3-bit opcode is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a register-reference type. If the bit is 1, the instruction is an input-output type. Note that the bit in position 15 of the instruction code is designated by the symbol / but is not used as a mode bit when the operation code is equal to 111.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. However, since register-reference and input-output instructions use the remaining 12 bits as part of the operation code, the total number of instructions can exceed eight. In fact, the total number of instructions chosen for the basic computer is equal to 25.

The instructions for the computer are listed in Table 4-2. The symbol designation is a three-letter word and represents an abbreviation intended for programmers and users. The hexadecimal code is equal to the equivalent hexadecimal number of the binary code used for the instruction. By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stand for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I. When I = 0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When I = 1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1.



	Hexadecimal code		
Symbol	I = 0	I = 1	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer



INP	F800	Input character to AC
OUT	F400	Output character from AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	Interrupt on
IOF	F040	Interrupt off

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binar) equivalent of the remaining 12 bits. The input-output instructions also use all 16 bits to specify an operation. The last four bits are always 1111, equivalent to hexadecimal F.

4-3-1 Instruction Set Completeness

Before investigating the operations performed by the instructions, let us discuss the type of instructions that must be included in a computer. A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions

2. Instructions for moving information to and from memory and processor registers

3. Program control instructions together with instructions that check status conditions

4. Input and output instructions

Arithmetic, logical, and shift instructions provide computational capabilities for processing the type of data that the user may wish to employ. The bulk of the binary information in a digital computer is stored in memory, but all computations are done in processor registers. Therefore, the user must have the capability of moving information between these two units. Decision making capabilities are an important aspect of digital computers. For example, two numbers can be compared, and if the first is greater than the second, it may be necessary to proceed differently than if the second is greater than the first. Program control instructions such as branch instructions are used to change the sequence in which the program is executed. Input and output instructions are needed for communication between the computer and the user. Programs and data must be transferred into memory and results of computations must be transferred back to the user.

NOTES

The instructions listed in Table 4-2 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction, ADD, and two related instructions, complement AC(CMA) and increment AC(INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions, CIR and CIL; can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting. There are three logic operations: AND, complement AC (CMA), and clear AC (CLA). The AND complement provide a NAND operation. Moving information from memory to AC is accomplished with the load AC(LDA) instruction. Storing information from AC into memory is done with the store AC (STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions. The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

Although the set of instructions for the basic computer is complete, it is not efficient because frequently used operations are not performed rapidly. An efficient set of instructions will include such instructions as subtract, multiply, OR, and exclusive-OR. These operations must be programmed in the basic computer. By using a limited number of instructions it is possible to show the detailed logic design of the computer. A more complete set of instructions would have made the design too complex. In this way we can demonstrate the basic principles of computer organization and design without going into excessive complex details.

4-4 Timing and Control

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization: hardwired control and microprogrammed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or changed. In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory. A hardwired control for the basic computer is presented in this section.

The block diagram of the control unit is shown in Fig. 4-6. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The position of this register in the common bus system is indicated in Fig. 4-4. The instruction register is shown again in Fig. 4-6, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 X 8 decoder. The eight outputs of the decoder are designated by the symbols D_o through D_7. The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T_0 through T_{15}. The internal logic of the control gates will be derived later when we consider the design of the computer in detail.

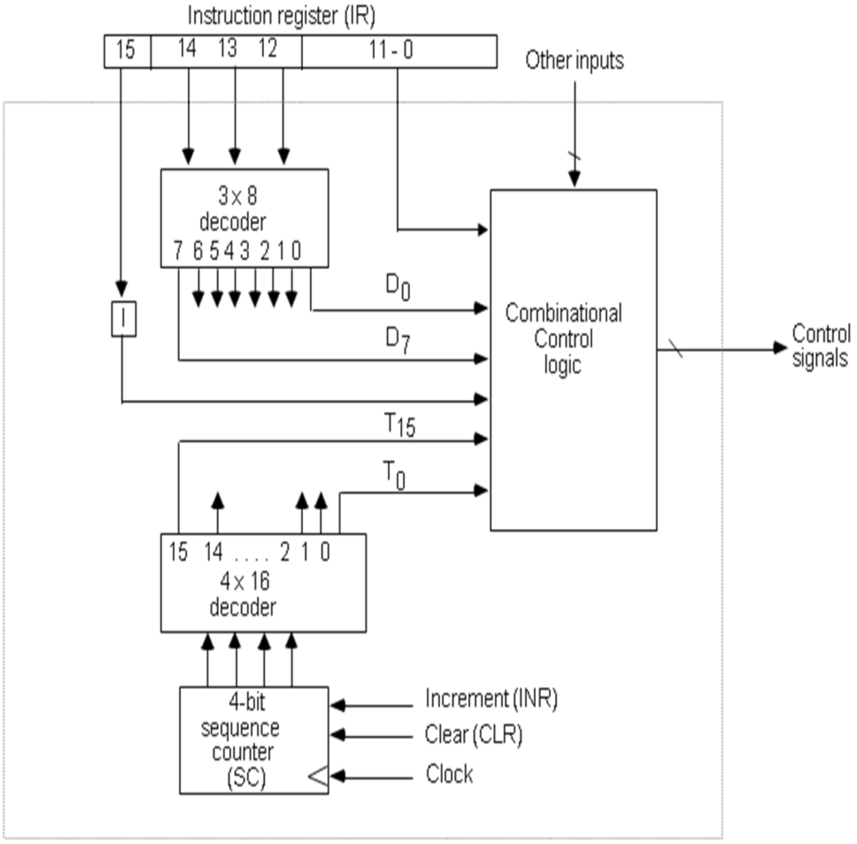


Figure 4-6 Control unit of Basic Computer

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 X 16 decoder. Once in awhile, the counter is cleared to 0, causing the next active timing signal to be T_O. As an example, consider the case where SC is incremented to provide timing

NOTES

signals T_o, T_1, T_2, T_3, and T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement

$$D_3T_4\text{: SC} \leftarrow 0$$

The timing diagram of Fig. 4-7 shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T_o out of the decoder. T_o is active during one clock cycle. The positive clock transition labeled T_o in the diagram will trigger only those registers whose control inputs are connected to timing signal To. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signalsT_0, T1, T_2, T_3, T_4 and so on, as shown in the diagram. (Note the relationship between the timing signal and its corresponding positive clock transition.) If SC is not cleared, the timing signals will continue with T_5, T_6, up to T_{15}, and back to T_o.

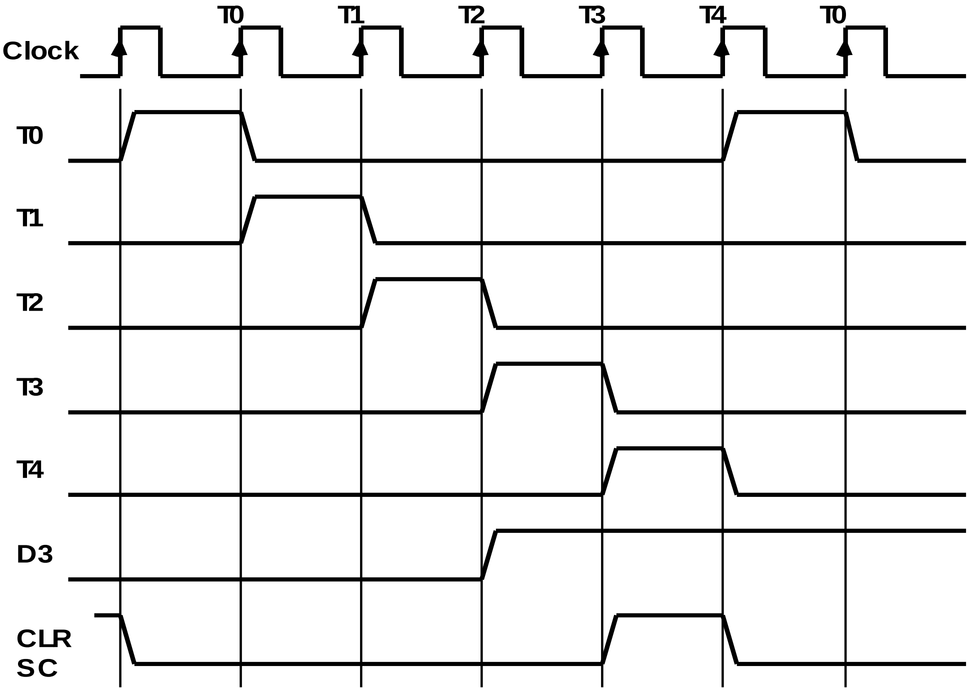


Figure 4-7 Examples of control timing signals

The last three waveforms in Fig. 4-7 show how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked 7_4 in the diagram) the counter is cleared to 0. This causes the timing signal T_o to become active instead of 7_5 that would have been active SC were incremented instead of cleared.

A memory read or writes cycle will be initiated with the rising edge of timing signal. It will be assumed that a memory cycle time is less than clock cycle time. According to this assumption, a memory read or writes cycle initiated by a timing signal will be completed by the time the next clock through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor until the memory word is available. To facilitate the presentation, we will assume that a wait period is not necessary in the basic computer.

To fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$T_0\text{: AR} \leftarrow \text{PC}$$

specifies a transfer of the content of PC into AR if timing signal T_o is active. T_0 is active during an entire clock cycle interval. During this time the content of PC is placed onto the bus (with $S_2S_1S_0$ = 010) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

4-5 Instruction Cycle

A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in rum is subdivided into a sequence of sub-cycles or phases. In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction. This process continues indefinitely unless a HALT instruction is encountered.

4-5-1 Fetch and Decode

Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

$$T_0: AR \leftarrow PC$$

$$T_1: IR \leftarrow M[AR], PC \leftarrow PC + 1$$

$$T_2: D_0 \ldots D_7 \leftarrow PC \text{ Decode } IR(12\text{-}14), AR \leftarrow IR(0\text{-}11), I \leftarrow IR(15)$$

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T_o. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal 7_1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At rime T_2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. Note that SC is incremented after each clock pulse to produce the sequence T_o, T_1,, and T_2.

Figure 4-8 shows how the first two register transfer statements are implemented in the bus system. To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

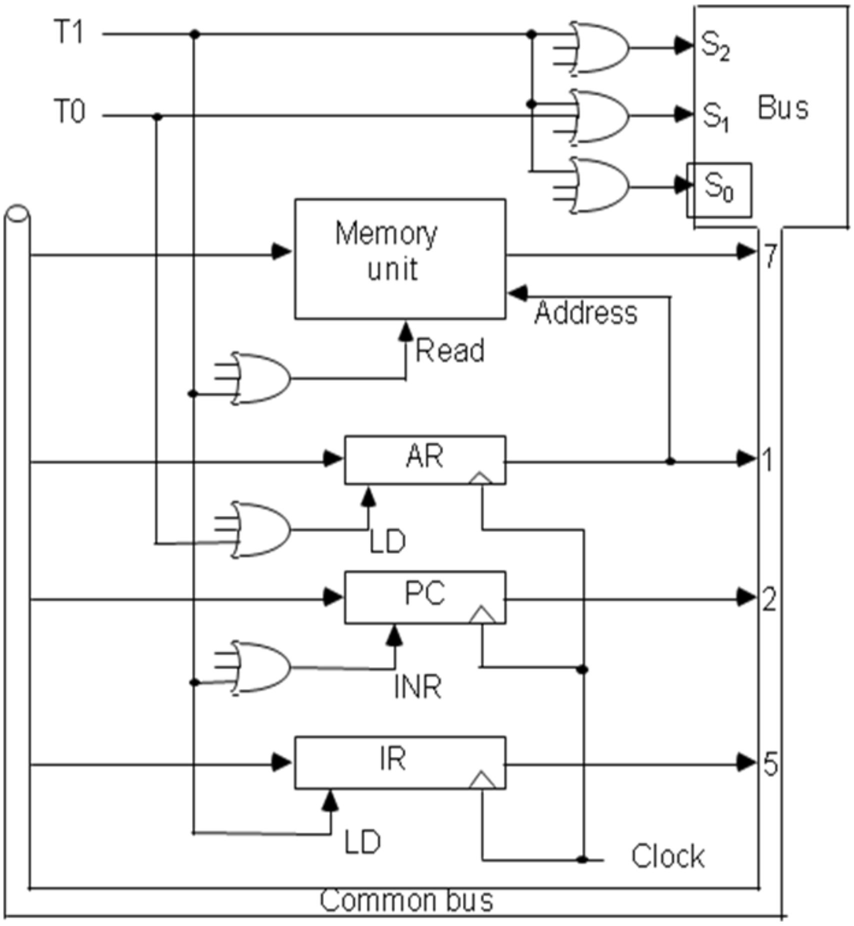


Figure 4-8 Register transfers for the fetch phase

1. Place the content of .PC onto the bus by making the bus selection inputs $S_2S_1S_o$ equal to 010.

2. Transfer the content of the bus to AR by enabling the LD input of AR

The next clock transition initiates the transfer from PC to AR since $T_0 = 1$. In order to implement the second statement

T_1: IR ← M [AR], PC ← PC + 1

it is necessary to use timing signal T_1 to provide the following connections in the bus system.

1. Enable the read input of memory.

2. Place the content of memory onto the bus by making $S_2S_1S_o$ =111.

3. Transfer the content of the bus to IR by enabling the LD input of IR.

4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since T_1= 1

Figure 4-8 duplicates a portion of the bus system and shows how T_0 and T_1 are connected to the control inputs of the registers, the memory, and the bus selection inputs. Multiple input OR gates are included in the diagram because there are other control functions that will initiate similar operations.

4-5-2 Determine the Type of Instruction

The timing signal that is active after the decoding is T_3. During time T_3, the control unit determines the type of instruction that was just read from memory. The flowchart of Fig. 4-9 presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding. The three possible instruction types available in the basic computer are specified in Fig. 4-5.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. From Fig. 4 -5 we determine that if D_7= 1, the instruction must be a register-reference or input-output type. If D_7= 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction which is now available in flip-flop I. If D_7 = 0 and I = 1, we have a memory reference instruction with an indirect address. It is then necessary to read the effective address from memory. The microoperation for the indirect address condition can be symbolized by the register transfer statement

$$AR \leftarrow M[AR]$$

Initially, AR holds the address part of the instruction. This address is used during the memory read operation. The word at the address given by AR is read from memory and placed on the common bus. The LD input of AR is then enabled to receive the indirect address that resided in the 12 least significant bits of the memory word.

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:

D'_7IT_3 : AR ← M[AR]

D'_7IT_3 : nothing

$D_7I'T_3$: Execute a register reference instruction

D_7IT_3 : Execute an input-output reference instruction

When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.

However, the sequence counter SC must be incremented when D'_7T_3= 1, so that the execution of the memory-reference instruction can be continued with timing variable T_4. A register-reference or input-output instruction can be executed with the clock associated with timing signal T_3. After the instruction is executed, SC is cleared to 0 and control returns to the fetch phase with T_o = 1.

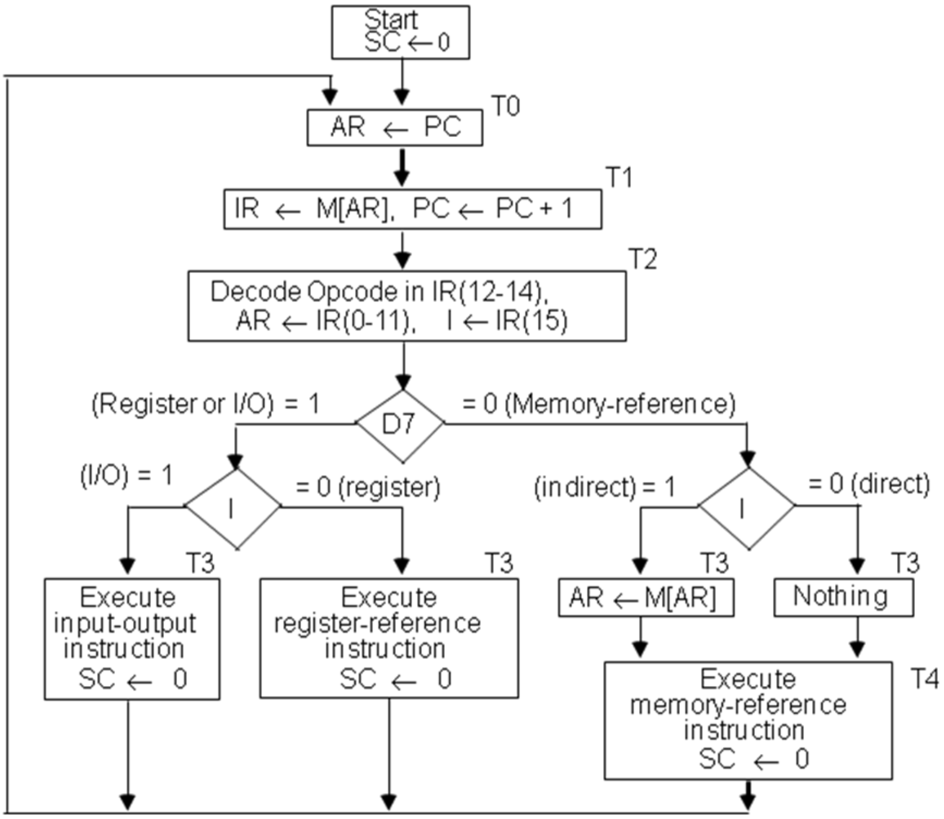

Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
Execute memory-reference instruction SC ← 0
T4

Figure 4-9 Flowchart for instruction cycle

Note that the sequence counter SC is either incremented or cleared to 0 with every positive clock transition. We will adopt the convention that if SC is incremented, we will not write the statement SC ← SC + 1, but it will be implied that the control goes to the next timing signal in sequence. When SC is to be cleared, we will include the statement SC ← 0.

The register transfers needed for the execution of the register-reference instructions are presented in this section. The memory-reference instructions are explained in the next section. The input-output instructions are included in Sec. 4-7.

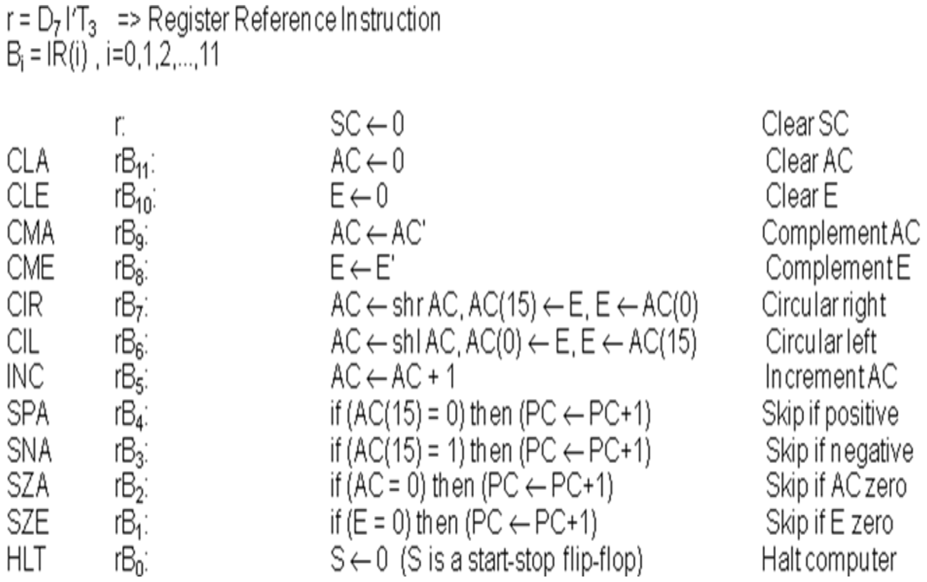
4-5-3 Register-Reference Instructions

Register-reference instructions are recognized by the control when D_7 = 1 and I = 0. These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions. These 12 bits are available in IR (0-11). They were also transferred to AR during time T_2.

The control functions and microoperations for the register-reference instructions are listed in Table 4-3. These instructions are executed with the clock transition associated with timing variable T_3. Each control function needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B, to bit i of IR, all control functions can be simply denoted by rB_i. For example, the instruction CLA has the hexa-decimal code 7800 (see Table 4-2), which gives the binary equivalent 0111 1000 0000 0000. The first bit is a zero and is equivalent to I'. The next three bits constitute the operation code and are recognized from decoder output D_7. Bit 11 in IR is 1 and is recognized from B_{11}. The control function that initiates the microoperation for this instruction is $D_7I'T_3B_{11}$ $^=$ rB_{11} the execution of a register-reference instruction is completed at time T_3. The sequence counter SC is cleared to 0 and the control goes back to fetch the next instruction with timing signal T_o.

The first seven register-reference instructions perform clear, complement, circular shift, and increment microoperations on the A C or E registers. The next four instructions cause a skip of the next instruction in sequence when a stated condition is satisfied. The skipping of the instruction is achieved by incrementing PC once again (in addition, it is being incremented during the fetch phase at time T_1). The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC (15) = 0; it is negative when AC (l5) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero. The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

TABLE 4-3 Execution Register- Reference Instruction



$r = D_7 I' T_3$ => Register Reference Instruction
$B_i = IR(i)$, i=0,1,2,...,11

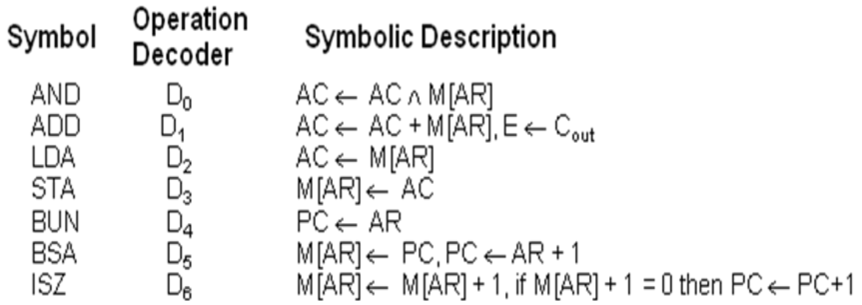
	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow AC'$	Complement AC
CME	rB_8:	$E \leftarrow E'$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$	Circular right
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$	Circular left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$	Skip if positive
SNA	rB_3:	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$	Skip if negative
SZA	rB_2:	if $(AC = 0)$ then $(PC \leftarrow PC+1)$	Skip if AC zero
SZE	rB_1:	if $(E = 0)$ then $(PC \leftarrow PC+1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start-stop flip-flop)	Halt computer

4-6 Memory-Reference Instructions

In order to specify the microoperations needed for the execution of each instruction, it is necessary that the function that they are intended to perform be defined precisely. Looking back to Table 4-2, where the instructions are listed, we find that some instructions have an ambiguous description. This is because the explanation of an instruction in words is usually lengthy, and not enough space is available in the table for such a lengthy explanation. We will now show that the function of the memory-reference instructions can be defined precisely by means of register transfer notation.

Table 4-4 lists the seven memory-reference instructions. The decoded output D_i, for i = 0,1,2,3,4,5, and 6 from the operation decoder that belongs to each instruction is included in the table. The effective address of the instruction is in the address register AR and was placed there during timing signal T_3 when I = 0, or during timing signal T_2 when I = 1. The execution of the memory-reference instructions starts with timing signal T_4. The symbolic description of each instruction is specified in the table in terms of register transfer notation. The actual execution of the instruction in the bus system will require a sequence of microoperations. This is because data stored in memory cannot be processed directly. The data must be read from memory to a register where they can be operated on with logic circuits. We now explain the operation of each instruction and list the control functions and microoperations needed for their execution. A flowchart that summarizes all the microoperations is presented at the end of this section.

TABLE 4-4 Memory-Reference Instructions



Symbol	Operation Decoder	Symbolic Description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, if $M[AR] + 1 = 0$ then $PC \leftarrow PC+1$

4-6-1 AND to AC

This is an instruction that performs the AND logic operation on pairs of bits in AC and the memory word specified by the effective address. The result of the operation is transferred to AC. The microoperations that execute this instruction are:

D_0T_4: $DR \leftarrow M[AR]$ Read operand

D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$ AND with AC

The control function for this instruction uses the operation decoder D_o since this output of the decoder is active when the instruction has an AND operation whose binary code value is 000. Two tuning signals are needed to execute the instruction. The clock transition associated with timing signal T_4 transfers the operand from memory into DR. The clock transition associated with the next timing signal T_5 transfers to AC the result of the AND logic operation between the contents of DR and AC. The same clock transition clears SC to 0, transferring control to timing signal T_o to start a new instruction cycle.

4-6-2 ADD to AC

This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop. The micro-operations needed to execute this instruction are

D_1T_4: $DR \leftarrow M[AR]$ Read operand

D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$ Add to AC and store carry in **E**

The same two timing signals, T_4 and T_5, are used again but with operation decoder D_1 instead of D_o, which was used for the AND instruction. After the instruction is fetched from memory and decoded, only one output of the operation decoder will be active, and that output determines the sequence of microoperations that the control follows during the execution of a memory reference instruction.

4-6-3 LDA: Load to AC

This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

$$D_2T_4: \quad DR \leftarrow M[AR]$$

$$D_2T_5: \quad AC \leftarrow DR, SC \leftarrow 0$$

Looking back at the bus system shown in Fig. 4-4 we note that there is no direct path from the bus into AC. The adder and logic circuit receive information from DR which can be transferred into AC. Therefore, it is necessary to read the memory word into DR first and then transfer the content of DR into AC. The reason for not connecting the bus to the inputs of AC is the delay encountered in the adder and logic circuit. It is assumed that the time it takes to read from memory and transfer the word through the bus as well as the adder and logic circuit is more than the time of one clock cycle. By not connecting the bus to the inputs of AC we can maintain one clock cycle per microoperation.

4-6-4 STA: Store AC

This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one microoperation:

$$D_3T_4: \quad M[AR] \leftarrow AC, SC \leftarrow 0$$

4-6-7 BUN: Branch Unconditionally

This instruction transfers the program to the instruction specified by the effective address. Remember that PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time T_1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one microoperation:

$$D_4T_4: \quad PC \leftarrow AR, SC \leftarrow 0$$

The effective address from AR is transferred through the common bus to PC. Resetting SC to 0 transfers control T_0. The next instruction is then fetched and executed from the memory address given by the new value in PC.

NOTES

4-6-8 BSA: Branch and Save Return Address

This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence (which is available in PC) into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified in Table 4-4 with the following register transfer:

$$M\ [AR] \leftarrow PC,\ PC \leftarrow AR + 1$$

A numerical example that demonstrates how this instruction is used with a subroutine is shown in Fig. 4-10. The BSA instruction is assumed to be in memory at address 20. The I bit is 0 and the address part of the instruction has the binary equivalent of 135. After the fetch and decode phases, PC contains 21, which is the address of the next instruction in the program (referred to as the return address). AR holds the effective address 135. This is shown in part (a) of the figure. The BSA instruction performs the following numerical operation:

$$M\ [135] \leftarrow 21,\ PC \leftarrow 135 + 1 = 136$$

The result of this operation is shown in part (b) of me figure. The return address 21 is stored in memory location 135 and control continues with the subroutine program starting from address 136. The return to the original program (at address 21) is accomplished by means of an indirect BUN instruction placed at the end of the subroutine. When this instruction is executed, control goes to the indirect phase to read the effective address at location 135, where it finds the previously saved address 21. When the BUN instruction is executed, the effective address 21 is transferred to PC. The next instruction cycle finds PC with the value 21, so control continues to execute the instruction at the return address. The BSA instruction performs the function usually referred to as a subroutine call. The indirect BUN instruction at the end of the subroutine performs the function referred to as a subroutine return. In most commercial computers, the return address associated with a subroutine is stored in either a processor register or in a portion of memory called a stack.

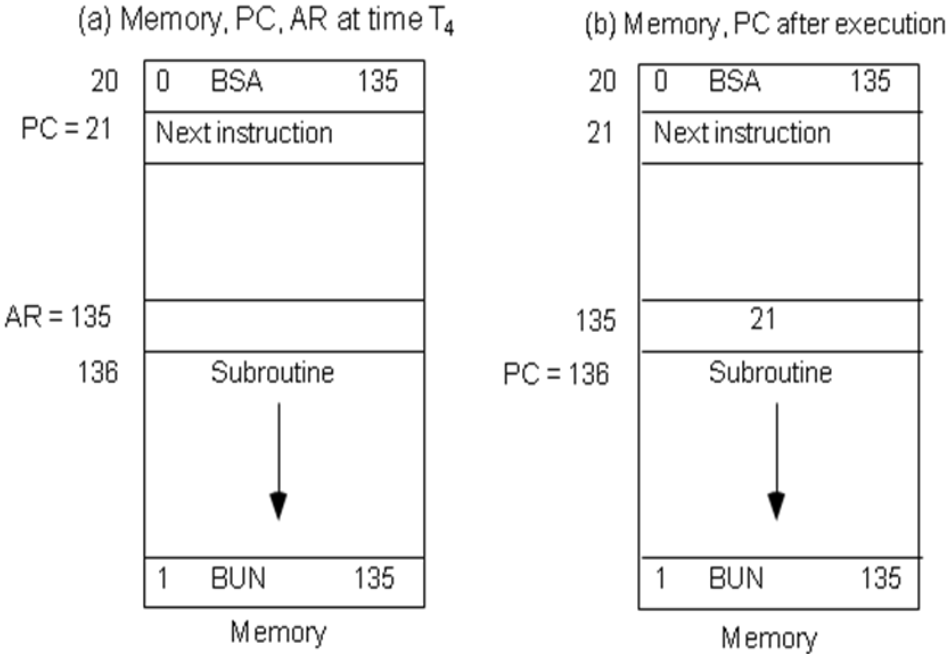


Figure 4-10 Example of BSA instruction execution

It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two microoperations:

$$D_5T_4: \quad M[AR] \leftarrow PC, \ \ AR \leftarrow AR + 1$$

$$D_5T_5: \quad PC \leftarrow AR, \ SC \leftarrow 0$$

Timing signal T_4 initiates a memory write operation, places the content of PC onto the bus, and enables the INR input of AR. The memory write operation is completed and AR is incremented by the time the next clock transition occurs. The bus is used at T_5 to transfer the content of AR to PC.

4-6-9 ISZ: Increment and Skip if Zero

These instruction increments the word specified by the effective address, and if the incremented value is equal to 0, .PC is incremented by 1. The programmer usually stores a negative number (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is incremented by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations:

D_6T_4: DR ← M[AR]

D_6T_5: DR ← DR + 1

D_6T_4: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

4-6-10 Control Flowchart

A flowchart showing all microoperations for the execution of the seven memory-reference instructions is shown in Fig. 4-11. The control functions are indicated on top of each box. The microoperations that are performed during time T_4, T_5, or T_6 depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T_o to start the next instruction cycle.

Note that we need only seven timing signals to execute the longest instruction (ISZ). The computer can be designed with a 3-bit sequence counter. The reason for using a 4-bit counter for SC is to provide additional timing signals for other instructions that are presented in the problems section.

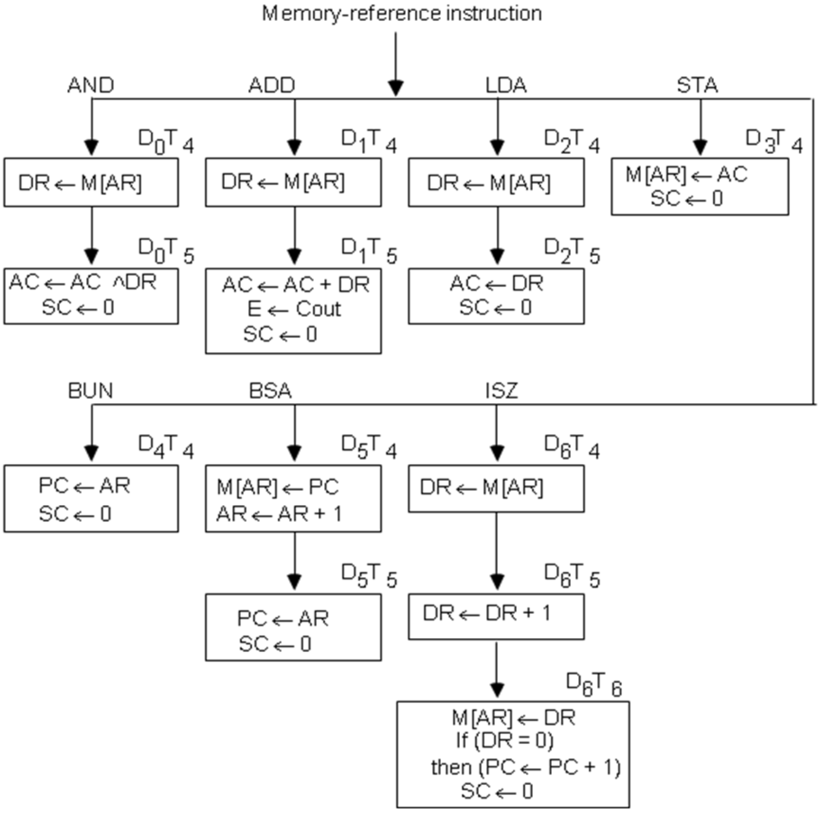


Figure 4-11 Flowchart for memory-reference instructions

4-7 Input-Output and Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device. Commercial computers include many types of input and output devices. To demonstrate the most basic requirements for input and output communication, we will use as an illustration a terminal unit with a keyboard and printer.

4-7-1 Input-Output Configuration

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. 4-12. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

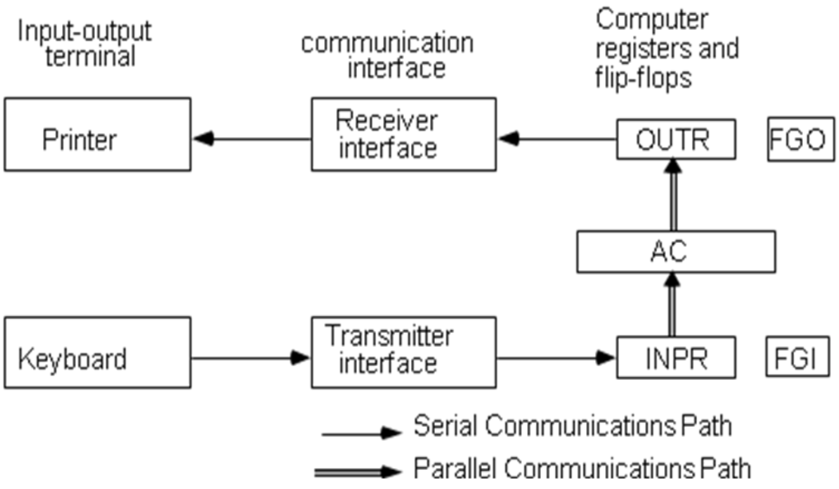


Figure 4-12 Input-Output Configuration

The input register INPR consists of eight bits and holds alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer. The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

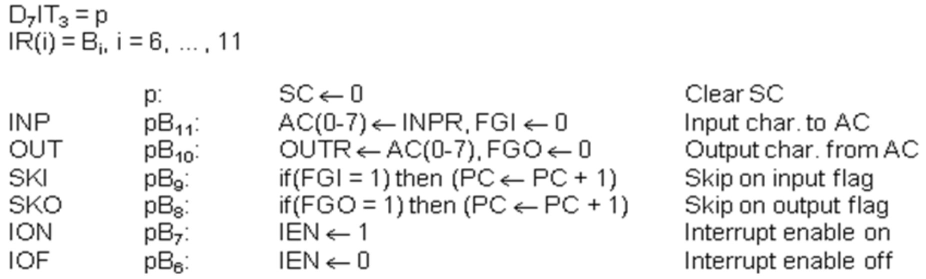
The output register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. The computer checks the flag bit; if it is 1, the information from AC is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1. The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character.

4-7-2 Input-Output Instructions

Input and output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when D_7 = 1 and I=1. The remaining bits of the instruction specify the particular operation. The control functions and microoperations for the input-output instructions are listed in Table 4-5. These instructions are executed with the clock transition associated with timing signal Tg. Each control function needs a Boolean relation D_7IT_3, which we designate for convenience by the symbol p. The control function is distinguished by one of the bits in IR(6-11). By assigning the symbol B_i to bit i of IR, all control functions can be denoted by pB_i, for i = 6 though 11. The sequence counter SC is cleared to 0 when p = D_7IT_3= 1.

The INP instruction transfers the input information from INPR into the eight low-order bits of AC and also clears the input flag to 0. The OUT instruction transfers the eight least significant bits of AC into the output register OUTR and clears the output flag to 0. The next two instructions in Table 4-5 check the status of the flags and cause a skip of the next instruction if the flag is 1. The instruction that is skipped will normally be a branch instruction to return and check the flag again. The branch instruction is not skipped if the flag is 0. If the flag is 1, the branch instruction is skipped and an input or output instruction is executed. The last two instructions set and clear an interrupt enable flip-flop IEN. The purpose of IEN is explained in conjunction with the interrupt operation.

TABLE 4-5 Input-Output Instructions



$D_7IT_3 = p$
$IR(i) = B_i, i = 6, \ldots, 11$

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$	Input char. to AC
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$	Output char. from AC
SKI	pB_9:	if(FGI = 1) then ($PC \leftarrow PC + 1$)	Skip on input flag
SKO	pB_8:	if(FGO = 1) then ($PC \leftarrow PC + 1$)	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

4-7-3 Program Interrupt

The process of communication just described is referred to as programmed control transfer. The computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device makes this type of transfer inefficient. To see why this is inefficient, consider a computer that can go through an instruction cycle in 1 μs. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to one character every 100,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction) the flag cannot interrupt the computer. When IEN is set to 1 (with the ION instruction) the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 4-13. An interrupt flip-flop R is included in the computer. When R = 0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop R is set to 1. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

The interrupt cycle is a hardware implementation of a branch and saves return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted. This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears IEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

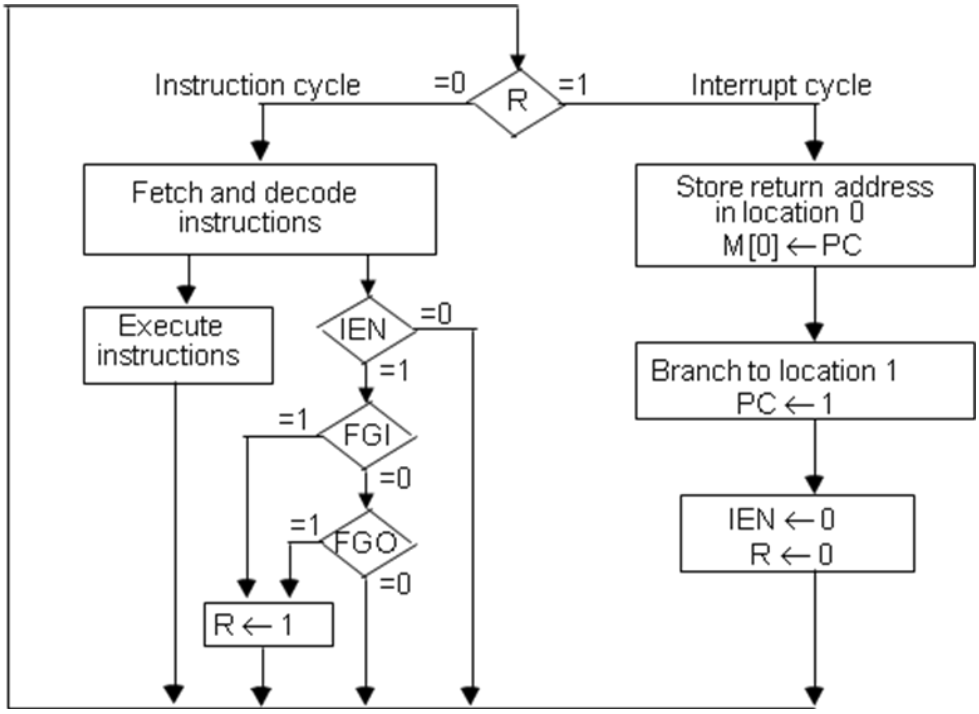


Figure 4-13 Flowchart for interrupt cycle

An example that shows what happens during the interrupt cycle is shown in Fig. 4-14. Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in Fig. 4-14(a).

When control reaches timing signal To and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 4-14(b).

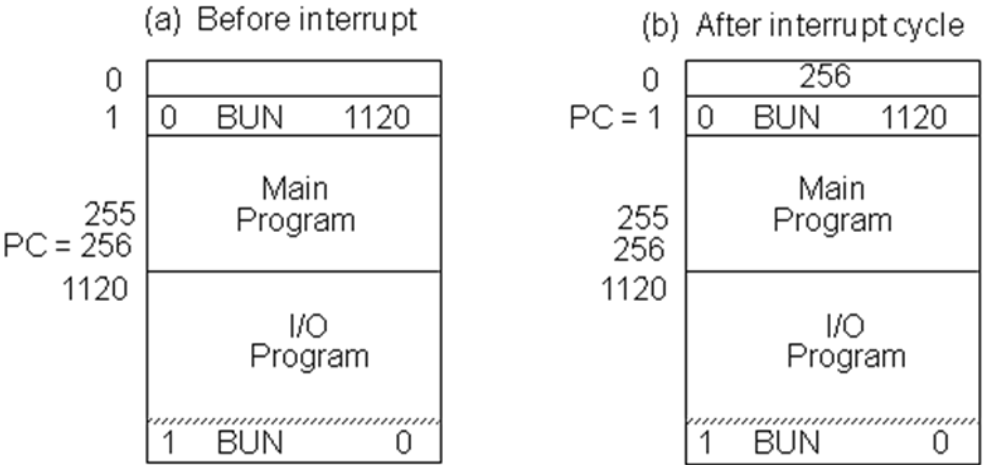


Figure 4-14 Demonstration of the interrupt cycle

The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I = 1) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

4-7-4 Interrupt Cycle

We are now ready to list the register transfer statements for the interrupt cycle. The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN = 1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T_0, T_1, or T_2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$$T_0'T_1'T_2'\ (IEN)(FGI + FGO):\quad R \leftarrow 1$$

The symbol + between FGI and FGO in the control function designates a logic OR operation. This is ANDed with IEN and $T_0'T_1'T_2'$.

e now modify the fetch and decode phases of the instruction cycle. Instead of using only timing signals T_0, T_1, and T_2 (as shown in Fig. 4-9) we will AND the three timing signals with R so that the fetch and decode phases will be recognized from the three control functions $R'T_0$, $R'T_1$, and $R'T_2$. The reason for this is that after the instruction is executed and SC is cleared to 0, the control will go through a fetch phase only if R = 0. Otherwise, if R = 1, the control will go through an interrupt cycle. The interrupt cycle stores the return address (available in PC) into memory location 0, branches to memory location 1, and clears IEN, R, and SC to 0. This can be done with the following sequence of microoperations:

RT_0: $AR \leftarrow 0$, $TR \leftarrow PC$

RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$

RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

During the first timing signal AR is cleared to 0, and the content of PC is transferred to the temporary register TR. With the second timing signal, the return address is stored in memory at location 0 and PC is cleared to 0. The third timing signal increments PC to 1, clears IEN and R, and control goes back to T_o by clearing SC to 0. The beginning of the next instruction cycle has the condition RT_o and the content of PC is equal to 1. The control then goes through an instruction cycle that fetches and executes the BUN instruction in location 1.

4-9 Summary

In this lesson, we have introduced you to vrious concepts relative to an instruction. We have discussed about basic instruction codes, operation codes, type of operands and operation in instructions and various addressing modes. We have also hihlighted the basic issue while desing instruction formats and presented details on the instruction sets. We have also discuss the inerrupt cycle.

4-10 Keywords

Instruction code: An instruction code is a group of bits that instruct the computer to perform a specific operation.

Operation code: The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.

Opcode: operation code abbreviated as opcode.

Accumulator: Computers that have a single-processor register usually assign to it the name accumulator and label it AC.

Effective address: An effective address is the value which is used by a fetch or store operation to specify which memory location is to be accessed by the operation from the perspective of the entity issuing the operation.

Program counter: A register in the central processing unit that contains the address of the next instruction to be executed. The PC is automatically incremented after each instruction is fetched to point to the following instruction.

Memory address: In computer science, a memory address is an identifier for a memory location, at which a computer program or a hardware device can store a piece of data for later retrieval.

Hardwired control: Hardwired control is a control mechanism to generate control signals by using appropriate finite state machine

4-11 Exercise

1**.** A computer uses a memory unit with 256K words of 32 bits each; A binary instruction code is stored in one word of memory. The instruction has four parts: an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.

a. How many bits are there in the operation code, the register code part, and the address part?

b. Draw the instruction word format and indicate the number of bits in each part.

c. How many bits are there in the data and address inputs of the memory?

2. What is the difference between a direct and an indirect address instruction? How many references to memory are needed for each type of instruction to bring an operand into a processor register?

3. The following register transfers are to be executed in the system of Fig. 4-4. For each transfer, specify: (1) the binary value that must be applied to bus select inputs *S2, S1, and S0;* (2) the register whose LD control input must be active (if any); (3) a memory read or write operation (if needed); and (4) the operation in the adder and logic circuit (if any).

a. AR←PC

b. IR←M[AR]

c. M[AR] ← TR

d. *AC* ← *DR, DR* ← *AC* (done simultaneously)

4. The content of *A C* in the basic computer is hexadecimal A937 and the initial value of E is 1. Determine the contents *of AC, E, PC, AR,* and IR hexadecimal after the execution of the CLA instruction. Repeat 11 more times, starting from each one of the register-reference instructions. The initial value of *PC is* hexadecimal 021.

5. An instruction at address 021 in the basic computer has /= 0, an operation code of the AND instruction, and an address part equal to 083 (all numbers are in hexadecimal). The memory word at address 083 contains the operand B8F2 and the content of AC is A937. Go over the instruction cycle and determine the contents of the following registers at the end of the execute phase: PC, AR, DR, AC, and IR. Repeat the problem six more times starting with an operation code of another memory-reference instruction.

4-12 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation And Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

5. COMPUTER DESCRIPTION

Structure

Objectives

At the end of the lesson you will be able to:

- Discuss the basic organisation of computer
- Discuss the organisation of accumulator logic

5-1 Introduction

Instead of using a flowchart, we can describe the operation of the computer with a list of register transfer statements. This is done by accumulating all the control functions and microoperations in one table. The entries in the table are taken from Figs. 4-11 in lesson 4 and Fig. 5-1, and Tables 4-3 and 4-5.

The control functions and microoperations for the entire computer are summarized in Table 5-1. The register transfer statements in this table describe in a concise form the internal organization of the basic computer. They also give all the information necessary for the design of the logic circuits of the computer. The control functions and conditional control statements listed in the table formulate the Boolean functions for the gates in the control unit. The list of microoperations specifies the type of control inputs needed for the registers and memory. A register transfer language is useful not only for describing the internal organization of a digital system but also for specifying the logic circuits needed for its design.

NOTES

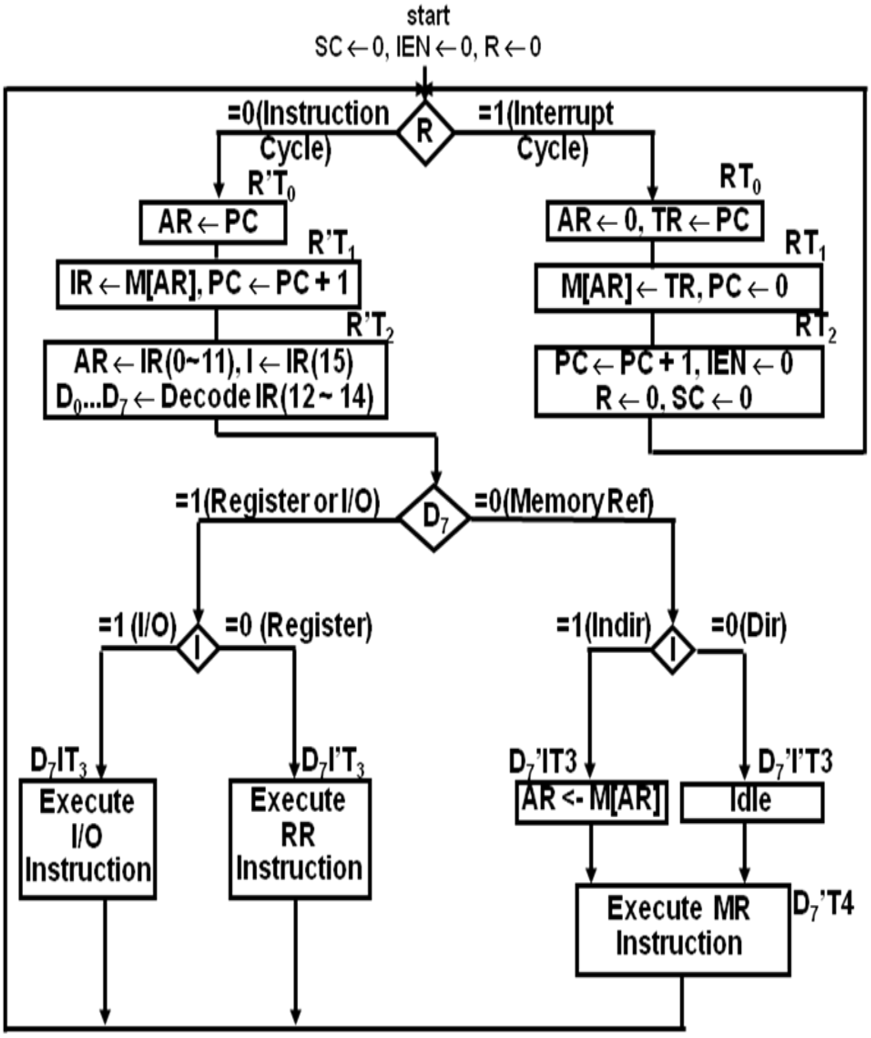


Figure 5-1 Flowchart for computer operation

5-2 Design of Basic Computer

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each

2. Nine registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

3. Seven flip-flops: I, S, E, R, IEN, FGI, and FGO

4. Two decoders: a 3 X 8 operation decoder and a 4 X 16 timing decoder

5. A 16-bit common bus

6. Control logic gates

7. Adder and logic circuit connected to the input of AC

The memory unit is a standard component that can be obtained readily from a commercial source. The flip-flops can be either of the *D or JK type,* The common bus system can be constructed with sixteen 8x1 multiplexers in a configuration. We are now going to show how to design the control logic gates. The next section deals with the design of the adder and logic circuit associated with *AC.*

5-2-1 Control Logic Gates

The block diagram of the control logic gates is shown in Fig. 4-6. The inputs to this circuit come from the two decoders, the I flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic are: A C bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC (15); DR bits 0 through 15 to check if DR = 0; and the values of the seven flip-flops. The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers

2. Signals to control the read and write inputs of memory

3. Signals to set, clear, or complement the flip-flops

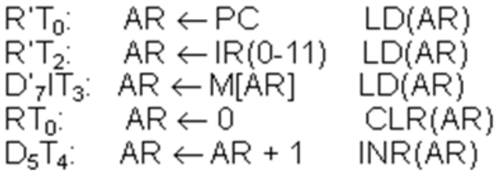
4. Signals for S_2 S_1 and S_0 to select a register for the bus

5. Signals to control the A C adder and logic circuit

The specifications for the various control signals can be obtained directly from the list of register transfer statements in Table 5-1.

5-2-2 Control of Registers and Memory

The registers of the computer connected to a common bus system are shown in Fig. 4-4. The control inputs of the registers are LD (load), INR (increment), and CLR (clear). Suppose that we want to derive the gate structure associated with the control inputs of AR. We scan Table 5-1 to find all the statements that change the content of AR:



$R'T_0$:	$AR \leftarrow PC$	LD(AR)
$R'T_2$:	$AR \leftarrow IR(0\text{-}11)$	LD(AR)
D'_7IT_3:	$AR \leftarrow M[AR]$	LD(AR)
RT_0:	$AR \leftarrow 0$	CLR(AR)
D_5T_4:	$AR \leftarrow AR + 1$	INR(AR)

NOTES

The first three statements specify transfer of information from a register or memory to AR. The content of the source register or memory is placed on the bus and the content of the bus is transferred into AR by enabling its LD control input. The fourth statement clears AR to 0 and last statement increments AR by 1. The control functions can be combined into three Boolean expressions as follows

$$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$$

$$CLR(AR) = RT_0$$

$$INR(AR) = D_5T_4$$

where LD (AR) is the load input of AR, CLR(AR) is the clear input of AR, and INR(AR) is the increment input of AR. The control gate logic associated with AR is shown in Fig. 5-2

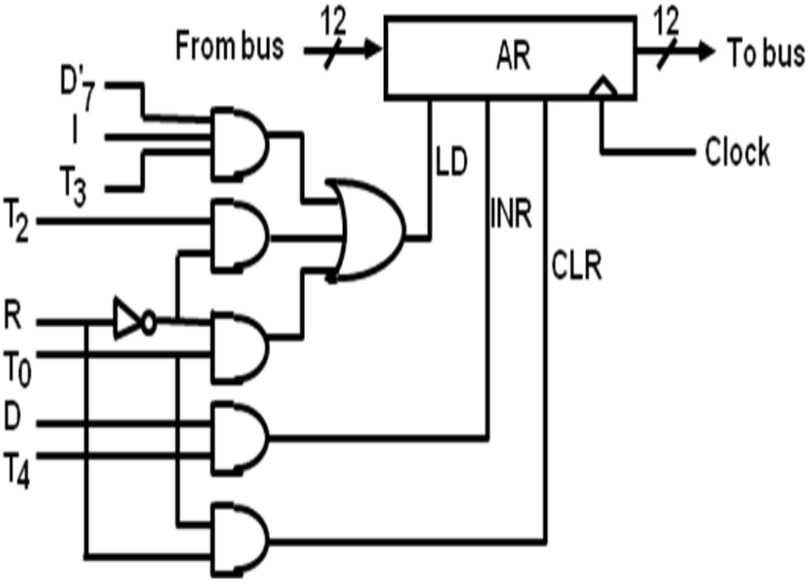
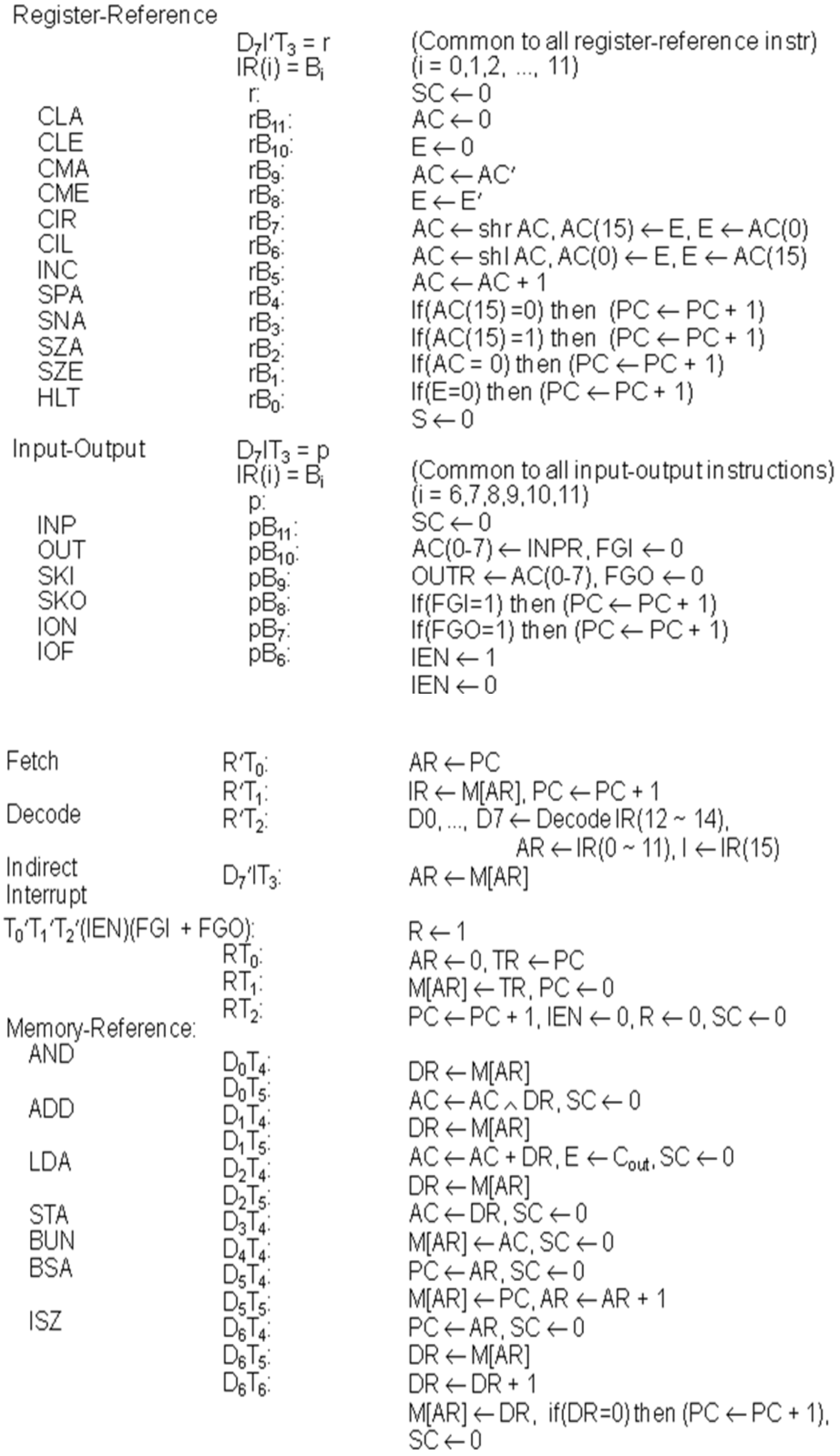


Figure 5-2 Control gates associated with AR

TABLE 5-1 Control Functions and Microoperations for the Basic Computer



Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	(i = 0,1,2, ..., 11)
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If$(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If$(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If$(AC = 0)$ then $(PC \leftarrow PC + 1)$
SZE	rB_1:	If$(E=0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-Output	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	(i = 6,7,8,9,10,11)
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$
SKI	pB_9:	If$(FGI=1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If$(FGO=1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$
Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR], PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D0, ..., D7 \leftarrow$ Decode $IR(12 \sim 14)$, $AR \leftarrow IR(0 \sim 11), I \leftarrow IR(15)$
Indirect	$D_7'IT_3$:	$AR \leftarrow M[AR]$
Interrupt		
	$T_0'T_1'T_2'(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0, TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR, PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$
Memory-Reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR, SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR, SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC, SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR, SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC, AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR, SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if$(DR=0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory. The logic gates associated with the read input of memory is derived by scanning Table 5-1 to find the statements that specify a read operation. The read operation is recognized from the symbol ←M [AR].

$$\text{Read} = R'T_1 + D'_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

The output of the logic gates that implement the Boolean expression above must be connected to the read input of memory.

5-2-3 Control of Single Flip-flops

The control gates for the seven flip-flops can be determined in a similar manner. For example, Table 5-1 shows that IEN may change as a result of the two instructions ION and IOF.

pB_7: IEN ← 1 (I/O Instruction)
pB_6: IEN ← 0 (I/O Instruction)
RT_2: IEN ← 0 (Interrupt)

$p = D_7IT_3$ (Input/Output Instruction)

If we use a JK flip-flip for IEN, the control gate logic will be as shown in Fig. 5-3

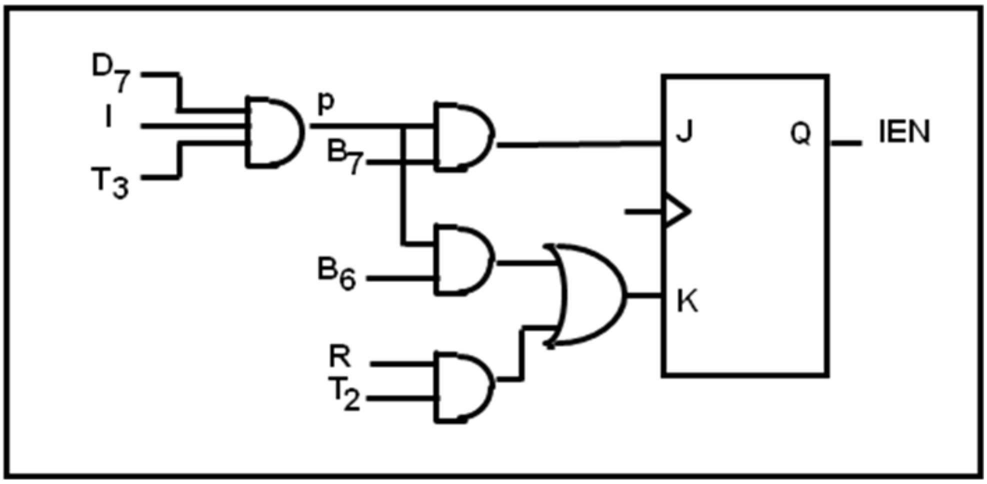


Figure 5-3 Control input for IEN

5-2-4 Control of Common Bus

The 16-bit common bus shown in Fig. 4-4 is controlled by the selection inputs S_2,S_1, and S_0. The decimal number shown with each bus input specifies the equivalent binary number that must be applied to the selection inputs in order to select the corresponding register. Table 5-2 specifies the binary numbers for $S_2S_1S_0$ that select each register. Each binary number is associated with a Boolean variable x_1 through x_7, corresponding to the gate structure that must be active in order to select the register or memory for the bus. For example, when $x_1 = 1$, the value of $S_2S_1S_0$ must be 001 and the output of AR will be selected for the bus. Table 5-2 is recognized as the truth table of a binary encoder. The placement of the encoder at the inputs of the bus selection logic is shown in Fig. 5-3. The Boolean functions for the encoder are

$$S_0 = x_1 + x_3 + x_5 + x_7$$

$$S_1 = x_2 + x_3 + x_6 + x_7$$

$$S_1 = x_4 + x_5 + x_6 + x_7$$

To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register onto the bus. For example, to find the logic that makes $x_1 = 1$, we scan all register transfer statements in Table 5-1 and extract those statements that have AR as a source.

$$D_4T_4\text{:}\ \ PC \leftarrow AR$$
$$D_5T_5\text{:}\ \ PC \leftarrow AR$$

Therefore, the Boolean function for x_1 is

$$x1 = D_4T_4 + D_5T_5$$

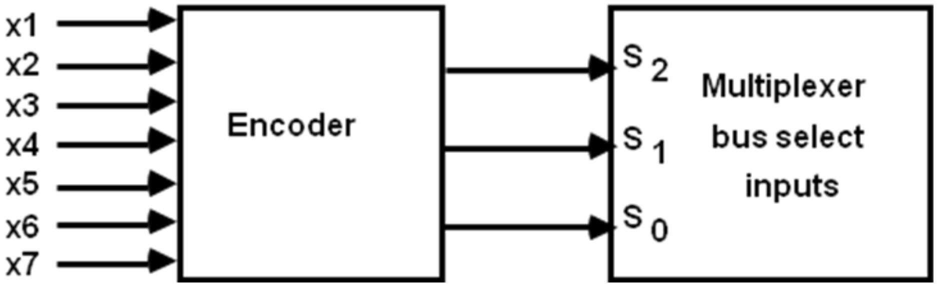
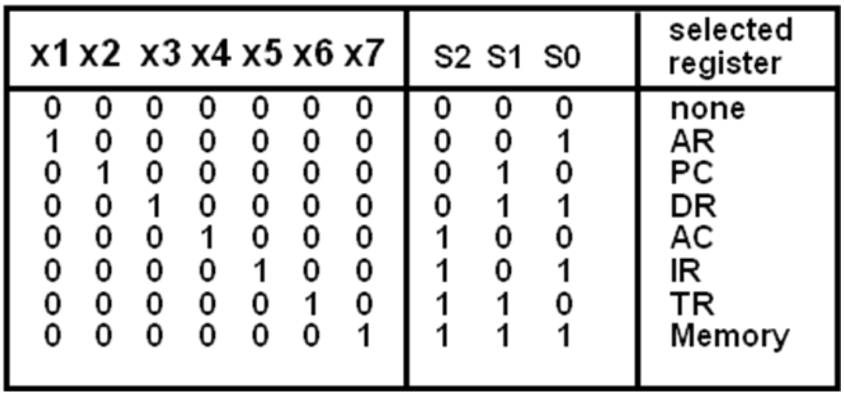


Figure 5-4 Encoder for bus selection inputs

TABLE 5-2 Encoder for Bus Selection Circuit



x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	Memory

The data output from memory are selected for the bus when $X_7 = 1$ and $S_2S_1S_0$ =111. The gate logic that generates X_7 must also be applied to the read input of memory. Therefore, the Boolean function for XJ is the same as the one derived previously for the read operation.

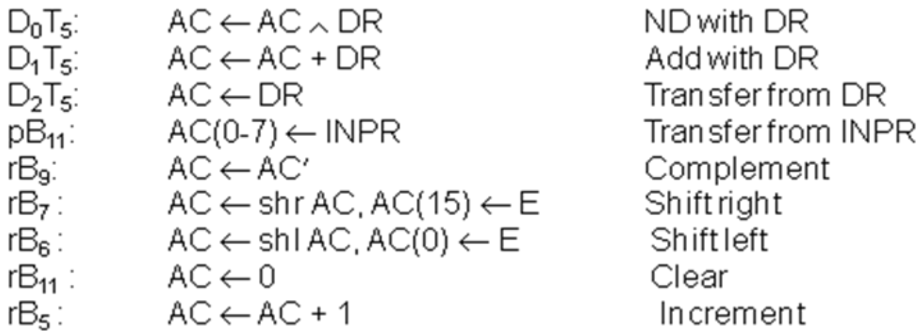
$$X_7 = R'T_1 + D'_7IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

In a similar manner we can determine the gate logic for the other registers.

5-3 Design of Accumulator Logic

The circuits associated with the reregister are shown in Fig. 5-4. The adder and logic circuit has three sets of inputs. One set of 16 inputs comes from the outputs of AC. Another set of 16 inputs comes from the data register DR. A third set of eight inputs comes from the input register INPR. The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit.

In order to design the logic associated with AC, it is necessary to go over the register transfer statements in Table 5-1 and extract all the statements that change the content of AC.



D_0T_5:	$AC \leftarrow AC \wedge DR$	ND with DR
D_1T_5:	$AC \leftarrow AC + DR$	Add with DR
D_2T_5:	$AC \leftarrow DR$	Transfer from DR
pB_{11}:	$AC(0\text{-}7) \leftarrow INPR$	Transfer from INPR
rB_9:	$AC \leftarrow AC'$	Complement
rB_7 :	$AC \leftarrow shr\ AC, AC(15) \leftarrow E$	Shift right
rB_6 :	$AC \leftarrow shl\ AC, AC(0) \leftarrow E$	Shift left
rB_{11} :	$AC \leftarrow 0$	Clear
rB_5 :	$AC \leftarrow AC + 1$	Increment

From this list we can derive the control logic gates and the adder and Logic circuit.

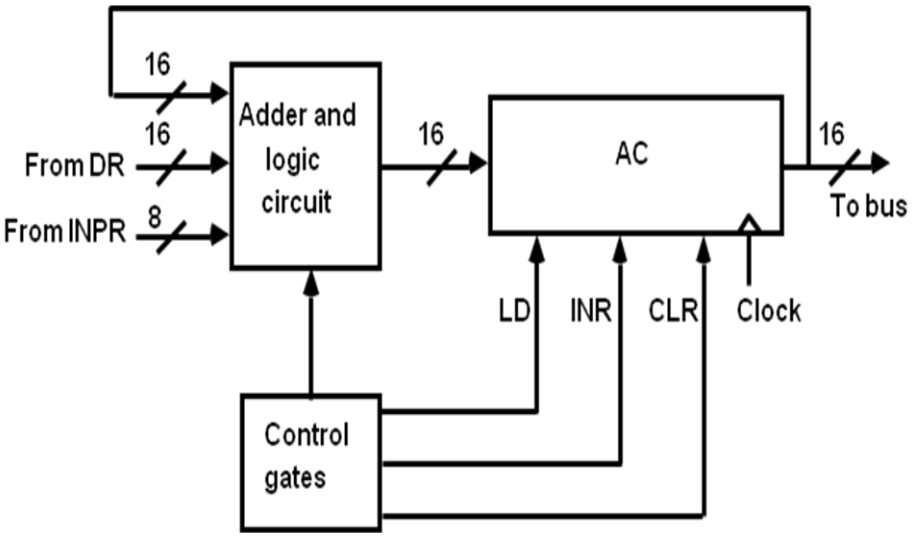


Figure 5-4 Circuit associated with AC

5-3 -1 Control of AC Register

The gate structure that controls the LD, INR, and CLR inputs of AC is shown in Fig. 5-5. The gate configuration is derived from the control functions in the list above. The control function for the clear microoperation is rB_{11}, where $r = D_7I'T_3$ and rB_{11} = IR (11). The output of the AND gate that generates this control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment microoperation is connected to the INR input of the register. The other seven microoperations are generated in the adder and logic circuit and are loaded into AC at the proper time. The outputs of the gates for each control function are marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

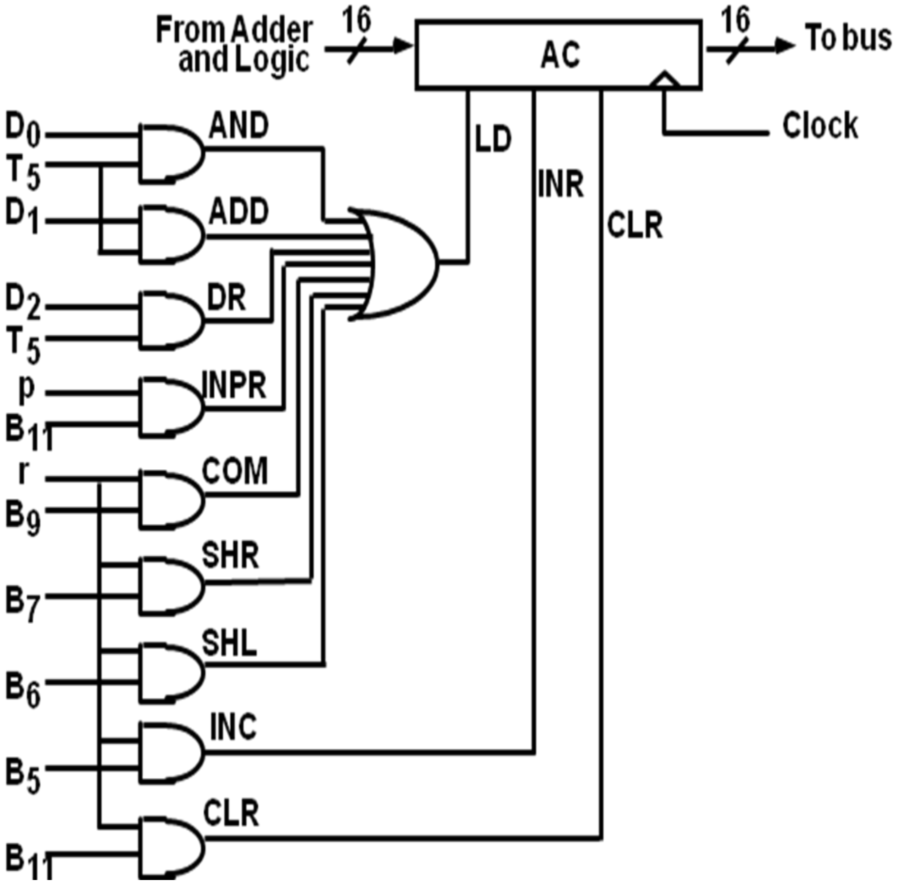


Figure 5-5 Gate structures for controlling the LD, INR, and CLR of A

5-3 -2 Adder and Logic Circuit

The adder and logic circuit can be subdivided into 16 stages, with each stage corresponding to one bit of AC. The load (LD) input is connected to the inputs of the AND gates. Figure 5-6 shows one such AC register stage (with the OR gates removed). The input is labeled I_i, and the output AC (i). When the LD input is enabled, the 16 inputs I, for i= 0, 1, 2,. . . , 15 are transferred to AC (0-15).

One stage of the adder and logic circuit consists of seven AND gates, one OR gate and a full-adder (FA), as shown in Fig. 5-6. The inputs of the gates with symbolic names come from the outputs of gates marked with the same symbolic name in Fig. 5-5. For example, the input marked ADD in Fig. 5-5 is connected to the output marked ADD in Fig. 5-5.

The AND operation is achieved by ANDing AC (i) with the corresponding bit in the data register DR (i). One stage of the adder uses a full-adder with the corresponding input and output carries. The transfer from INPR to AC is only for bits 0 through 7. The complement microoperation is obtained by inverting the bit value in AC. The shift-right operation transfers the bit from AC (i+ 1), and the shift-left operation transfers the bit from AC (i - 1). The complete adder and logic circuit consists of 16 stages connected together.

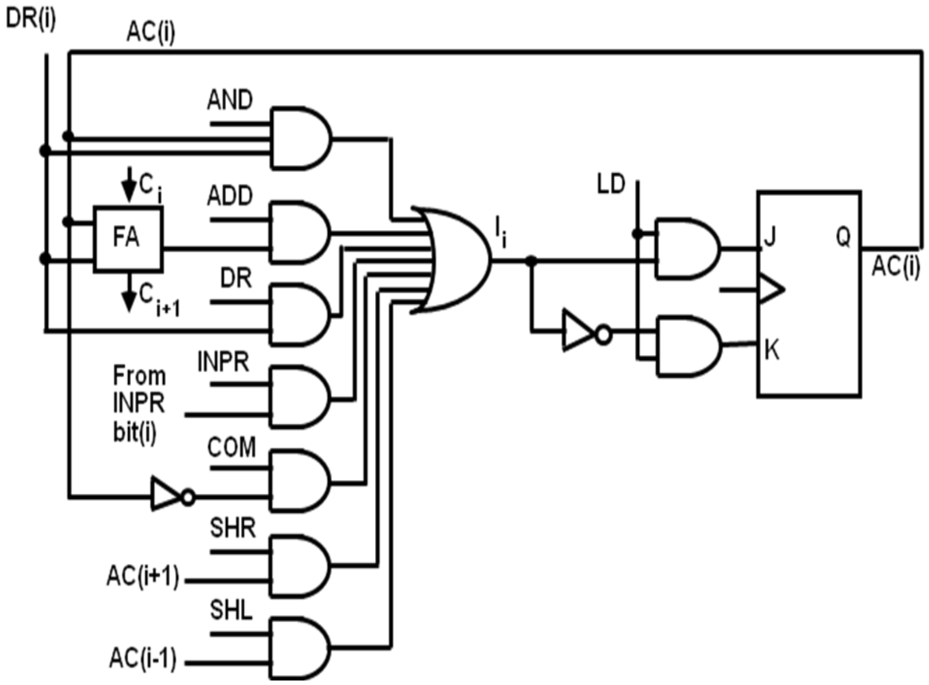


Figure 5-6 One stage of Adder and Logic circuit

5-4 Summary

In this lesson, we have discussed about two main components of the organisation, the design of basic computer and design of accumulator logic unit. We have explained the concept of the memory unit, registers, flip-flops and decoders. We have also introduce the concept of accumulator Logic unit

5-5 Keywords

Flow chart: A flowchart is common type of chart, represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.

Control unit: The control unit is the circuitry that controls the flow of information through the processor, and coordinates the activities of the other units within it

Flip-Flops: In digital circuits, a flip-flop is a term referring to an electronic circuit (a bistable multivibrator) that has two stable states and thereby is capable of serving as one bit of memory.

Control logic: Control logic is the part of a software architecture that controls what the program will do. This part of the program is also called the controller.

5-6 Exercise

1. A digital computer has a memory unit with a capacity of 16,384 words, 40 bits per word. The instruction code format consists of six bits for the operation part and 14 bits for the address part (no indirect mode bit). Two instructions are packed in one memory word, and a 40-bit instruction register *IR* is available in the control unit. Formulate a procedure for fetching and executing instructions for this computer.

2. Derive the control gates associated with the program counter *PC* in the basic computer.

3. Derive the control gates for the write input of the memory in the basic computer.

4. Show the complete logic of the interrupt flip-flops *R* in the basic computer. Use a JK flip-flop and minimize the number of gates.

5. Derive the Boolean logic expression for x_2 (see Table 5-1). Show that x_2 can be generated with one AND gate and one OR gate.

6. Derive the Boolean expression for the gate structure that clears the sequence counter *SC* *to 0.* Draw the logic diagram of the gates and show how the output is connected to the INR and CLR inputs of *SC*. Minimize the number of gates.

5-7 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

UNIT - II

1. MICROPROGRAMMED CONTROL

Structure

Objectives

At the end of the lesson you will be able to:

- Define the microprogrammed control unit
- Identify types and formats of microinstruction
- Explain the working of a microprogrammed control unit
- Discuss microprogram example

1-1 Introduction

The major functional parts in a digital computer are Central Processing Unit (CPU), Memory, and Input-output. The main digital hardware functional units of CPU are control unit, arithmetic and logic unit, and registers. The function of the control unit in a digital computer is to initiate sequences of microoperations. The number of different types of microoperations that are available in a given system is finite. The complexity of the digital system is derived from the number of sequences of microoperations that are performed. Two methods of implementing control unit are hardwired control and microprogrammed control. The design of

NOTES

hardwired control involves the use of fixed instructions, fixed logic blocks of and/or arrays, encoders, decoders, etc. The key characteristics of hardwired control logic are high-speed operation, expensive, relatively complex, and no flexibility of adding new instructions. Example CPUs with hardwired logic control are Intel 8085, Motorola 6802, Zilog 80, and any RISC (Reduced Instruction Set Computer) CPUs. When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be *hardwired.* Microprogramming is a second alternative for designing the control unit of a digital computer. The principle of microprogramming is an elegant and systematic method for controlling the microoperation sequences in a digital computer. For example, CPUs with microprogrammed control unit are Intel 8080, Motorola 68000, and any CISC (Complex Instruction Set Computer) CPUs.

The control function that specifies a microoperation is a binary variable. When it is in one binary state, the corresponding microoperation is executed. *A* control variable in the opposite binary state does not change the state of the registers in the system. The active state of a control variable may be either the 1 state or the 0 state, depending on the application. In a bus-organized system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and arithmetic logic units.

The control unit initiates a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated, while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a control word. As such, control words can be programmed to perform various operations on the components of the system. A control unit whose binary control variables are stored in memory is called a *microprogrammed control unit.* Each word in control memory contains within it a *microinstruction.* The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes a *microprogram.* Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM). The content of the words in ROM are fixed and cannot be altered by simple programming since no writing capability is available in the ROM. ROM words are made permanent during the hardware production of the unit. The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies a microinstruction.

A more advanced development known as *dynamic* microprogramming permits a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory that is part of a control unit is referred to as a *control memory.*

A computer that employs a microprogrammed control unit will have two separate memories: a main memory and a control memory. The main memory is available to the user for storing the programs. The contents of main memory may alter when the data are manipulated and every time that the program is changed. The user's program in main memory consists of machine instructions and data. In contrast, the control memory holds a fixed microprogram that cannot be altered by the occasional user. The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations. Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the microoperations to fetch the instruction from main memory; to evaluate the effective address, to execute the operation specified by the instruction, and to return control to the fetch phase in order to repeat the cycle for the next instruction.

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Fig. 1-1. The control memory is assumed to be a ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.

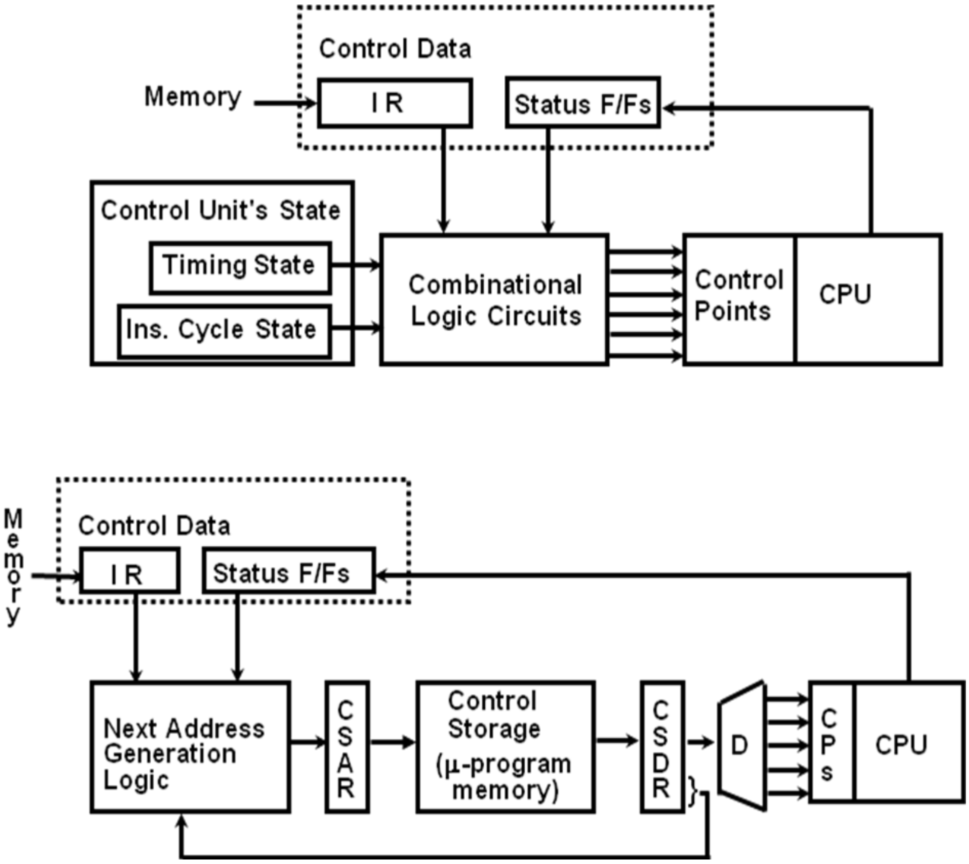


Figure 1-1 Microprogrammed control organisation

The microinstruction contains a control word that specifies one or *more* micro-operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory.

The next address generator is sometimes called a microprogram *sequencer,* as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways, depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a *pipeline register.* It allows the execution of the microoperations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly from the control memory. It must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory. Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

The main advantage of the microprogrammed control is the fact that once the hardware configuration is established there should be no need for further hardware or wiring changes. If we want to establish a different

NOTES

control sequence for the system, all we need to do is specify a different set of microinstructions for control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

1-2 Address Sequencing

Microinstructions are stored in control memory in groups, with each group specifying a *routine.* Each computer instruction has its own microprogram routine in control memory to generate the microoperations that execute the instruction. The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another. To appreciate the address sequencing in a microprogram control unit, let us enumerate the steps that the control must undergo during the execution of a single computer instruction.

An initial address is loaded into the control address register when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine. The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions. At the end of the fetch routine, the instruction is in the instruction register of the computer.

The control memory next must go through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers. The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction. When the effective address computation routine is completed, the address of the operand is available in the memory address register.

The next step is to generate the microoperations that execute the instruction fetched from memory. The microoperation steps to be generated in processor registers depend on the operation code part of the instruction. Each instruction has its own microprogram routine stored in a given location of control memory. The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a *mapping* process. A mapping procedure is a rule that transforms the instruction code into a control memory address. Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register, but sometimes the sequence of microoperations will depend on values of certain status bits in processor registers. Microprograms that employ subroutines will require an external register for storing the return address. Return addresses cannot be stored in ROM because the unit has no writing capability.

When the execution of the instruction is completed, control must return to the fetch routine. This is accomplished by executing an unconditional branch microinstruction to the fast address of the fetch routine. In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.

2. Unconditional branch or conditional branch, depending on status bit conditions.

3. A mapping process from the bits of the instruction to an address for control memory.

4. A facility for subroutine call and return.

Figure 1-2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address. The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained. The diagram shows four different paths from which the control address register (CAR) receives the address. The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence. Branching is achieved by specifying the branch address in one of the fields of the microinstruction. Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition. An external address is transferred into control memory via a mapping logic circuit. The return address for a subroutine is stored in a special register whose value is then used when the microprogram wishes to return from the subroutine.

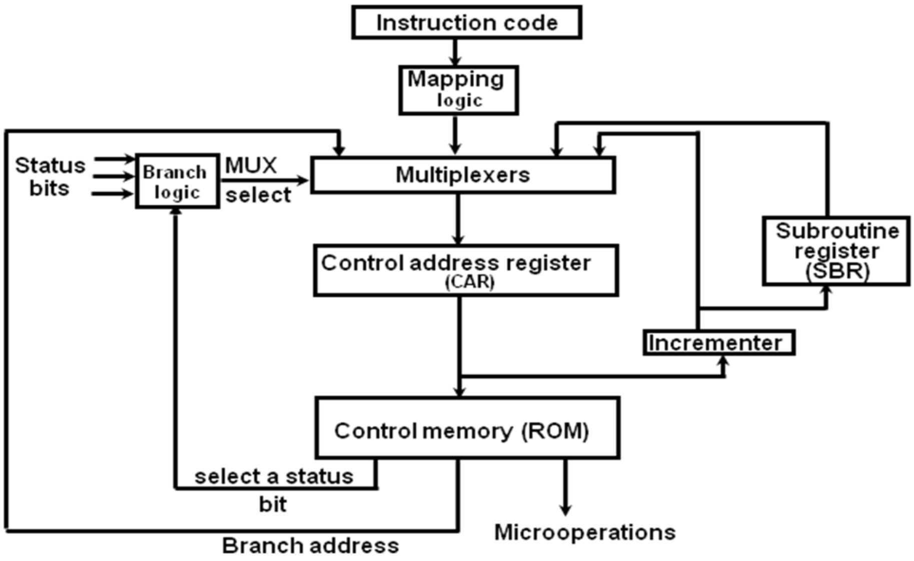


Figure 1-2 Selection of address for control memory

1-2-1 Conditional Branching

The branch logic of Fig. 1-2 provides decision-making capabilities in the control unit. The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions. Information in these bits can be tested and actions initiated based on their condition: whether their value is 1 or 0. The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.

The branch logic hardware may be implemented in a variety of ways. The| simplest way is to test the specified condition and branch to the indicated address if the condition is met; otherwise, the address register is incremented. This can be implemented with a multiplexer. Suppose that there are eight status| bit conditions in the system. Three bits in the microinstruction are used to specify any one of eight status bit conditions. These three bits provide the selection variables for the multiplexer. If the selected status bit is in the 1 state, the output of the multiplexer is 1; otherwise, it is 0. A 1 output in the multiplexer genera a control signal to transfer the branch address from the microinstruction into t control address register. A 0 output in the multiplexer causes the address register to be incremented. In this configuration, the microprogram follows one of two possible paths, depending on the value of the selected status bit.

An unconditional branch microinstruction can be implemented by loading the branch address from control memory into the control address register. This can be accomplished by fixing the value of one status bit at the input of the multiplexer, so it is always equal to 1. A reference to this bit by the status bit select lines from control memory causes the branch address to be loaded into the control address register unconditionally.

1-2-2 Mapping of Instruction

A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located. The status bits for this type of branch are the bits in the operation code part of the instruction. For example, a computer with a simple instruction format as shown in Fig. 1-3 has an operation code of four bits which can specify up to 16 distinct instructions. Assume further that the control memory has 128 words, requiring an address of seven bits. For each operation code there exists a microprogram routine in control memory that executes the instruction. One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in Fig. 7-3. This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register. This provides for each computer instruction a microprogram routine with a capacity of four microinstructions. If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.

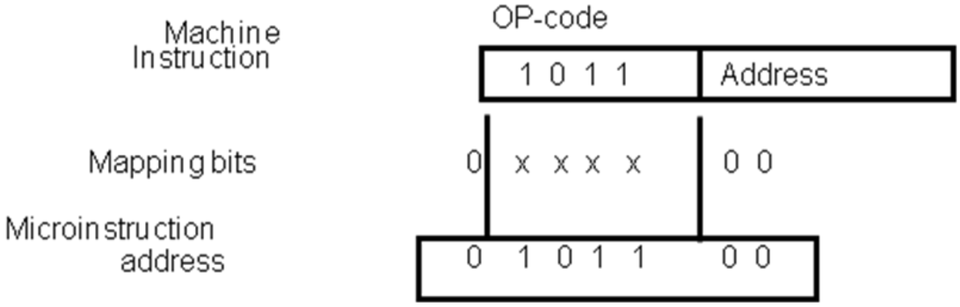


Figure 1-3 Mapping from instruction code to microinstruction address

One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function. In this configuration, the bits of the instruction specify the address of a mapping ROM. The contents of the mapping ROM give the bits for the control address register. In this way the microprogram routine that executes the instruction can be placed in any desired location in control memory. The mapping concept provides flexibility for adding instructions for control memory as the need arises.

The mapping function is sometimes implemented by means of an integrated circuit called programmable logic device or PLD. A PLD is similar to ROM in concept except that it uses AND and OR gates with internal electronic fuses. The interconnection between inputs, AND gates, OR gates, and outputs can be programmed as in ROM. A mapping function that can be expressed in terms of Boolean expressions can be implemented conveniently with a PLD.

1-2-3 Subroutines

Subroutines are programs that are used by other routines to accomplish a particular task. A subroutine can be called from any point within the main body of the microprogram. Frequently, many microprograms contain identical sections of code. Microinstructions can be saved by employing subroutines that use common sections of microcode. For example, the sequence of microoperations needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. This sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

Microprograms that use subroutines must have a provision for storing the return address during a subroutine call and restoring the address during a subroutine return. This may be accomplished by placing the incremented out-subroutine register put from the control address register into a subroutine register and branching to the beginning of the subroutine. The subroutine register can then become the source for transferring the address for the return to the main routine. The best way to structure a register file that stores addresses for subroutines is to organize the registers in a last-in, first-out (LIFO) stack.

1-3 Microprogram Example

Once the configuration of a computer and its microprogrammed control unit, is established, the designer's task is to generate the microcode for the control, memory. This code generation is called microprogramming and is a process similar to conventional machine language programming. To appreciate this process, we present here a simple digital computer and show how it is microprogrammed.

1-3-1 Computer Configuration

The block diagram of the computer is shown in Fig. 1-4. It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing the microprogram. Four registers are associated with the processor unit and two with the control unit. The processor registers are program counter *PC,* address register *AR,* data register *DR,* and accumulator register *AC.* The control unit has a control address register *CAR* and a subroutine register *SBR.* The control memory and its registers are organized as a microprogrammed control unit, as shown in Fig. 1-4.

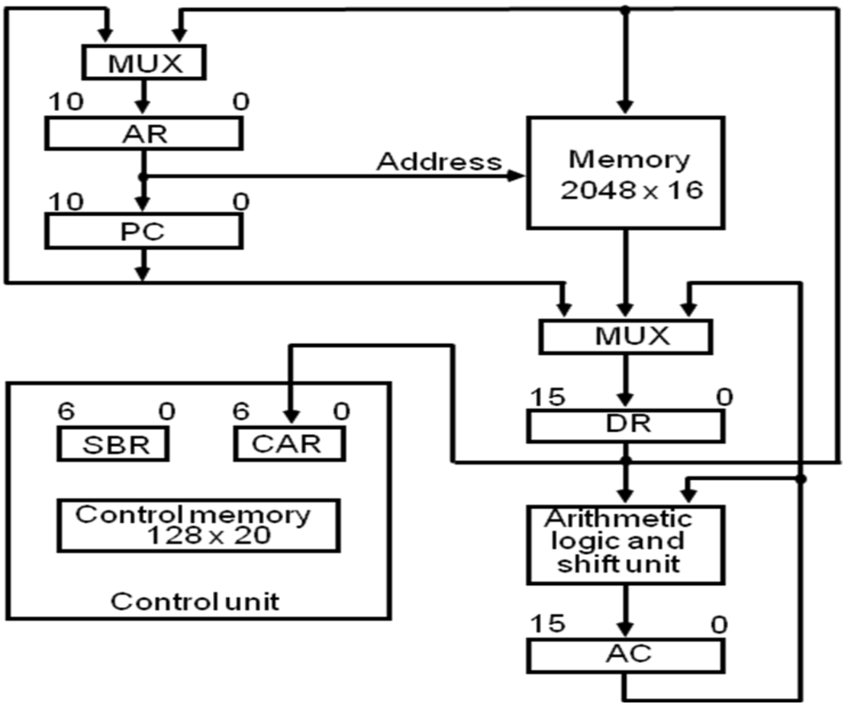
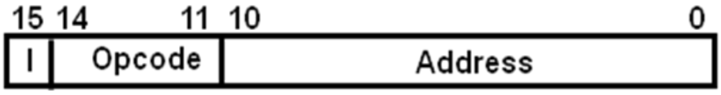


Figure 1-4 Computer hardware Configuration

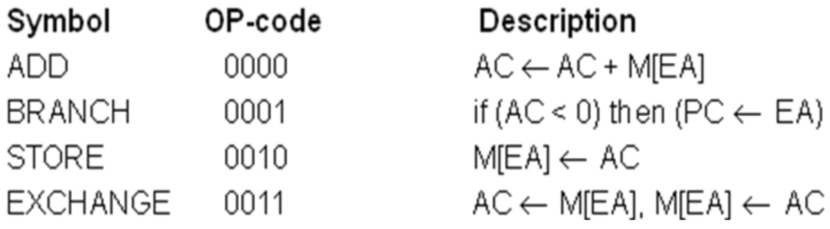
The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. *DR* can receive information from *AC, PC,* or memory. *AR* can receive information from .PC or *DR. PC can* receive information only from *AR.* The arithmetic, logic, and shift unit performs microoperations with data from *AC* and *DR* and places the result in *AC. Note* that memory receives its address from *AR.* Input data written to memory come from *DR,* and data read from memory can go only to *DR.*

The computer instruction format is depicted in Fig. 1-5 (a). It consists of three fields: a 1-bit field for indirect addressing symbolized by I, a 4-bit operation code (opcode), and an 11-bit address field. Figure 1-5 (b) lists four of the 16 possible memory-reference instructions. The ADD instruction adds the content of the operand found in the effective address to the content *of AC. The* BRANCH instruction causes a branch to the effective address if the operand in *AC is* negative. The program proceeds with the next consecutive instruction if *AC is* not negative. The *AC is* negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between *AC and* the memory word specified by the effective address.

It will be shown subsequently that each computer instruction must be microprogrammed. In order not to complicate the microprogramming example, only four instructions are considered here. It should be realized that 12 other instructions can be included and each instruction must be microprogrammed by the procedure outlined below.



(a) Machine instruction format



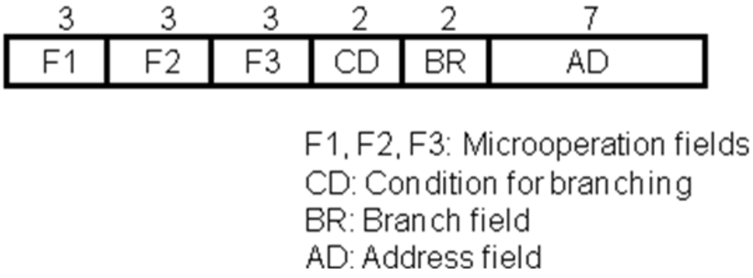
Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

(b) Four computer instructions

Figure1-5 computer instruction format

1-3-2 Microinstruction Format

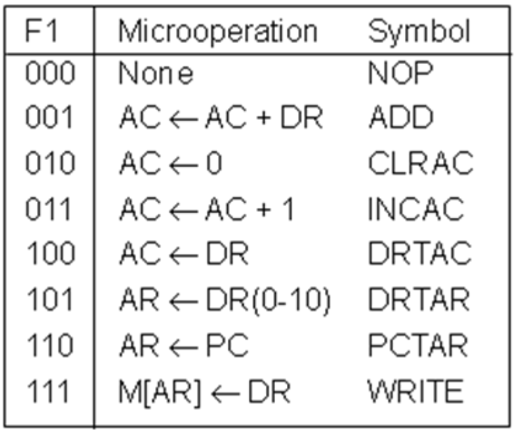
The microinstruction format for the control memory is shown in Fig.1-6. 20 bits of the microinstruction are divided into four functional parts. The fields Fl, F2, and F3 specify microoperations for the computer. The CD field selects status bit conditions. The BR field specifies the type or branch to be used. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

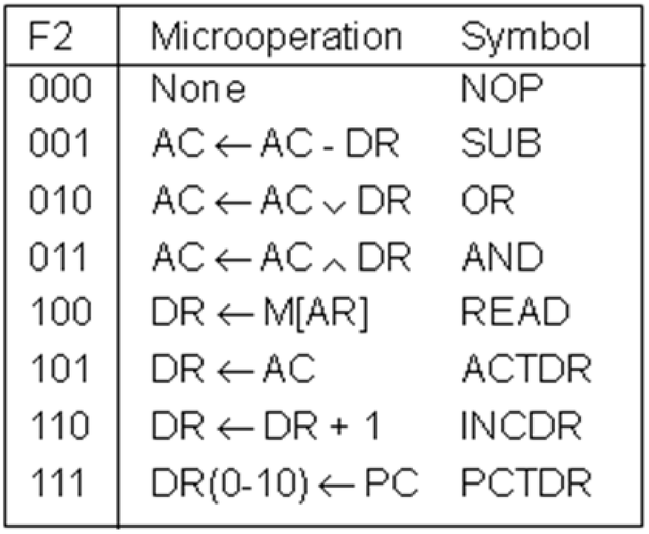
Figure 1-6 Macroinstruction code format (20 bits)

The microoperations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct microoperations as listed in Table 1-1. This gives a total of 21 microoperations. No more than three microoperations can be chosen for a microinstruction, one from each field. If fewer than three microoperations are used, one or more of the fields will use the binary code 000 for no operation. As an illustration, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from Fl.

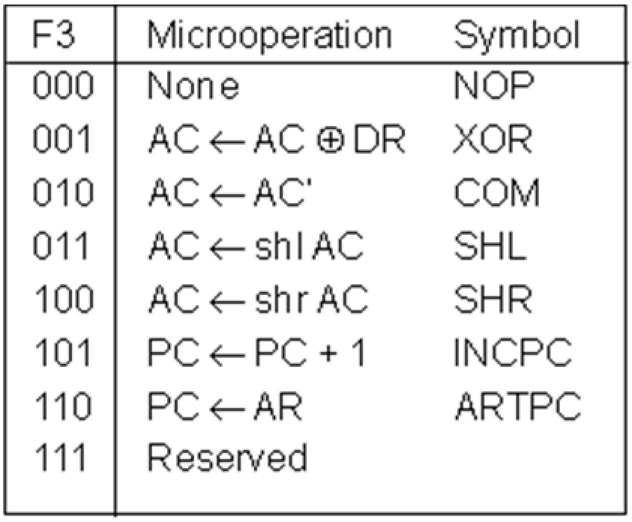


F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

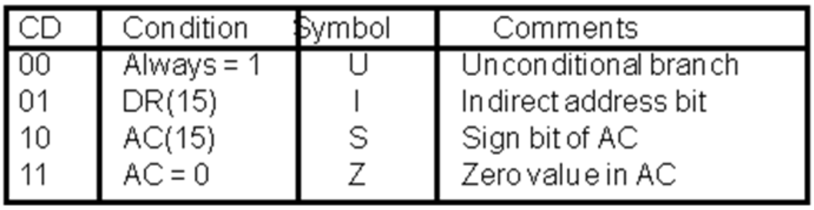
NOTES



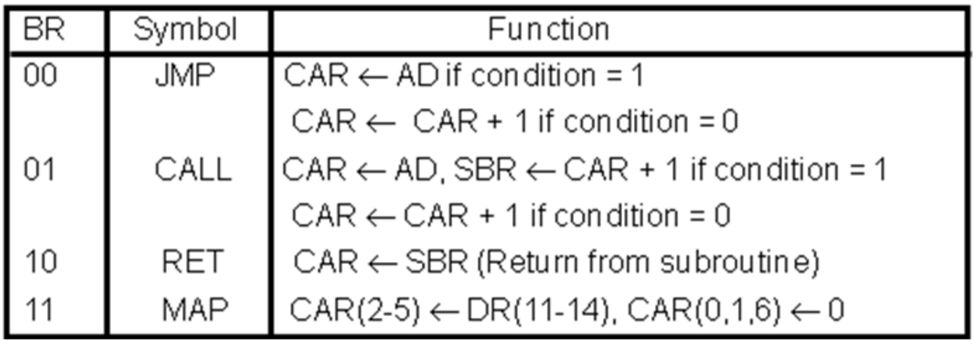
F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR



F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	



CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC



BR	Symbol	Function

00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0

01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0

10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

TABLE 1-1 Symbols and Binary Code for Microinstruction Fields

DR ← M [AR] with F2 =100

PC ← PC + 1 with F3 =101

The nine bits of the micro-operation fields will then be 000 100 101. It is important to realize that two or more conflicting microoperations cannot be specified simultaneously. For example, a microoperation field 010 001 000 has no meaning because it specifies the operations to clear *AC* to 0 and subtract *DR* from *AC* at the same time.

Each microoperation in Table 1-1 is defined with a register transfer statement and is assigned a symbol for use in a symbolic microprogram. All transfer-type micro-operations symbols use five letters. The first two letters designate the source register, the third letter is always a T, and the last two letters designate the destination register. For example, the microoperation that specifies the transfer *AC* ← *DR* (Fl = 100) has the symbol DRTAC, which stands for a transfer from *DR* to *AC.*

The *CD* (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 1-1. The first condition is always a 1, so that a reference to *CD = 00* (or the symbol U) will always find the condition to be true. When this condition is used in conjunction with the BR (branch) field, it provides an unconditional branch operation. The indirect bit I is available from bit 15 of *DR* after an instruction is read from memory. The sign bit of *AC* provides the next status bit. The zero value, symbolized by *Z,* is a binary variable whose value is equal to 1 if all the bits in *AC* are equal to zero. We will use the symbols U, I, S, and Z for the four status bits when we write microprograms in symbolic form.

The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction. As shown in Table 7-1, when BR = 00, the control performs a jump (JMP) operation (which is similar to a branch), and when BR = 01, it performs a call to subroutine (CALL) operation. The two operations are identical except that a call microinstruction stores the return address in the subroutine register *SBR* The jump and call operations depend on the value of the CD field. It the status bit condition specified in the CD field is equal to 1, the next address in the AD field is transferred to the control address register *CAR.* Otherwise, *CAR* is incremented by 1.

The return from subroutine is accomplished with a BR field equal to 10. This causes the transfer of the return address from *SBR to CAR.* The mapping from the operation code bits of the instruction to an address for *CAR* is accomplished when the BR field is equal to 11. This mapping is as depicted in Fig. 1-3. The bits of the operation code are in DR (ll-14) after an instruction is read from memory. Note that the List two conditions in the BR field are independent of the values in the CD and AD fields.

1-3-3 Symbolic Microinstructions

The symbols defined in Table 1-1 can be used to specify microinstructions in symbolic form. A symbolic microprogram can be translated into its binary equivalent by means of an assembler. The simplest and most straightforward way to formulate an assembly language for a microprogram is to define symbols for each field of the microinstruction and to give users the capability for defining their own symbolic addresses.

Each line of the assembly language microprogram defines a symbolic microinstruction. Each symbolic microinstruction is divided into five fields: label, microoperations, CD, BR, and AD. The fields specify the following information:

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).

2. The microoperation's field consists of one, two, or three symbols, separated by commas, from those defined in Table 1-1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no microoperations. This will be translated by the assembler to nine zeros.

3. The CD field has one of the letters U, I, S, or Z.

4. The BR field contains one of the four symbols defined in Table 1-1.

5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:

a. With a symbolic address, this must also appear as label.

b. With the symbol NEXT to designate the next address in sequence.

c. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

We will use also the pseudo instruction ORG to define the origin, or first address, of a microprogram routine. Thus the symbol ORG 64 informs the assembler to place the next microinstruction in control memory at decimal address 64, which is equivalent to the binary address 1000000

1-3-4 Fetch Routine

The control memory has 128 words, and each word contains 20 bits. To microprogram the control memory, it is necessary to determine the bit values of each of the 128 words. The first 64 words (addresses 0 to 63) are to be occupied by the routines for the 16 instructions. The last 64 words may be used for any other purpose. A convenient starting location for the fetch routine is address 64. The microinstructions needed for the fetch routine are

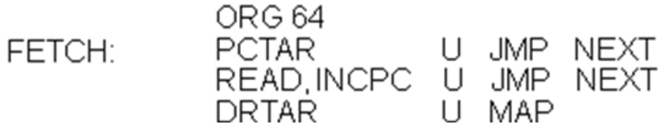
AR ← PC

DR ← M [AR], PC ← PC + 1

AR ← DR (0-10), CAR (2-5) ← DR (11-14), CAR (0, 1, 6) ← 0

The address of the instruction is transferred from *PC to AR* and the instruction is then read from memory into *DR.* Since no instruction register is available, the instruction code remains in *DR.* The address part is transferred to *AR* and then control is transferred to one of 16 routines by mapping the operation code part of the instruction from *DR* into *CAR.*

The fetch routine needs three microinstructions, which are placed in control memory at addresses 64, 65, and 66. Using the assembly language conventions defined previously, we can write the symbolic microprogram for the fetch routine as follows:



```
                ORG 64
FETCH:          PCTAR        U  JMP  NEXT
                READ,INCPC   U  JMP  NEXT
                DRTAR        U  MAP
```

The translation of the symbolic microprogram to binary produces the following binary microprogram. The bit values are obtained from Table 1-1.

The three microinstructions that constitute the fetch routine have been listed in three different representations. The register transfer representation shows the internal register transfer operations that each microinstruction implements. The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory. It is customary to write microprograms in symbolic form and then use an assembler program to obtain a translation to binary.

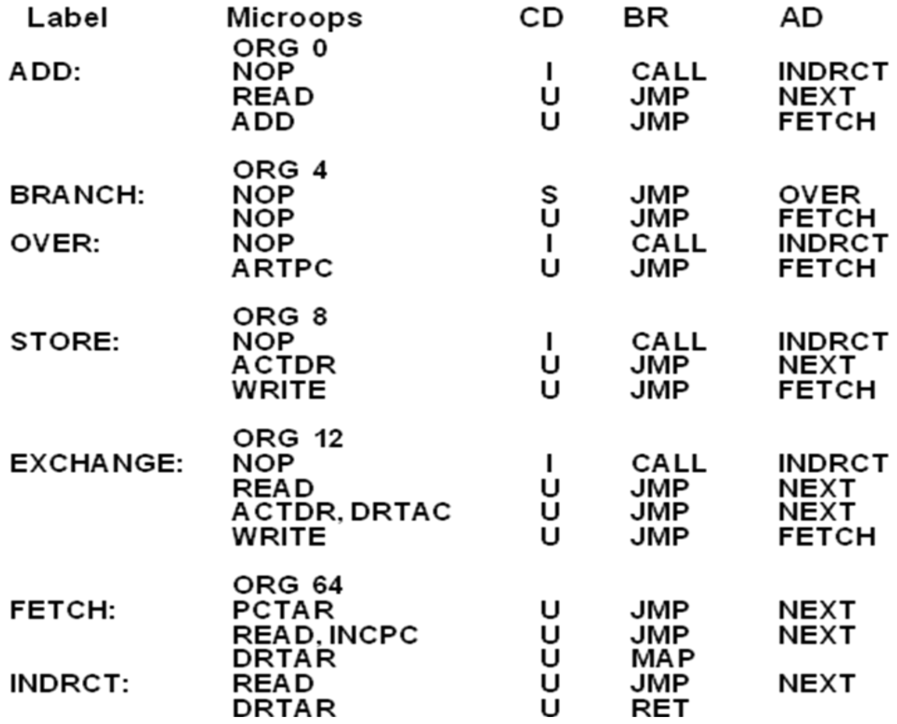
1-3-5 Symbolic Microprogram

The execution of the third (MAP) microinstruction in the fetch routine results in a branch to address OxxxxOO, where xxxx are the four bits of the operation code. For example, if the instruction is an ADD instruction whose operation code is 0000, the MAP microinstruction will transfer to *CAR* the address 0000000, which is the start address for the ADD routine in control memory. The first address for the BRANCH and STORE routines are 0 0001 00 (decimal 4) and 0 0010 00 (decimal 8), respectively. The first address of the other 13 routines is at address values 12, 16, 20, . . . , 60. This gives four words in control memory for each routine.

In each routine we must provide microinstructions for evaluating the effective address and for executing the instruction. The indirect address mode is associated with all memory-reference instructions. A saving in the number of control memory words may be achieved if the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by INDRCT, is located right after the fetch routine, as shown in Table 1-2. The table also shows the symbolic microprogram for the fetch routine and the microinstruction routines that execute four computer instructions.

To see how the transfer and return from the indirect subroutine occurs, assume that the MAP microinstruction at the end of the fetch routine caused a branch to address 0, where the ADD routine is stored. The first microinstruction in the ADD routine calls subroutine INDRCT, conditioned on status bit *I.* If I = 1, a branch to INDRCT occurs and the return address (address 1 in this case) is stored in the subroutine register *SBR.* The INDRCT subroutine has two microinstructions:

```
INDRCT:    READ    U  JMP  NEXT
           DRTAR   U  RET
```



Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

TABLE 1-2 Partial Symbolic Microprogram

Remember that an indirect address considers the address part of the instruction as the address where the effective address is stored rather than the address of the operand. Therefore, the memory has to be accessed to get the effective address, which is then transferred to *AR.* The return from subroutine (RET) transfers the address from *SBR* to *CAR,* thus returning to the| second microinstruction of the ADD routine.

The execution of the ADD instruction is carried out by the microinstructions at addresses 1 and 2. The first microinstruction reads the operand from memory into *DR.* The second microinstruction performs an add microoperation with the content of *DR* and *AC* and then jumps back to the beginning of the fetch routine.

The BRANCH instruction should cause a branch to the effective address if *AC* < 0. The *AC* will be less than zero if its sign is negative, which is detected from status bit 5 being a 1. The BRANCH routine in Table 1-2 starts by checking the value of *S.* If *S is* equal to 0, no branch occurs and the next microinstruction causes a jump back to the fetch routine without altering the content *of PC.* If S is equal to 1, the first JMP microinstruction transfers control to location OVER. The microinstruction at this location calls the INDRCT subroutine if I = 1. The effective address is then transferred from *AR* to .PC and the microprogram jumps back to the fetch routine.

The STORE routine again uses the INDRCT subroutine if I= 1. The content *of AC is* transferred into *DR.* A memory write operation is initiated to store the content of *DR* in a location specified by the effective address in *AR.*

The EXCHANGE routine reads the operand from the effective address and places it in *DR.* The contents of *DR* and *AC* are interchanged in the third microinstruction. This interchange is possible when the registers are of the edge-triggered type. The original content of *AC* that is now in *DR is* stored back in memory.

Note that Table 1-2 contains a partial list of the microprogram. Only four out of 16 possible computer instructions have been microprogrammed. Also, control memory words at locations 69 to 127 have not been used. Instructions such as multiply, divide, and others that require a long sequence of microoperations will need more than four microinstructions for their execution. Control memory words 69 to 127 can be used for this purpose.

1-3-5 Binary Microprogram

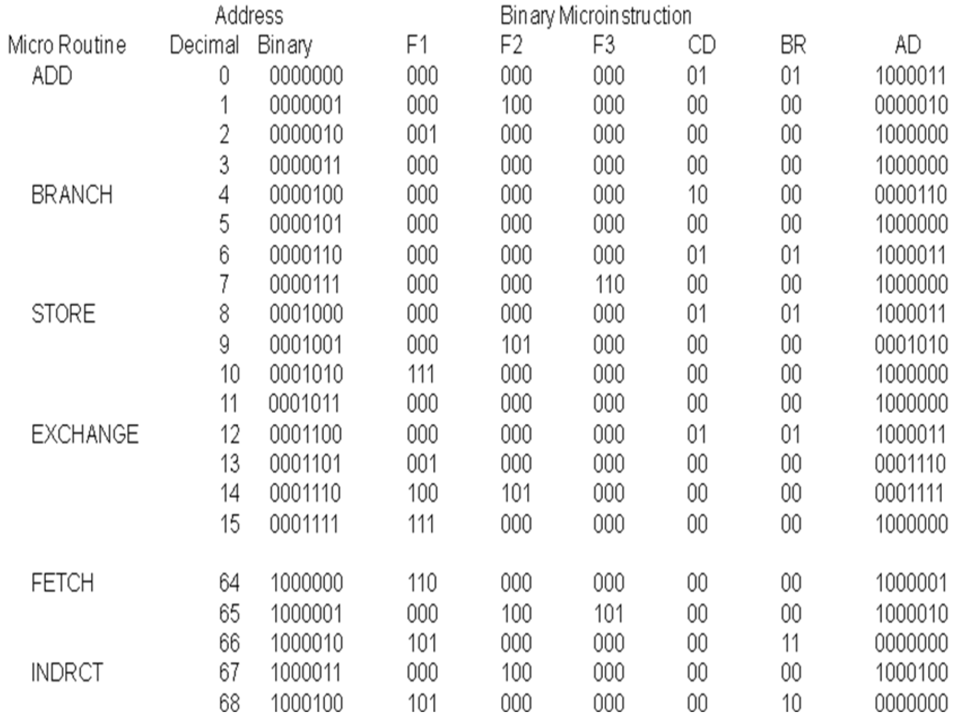
The symbolic microprogram is a convenient form for writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in memory. The symbolic microprogram must be translated to binary either by means of an assembler program or by the user if the microprogram is simple enough as in this example.

NOTES

The equivalent binary form of the microprogram is listed in Table 1-3. The addresses for control memory are given in both decimal and binary. The binary-content of each microinstruction is derived from the symbols and their equivalent binary values as defined in Table 1-1.

Note that address 3 has no equivalent in the symbolic microprogram since the ADD routine has only three microinstructions at addresses 0, 1, and 2. The next routine starts at address 4. Even though address 3 is not used, some binary value must be specified for each word in control memory. We could have specified all 0's in the word since this location will never be used. However, if some unforeseen error occurs, or if a noise signal sets *CAR* to the value of 3, it will be wise to jump to address 64, which is the beginning of the fetch routine.

The binary microprogram listed in Table 1-3 specifies the word content of the control memory. When a ROM is used for the control memory, the microprogram binary list provides the truth table for fabricating the unit. This fabrication is a hardware process and consists of creating a mask for the ROM so as to produce the 1's and 0's for each word. The bits of ROM are fixed once the internal links are fused during the hardware production. The ROM is made of 1C packages that can be removed if necessary and replaced by other packages. To modify the instruction set of the computer, it is necessary to generate a new microprogram and mask a new ROM. The old one can be removed and the new one inserted in its place.



	Address		Binary Microinstruction					
Micro Routine	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

TABLE 1-3 Binary Microprogram for Control Memory (partial)

If a writable control memory is employed, the ROM is replaced by a RAM. The advantage of employing a RAM for the control memory is that the| microprogram can be altered simply by writing a new pattern of 1's and 0's without resorting to hardware procedures. A writable control memory possesses the flexibility of choosing the instruction set of a computer dynamically by changing the microprogram under processor control. However, most microprogrammed systems use a ROM for the control memory because it is cheaper and faster than a RAM and also to prevent the occasional user from changing the architecture of the system.

1-4 Summary

In this lesson, we have discussed about the micro-programmed control unit. The key to such a unit is a microinstruction. A microinstruction has been defined in the unit. In addition we have also explained a basic microprogrammed control unit. The two basic functions of microprogrammed control: macroinstruction sequencing and macroinstruction execution has also discussed in this lesson.

1-5 Keywords

Control word: The control variables at any given time can be represented by a string of 1's and 0's called a control word.

Microinstruction: The microinstruction specifies one or more microoperations for the system.

Microprogram: A microprogram is a small program that is usually put onto a computer chip. It has instructions on how to do simple things.

Control memory: A memory that is part of a control unit is referred to as a control memory

Sequencer: The next address generator is called sequencer.

Pipeline: An instruction pipeline is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

Mapping: The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a *mapping* process. A mapping procedure is a rule that transforms the instruction code into a control memory address.

Microoperations: The operations on the data in registers are called microoperations.

NOTES

1-6 Exercise

1. What is the difference between a microprocessor and a microprogram? Is it possible to design a microprocessor without a microprogram? Are all microprogrammed computers also microprocessors?

2. Explain the difference between hardwired control and microprogrammed control. Is it possible to have a hardwired control associated with a control memory?

3. Define the following: (a) microoperation; (b) microinstruction; (c) microprogram; (d) microcode.

4. The microprogrammed control organization shown in Fig. 1-1 has the following propagation delay times. 40ns to generate the next address, 10 ns to transfer the address into the control address register, 40ns to access the control memory ROM, 10 ns to transfer the microinstruction into the control data register, and 40ns to perform the required microoperations specified by the control word. What is the maximum clock frequency that the control can use? What would the clock frequency be if the control data register is not used?

5. The system shown in Fig. 1-2 uses a control memory of 1024 words of 32 bits each. The microinstruction has three fields as shown in the diagram. The micro-operations field has 16 bits.

a. How many bits are there in the branch address field and the select field?

b. If there are 16 status bits in the system, how many bits of the branch logic are used to select a status bit?

c. How many bits are left to select an input for the multiplexers?

6. The control memory in Fig. 2-2 has 4096 words of 24 bits each.

a. How many bits are there in the control address register?

b. How many bits are there in each of the four inputs, shown going into the multiplexers?

1-7 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings —Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

2. DESIGN OF CONTROL UNIT

Structure

Objective

At the end of the lesson you will be able to:

- Describe about microprogram sequencer
- Discuss basics of the microarchitecture
- Discuss about hardwired control

2-1 Introduction

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate microoperations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate microoperations can be reduced by grouping mutually exclusive variables into fields and encoding the *k-bits* in each field to provide 2* microoperations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Figure 2-1 shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 X 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 1-1in lesson1. For example, when *F1* = 101 (binary 5), the next clock pulse transition transfers the content of *DR (0-10) to AR* (symbolized by DRTAR in Table 1-1). Similarly, when *F1* = 110 (binary 6) there is a transfer from *PC to AR* (symbolized by PCTAR). As shown in Fig. 2-1, outputs 5 and 6 of decoder *Ft* are connected to the load input of *AR so that* when either one of these outputs is active, information from the multiplexers is transferred to *AR.* The multiplexers select the information from *DR* when output 5 is active and from *PC* when output 5 is inactive. The transfer into *AR* occurs with a clock pulse transition only when output 5 or output 6 of the decoder is active. The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

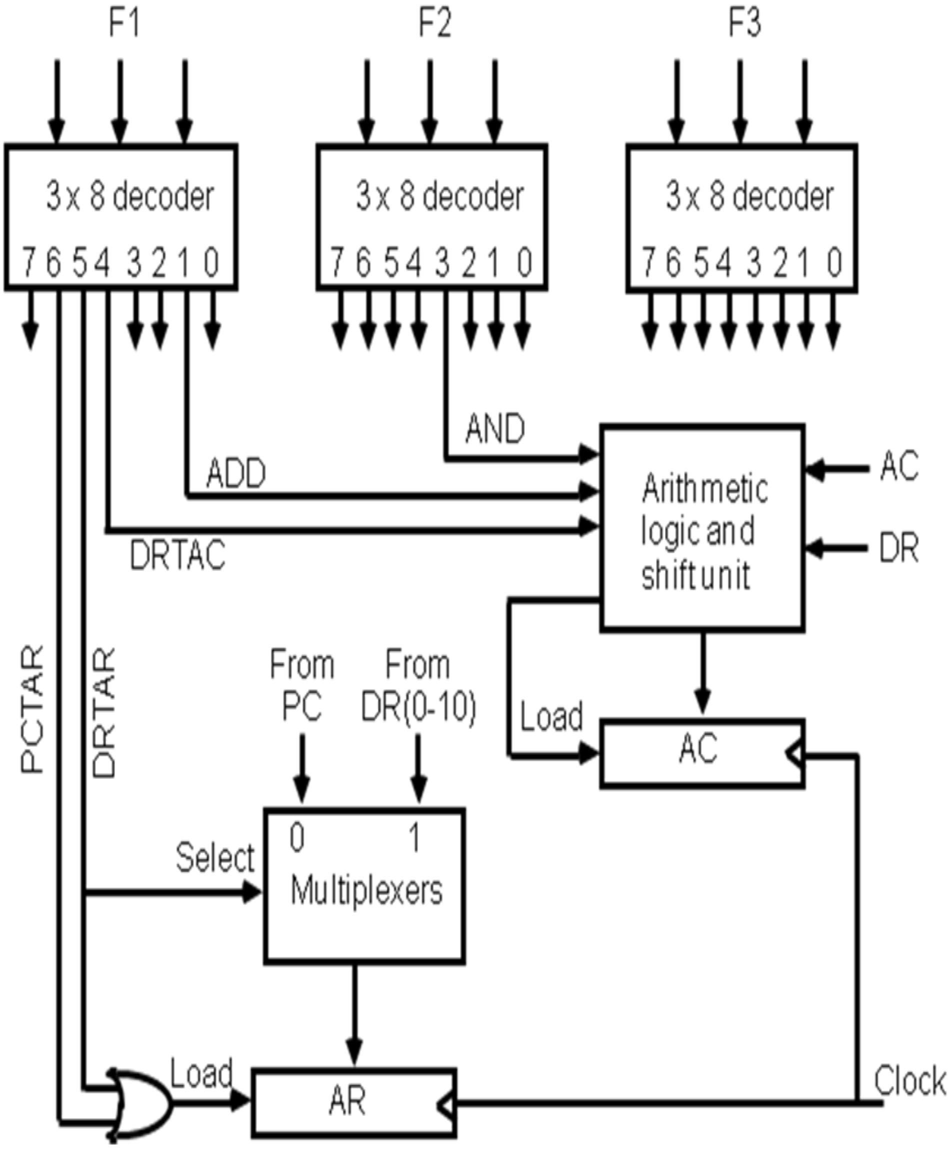


Figure 2-1 Decoding of microoperation fields

2-2 Microprogram Sequencer

The basic components of a microprogrammed control unit are the control memory and the circuits that select the next address. The address selection part is called a microprogram sequencer. A microprogram sequencer can be constructed with digital functions to suit a particular application. However, just as there are large ROM units available in integrated circuit packages, so are general-purpose sequencers suited for the construction of microprogram control units. To guarantee a wide range of acceptability, an integrated circuit sequencer must provide an internal organization that can be adapted to a wide range of applications.

The purpose of a microprogram sequencer is to present an address to the control memory so that a microinstruction may be read and executed. The next-address logic of the sequencer determines the specific address source to be loaded into the control address register. The choice of the address source is guided by the next-address information bits that the sequencer receives from the present microinstruction. Commercial sequencers include within the unit an internal register stack used for temporary storage of addresses during microprogram looping and subroutine calls. Some sequencers provide an output register which can function as the address register for the control memory.

To illustrate the internal structure of a typical microprogram sequencer we will show a particular unit that is suitable for use in the microprogram computer example developed in the preceding section. The block diagram of the microprogram sequencer is shown in Fig. 2-2. The control memory is included in the diagram to show the interaction between the sequencer and the memory attached to it. There are two multiplexers in the circuit. The first multiplexer selects an address from one of four sources and routes it into a control address register *CAR.* The second multiplexer tests the value of a selected status bit and the result of the test is applied to an input logic circuit. The output from *CAR* provides the address for the control memory. The content of *CAR* is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR.* The other three inputs to multiplexer number 1 come from the address field of the present microinstruction, from the output *of SBR,* and from an external source that maps the instruction. Although the diagram shows a single subroutine register, a typical sequencer will have a register stack about four to eight levels deep. In this way, a number of subroutines can be active at the same time. A push and pop operation, in conjunction with a stack pointer, stores and retrieves the return address during the call and return microinstructions.

The CD (condition) field of the microinstruction selects one of the status bits in the second multiplexer. If the bit selected is equal to 1, the *T* (test) variable is equal to 1; otherwise, it is equal to 0. The *T* value together with the two bits from the BR (branch) field goes to an input logic circuit. The input logic in a particular sequencer will determine the type of operations that are available in the unit. Typical sequencer operations are: increment, branch or jump, call and return from subroutine, load an

NOTES

external address, push or pop the stack, and other address sequencing operations. With three inputs, the sequencer can provide up to eight address sequencing operation. Some commercial sequencers have three or four inputs in addition to the *T* input and thus provide a wider range of operations.

The input logic circuit in Fig. 2-2 has three inputs, I_0,I_1, and *T,* and three outputs. S_o, S_1, and *L.* Variables S_o and S_1 select one of the source addresses for *CAR.* Variable *L* enables the load input in *SBR.* The binary values of the two selection variables determine the path in the multiplexer. For example, with S_1,S_o = 10, multiplexer input number 2 is selected and establishes a transfer path from *SBR to CAR.* Note that each of the four inputs as well as the output of MUX 1 contains a 7-bit address.

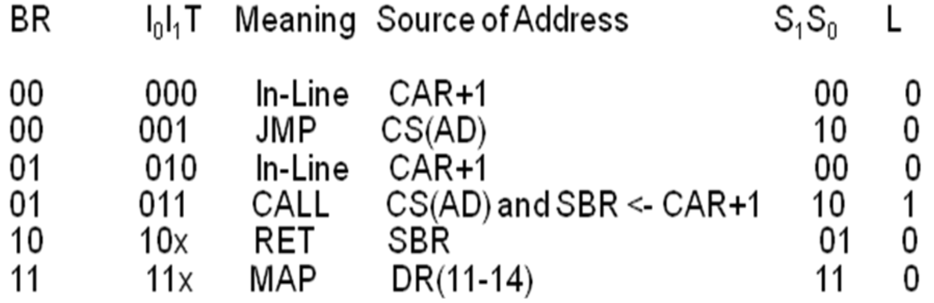
The truth table for the input logic circuit is shown in Table 2-1. Inputs I_0 and I_1 are identical to the bit values in the BR field. The function listed in each entry was defined in Table 1-1 in lesson 1. The bit values for S_1 and S_o are determined from the stated function and the path in the multiplexer that establishes the required transfer. The subroutine register is loaded with the incremented value of *CAR* during a call microinstruction (BR = 01) provided that the status bit condition is satisfied (T= 1). The truth table can be used to obtain the simplification fled Boolean functions for the input logic circuit:

$$S_0 = I_0$$

$$S_1 = I_0I_1 + I_0'T$$

$$L = I_0'I_1T$$

The circuit can be constructed with three AND gates, an OR gate, and an inverter.



BR	I_0I_1T	Meaning	Source of Address	S_1S_0	L
00	000	In-Line	CAR+1	00	0
00	001	JMP	CS(AD)	10	0
01	010	In-Line	CAR+1	00	0
01	011	CALL	CS(AD) and SBR <- CAR+1	10	1
10	10x	RET	SBR	01	0
11	11x	MAP	DR(11-14)	11	0

TABLE 2-1 Input Logic Truth Table for Microprogram Sequencer

Note that the incrementer circuit in the sequencer of Fig. 2-2 is not a counter constructed with flip-flops but rather a combinational circuit constructed with gates. A combinational circuit incrementer can be designed by cascading a series of half-adder circuits. The output carry from one stage must be applied to the input of the next stage. One input in the first least significant stage must be equal to 1 to provide the increment-by-one operation.

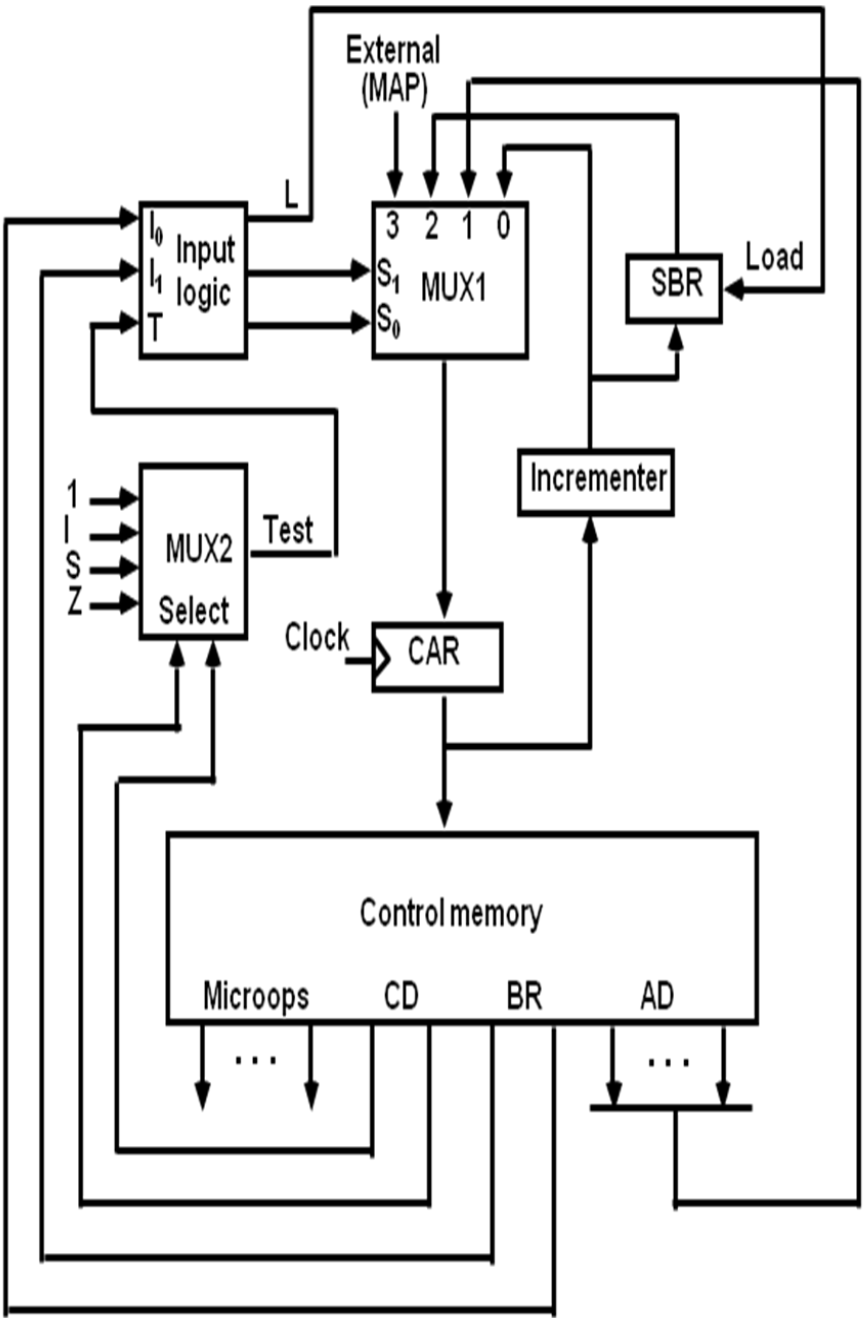


Figure 2-2 Microprogram sequence for a control memory

2-4 Basics of the Microarchitecture

The functionality of the microarchitecture centers around the fetch-execute cycle, which is in some sense the “heart” of the machine, the steps involved in the fetch-execute cycle are:

1) Fetch the next instruction to be executed from memory.

2) Decode the opcode.

3) Read operand(s) from main memory or registers, if any.

4) Execute the instruction and store results.

5) Go to Step 1.

It is the microarchitecture that is responsible for making these five steps. The microarchitecture fetches the next instruction to be executed, determines which instruction it is, fetches the operands, and executes the instruction, stores the results, and then repeats. The microarchitecture consists of a data section which contains registers and an ALU, and a control section, as illustrated in Figure 2-3. The data section is also referred to as the datapath. Microprogrammed control uses a special purpose microprogram, not visible to the user, to implement operations on the registers and on other parts of the machine. Often, the microprogram contains many program steps that collectively implement a single (macro) instruction. Hardwired control units adopt the view that the steps to be taken to implement an operation comprise states in a finite state machine, and the design proceeds using conventional digital design methods In either case, the datapath remains largely unchanged, although there may be minor differences to support the differing forms of control. In designing the ARC control unit, the microprogrammed approach will be explored first, and then the hardwired approach, and for both cases the datapath will remain unchanged.

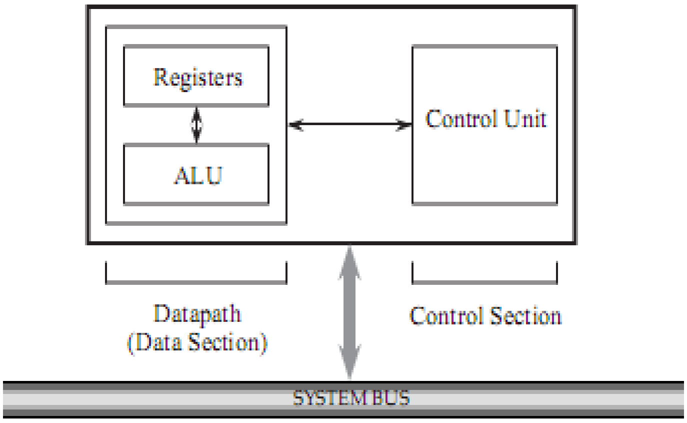


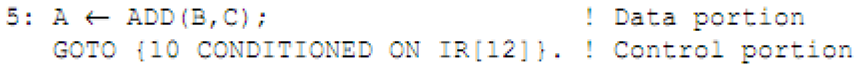
Figure 2-3 High level view of microarchitecture.

2-5 Hardwired Control

An alternative approach to a microprogrammed control unit is to use a hardwired approach, in which a direct implementation is created using flip-flop and logic gates, instead of using a control store and a micro-word selection mechanism. States in a finite state machine replace steps in the microprogram.

In order to manage the complexity of design for a hardwired approach, a hardware description language (HDL) is frequently used to represent the control structure. One example of an HDL is VHDL, which is an acronym for VHSIC Hardware Description Language (in which VHSIC is yet another acronym for Very High Speed Integrated Circuit). VHDL is used for describing architecture at a very high level, and can be compiled into hardware designs through a process known as silicon compilation. For the hardwired control unit we will design here, a lower level HDL that is sometimes referred to as a register transfer language (RTL) is more appropriate.

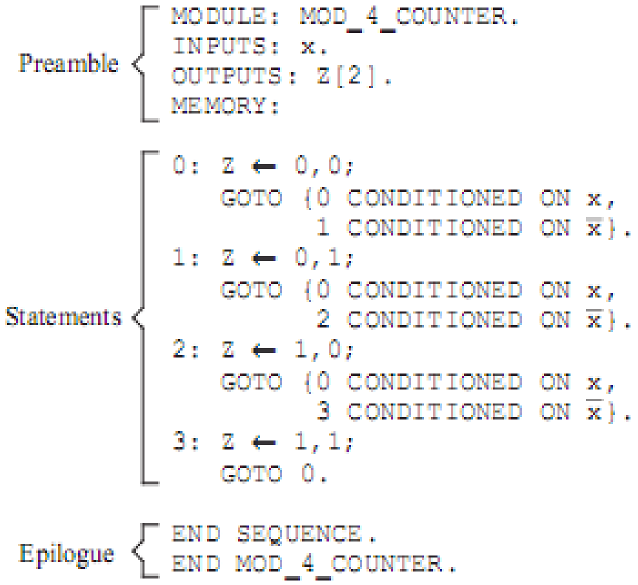
We will defined a simple HDL/RTL in this section that loosely resembles Hill & Peterson's A Hardware Programming Language (AHPL) (Hill and Peterson, 1987). The general idea is to express a control sequence as a series of numbered statements, which can then be directly translated into a hardware design. Each statement consists of a data portion and a transfer portion, as shown below:



```
5: A ← ADD(B,C);                        ! Data portion
   GOTO {10 CONDITIONED ON IR[12]}.     ! Control portion
```

The statement is labeled "5," which means that it is preceded by statement 4 and is succeeded by statement 6, unless an out-of-sequence transfer of control takes place. The left arrow (←) indicates a data transfer, to register A for this case. The "ADD (B, C)" construct indicates that registers B and C are sent to a combinational logic unit (CLU) that performs the addition. Comments begin with an exclamation mark (!) and terminate at the end of the line. The GOTO construct indicates a transfer of control. For this case, control is transferred to statement 10 if bit 12 of register IR is true, otherwise control is transferred to the next higher numbered statement (6 for this case).

Figure 2-4 shows an HDL description of modulo 4 counters. The counter produces the output sequence: 00, 01, 10, 11 and then repeats as long as the input line x is 0. If the input line is set to 1, then the counter returns to state 0 at the end of the next clock cycle. The comma is the catenation operator, and so the statement "Z ← 0,0;" assigns the two-bit pattern 00 to the two-bit output Z.


Preamble
Statements
Epilogue

```
MODULE: MOD_4_COUNTER.
INPUTS: x.
OUTPUTS: Z[2].
MEMORY:

0: Z ← 0,0;
   GOTO {0 CONDITIONED ON x,
         1 CONDITIONED ON x̄}.
1: Z ← 0,1;
   GOTO {0 CONDITIONED ON x,
         2 CONDITIONED ON x̄}.
2: Z ← 1,0;
   GOTO {0 CONDITIONED ON x,
         3 CONDITIONED ON x̄}.
3: Z ← 1,1;
   GOTO 0.

END SEQUENCE.
END MOD_4_COUNTER.
```

Figure 2-4 HDL sequence for a resettable modulo 4 counters.

The HDL sequence is composed of three sections: the preamble, the numbered statements, and the epilogue. The preamble names the module with the “MODULE” keyword and declares the inputs with the “INPUTS” keyword, the outputs with the “OUTPUTS” keyword, and the parity (number of signals) of both, as well as any additional storage with the “MEMORY” keyword (none for this example). The numbered statements follow the preamble. The epilogue closes the sequence with the key phrase “END SEQUENCE.” The key phrase “END MOD_4_COUNTER” closes the description of the module. Anything that appears between “END SEQUENCE” and “END MOD_4_COUNTER” occurs continuously, independent of the statement number. There are no such statements for this case.

In translating an HDL description into a design, the process can be decomposed into separate parts for the control section and the data section. The control section deals with how transitions are made from one statement to another. The data section deals with producing outputs and changing the values of any memory elements.

We consider the control section first. There are four numbered statements, and so we will use four flip-flops, one for each statement, as illustrated in Figure 2-5. This is referred to as a one-hot encoding approach, because exactly one flip-flop holds a true value at any time. Although four states can be encoded using only two flip-flops, studies have shown that the one-hot encoding approach results in approximately the same circuit area when compared with a more densely encoded approach; but more importantly, the complexity of the transfers from one state to the next are generally simpler and can be implemented with shallow combinational logic circuits, which means that the clock rate can be faster for a one-hot encoding approach than for a densely encoded approach.

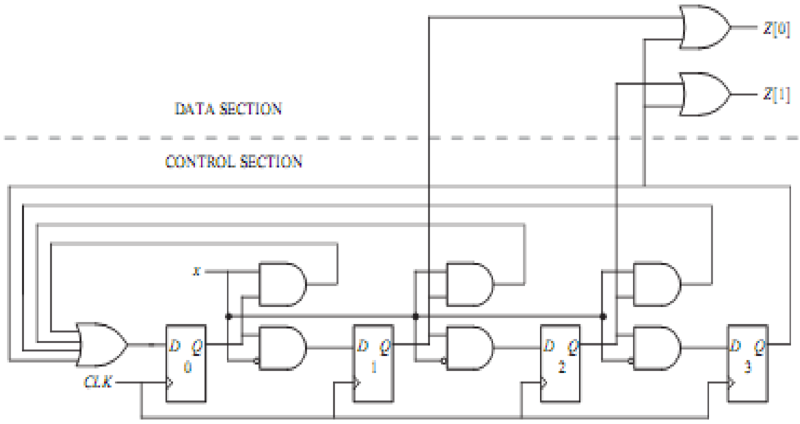


Figure 2-5 Logic design for a modulo 4 counter described in HDL.

In designing the control section, we first draw the flip-flops, apply labels as appropriate, and connect the clock inputs. The next step is to simply scan the numbered statements in order and add logic as appropriate for the transitions. From statement 0, there are two possible transitions to statements 0 or 1, conditioned on x or its complement, respectively. The output of flip-flop 0 is thus connected to the inputs of flip-flops 0 and 1, through AND gates that take the value of the x input into account. Note that the AND gate into flip-flop 1 has a circle at one of its inputs, which is a simplified notation that means x is complemented by an inverter before entering the AND gate.

A similar arrangement of logic gates is applied for statements 1 and 2, and no logic is needed at the output of flip-flop 3 because statement 3 returns to statement 1 unconditionally. The control section is now complete and can execute correctly on its own. No outputs are produced, however, until the data section is implemented.

We now consider the design of the data section, which is trivial for this case. Both bits of the output Z change in every statement, and so there is no need to condition the generation of an output on the state. We only need to produce the correct output values for each of the statements. The least significant bit of Z is true in statements 1 and 3, and so the outputs of the corresponding control flip-flops are ORed to produce Z[0]. the most significant bit of Z is true in statements 2 and 3, and so the outputs of the corresponding control flip-flops are ORed to produce Z[1]. The entire circuit for the mod 4 counter is now complete, as shown in Figure 2-5.

2-6 Summary

In this lesson, we have discussed about the micro-programmed sequences. The key to such a unit is a microinstruction. A microinstruction has been defined in the unit. In addition we have also explained a basic hardware control unit.

2-7 Keywords

Decoding: Decoding is the reverse of encoding, which is the process of transforming information from one format into another.

HDL: Hardware Description Language.

VHDL: Verilog Hardware Description Language.

VHSIC: Very High Speed Integrated Circuit.

RTL: Register Transfer Language

2-8 Exercise

1. Insert an exclusive-OR gate between MUX 2 and the input logic of Fig. *2-2* one input to the gate comes from the test output of the multiplexer. The other input to the gate comes from a bit labeled p (for polarity) in the microinstruction from control memory. The output of the gate goes to the input *T* of the input logic. What does the polarity control *P* accomplish?

2. Design a 7-bit combinational circuit incrementer for the microprogram sequencer of Fig. 2-2. Modify the incrementer by including a control input *D.* When *D* = 0, the circuit increments by one, but when *D* = 1, the circuit increments by two.

3. A computer has 16 registers, an ALU (arithmetic logic unit) with 32 operations, and a shifter with eight operations, all connected to a common bus system.

a. Formulate a control word for a microoperation.

b. Specify the number of bits in each field of the control word and give a general encoding scheme.

c**.** Show the bits of the control word that specify the microoperation

$$R4 \leftarrow R5 + R6.$$

2-9 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings —Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes.

3. CENTRAL PROCESSING UNIT

Structure

Objectives

At the end of lesson you will be able to:

- Discuss about CPU basics
- Discuss different register sets
- Describe datapath
- Define instruction cycle.

3-1 Introduction

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special- purpose registers. General-purpose registers are used for any purpose, hence the name general purpose. Special-purpose registers have specific functions within the CPU. For example, the program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next. Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed. The ALU provides the circuitry needed to perform the arithmetic, logical and shift operations demanded of the instruction set. The control unit is the entity responsible for fetching the instruction to be executed from the main memory and decoding and then executing it. Figure 3.1 shows the main components of the CPU and its interactions with the memory system and the input/output devices.

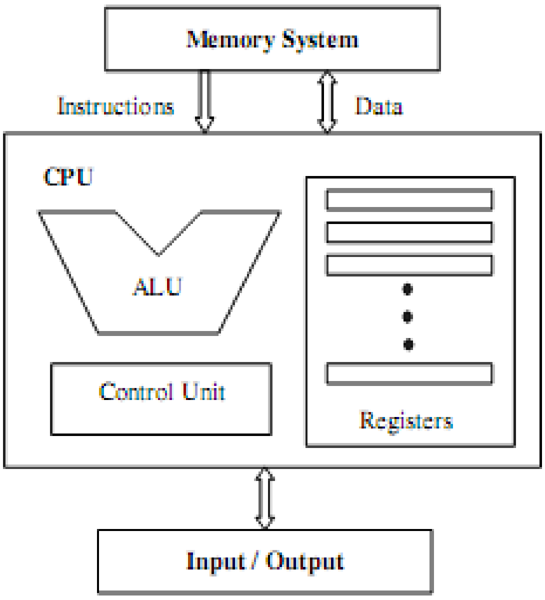


Figure 3-1 Central processing unit main components and interactions with the memory and I/O

The CPU fetches instructions from memory, reads and writes data from and to memory, and transfers data from and to input/output devices. A typical and simple execution cycle can be summarized as follows:

1. The next instruction to be executed, whose address is obtained from the PC, is fetched from the memory and stored in the IR.
2. The instruction is decoded.
3. Operands are fetched from the memory and stored in CPU registers, if needed.
4. The instruction is executed.
5. Results are transferred from CPU registers to the memory, if needed.

The execution cycle is repeated as long as there are more instructions to execute. A check for pending interrupts is usually included in the cycle. Examples of interrupts include I/O device request, arithmetic overflow, or a page fault. When an interrupt request is encountered, a transfer to an interrupt handling routine takes place. Interrupt handlings routines are programs that are invoked to collect the state of the currently executing program, correct the cause of the interrupt, and restore the state of the program.

The actions of the CPU during an execution cycle are defined by micro-orders issued by the control unit. These micro-orders are individual control signals sent over dedicated control lines. For example, let us assume that we want to execute an instruction that moves the contents of register X to register Y. Let us also assume that both registers are connected to the data bus, D. The control unit will issue a control signal to tell register X to place its contents on the data bus D. After some delay, another control signal will be sent to tell register Y to read from data bus D. The activation of the control signals is determined using either hardwired control or microprogramming. These concepts are explained later in this chapter.

3 -2 Register set

Registers are essentially extremely fast memory locations within the CPU that are used to create and store the results of CPU operations and other calculations. Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register. General-purpose registers can be used for multiple purposes and assigned to a variety of functions by the programmer. Special-purpose registers are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses. The length of a data register must be long enough to hold values of most data types. Some machines allow two contiguous registers to hold double-length values. Address registers may be dedicated to a particular addressing mode or may be used as address general purpose. Address registers must be long enough to hold the largest address. The number of registers in a particular architecture affects the instruction set design. A very small number of registers may result in an increase in memory references. Another type of registers is used to hold processor status bits, or flags. These bits are set by the CPU as the result of the execution of an operation. The status bits can be tested at a later time as part of another operation.

3-2-1 Memory Access Registers

Two registers are essential in memory write and read operations: the memory data register (MDR) and memory address register (MAR). The MDR and MAR are used exclusively by the CPU and are not directly accessible to programmers.

In order to perform a write operation into a specified memory location, the MDR and MAR are used as follows:

1. The word to be stored into the memory location is first loaded by the CPU into MDR.
2. The address of the location into which the word is to be stored is loaded by the CPU into a MAR.
3. A write signal is issued by the CPU.

Similarly, to perform a memory read operation, the MDR and MAR are used as follows:

1. The address of the location from which the word is to be read is loaded into the MAR.
2. A read signal is issued by the PU.
3. The required word will be loaded by the memory into the MDR ready for use by the CPU.

3-2-2 Instruction Fetching Registers

Two main registers are involved in fetching an instruction for execution: the program counter (PC) and the instruction register (IR). The PC is the register that contains the address of the next instruction to be fetched. The fetched instruction is loaded in the IR for execution. After a successful instruction fetch, the PC is updated to point to the next instruction to be executed. In the case of a branch operation, the PC is updated to point to the branch target instruction after the branch is resolved, that is, the target address is known.

3-2-3 Condition Registers

Condition registers, or flags, are used to maintain status information. Some architecture contains a special program status word (PSW) register. The PSW contains bits that are set by the CPU to indicate the current status of an executing program. These indicators are typically for arithmetic operations, interrupts, memory protection information, or processor status.

3-2-4 Special-Purpose Address Registers

The address of the operand is obtained by adding a constant to the content of a register, called the index register. The index register holds an address displacement. Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol X to indicate the constant to be added.

Segment Pointers In order to support segmentation, the address issued by the processor should consist of a segment number (base) and a displacement (or an offset) within the segment. A segment register holds the address of the base of the segment.

Stack Pointer A stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the Push and the Pop operations. A specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed. In the stack push operation, the SP value is used to indicate the location (called the top of the stack). After storing (pushing) this value, the SP is incremented (in some architectures, e.g. X86, the SP is decremented as the stack grows low in memory).

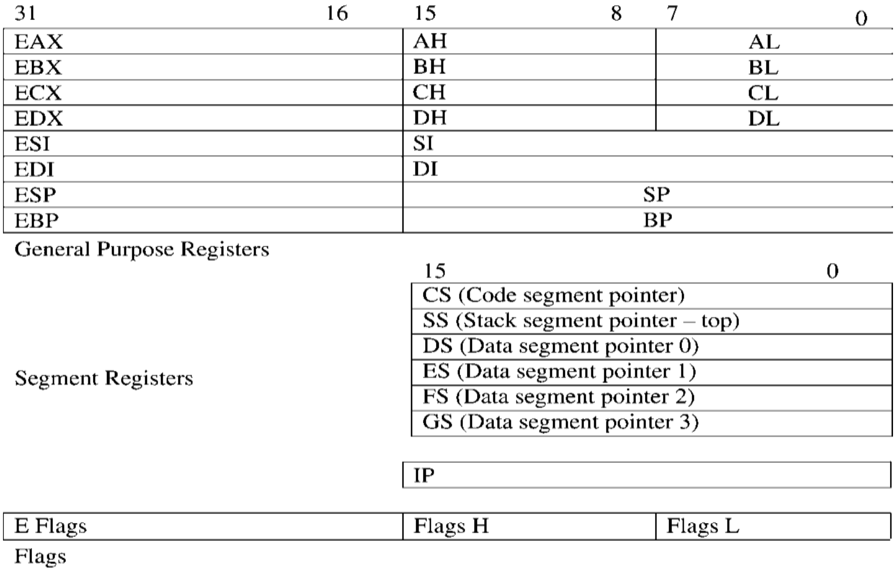
3-2-5 80386 Registers

The Intel basic programming model of the 386, 486, and the Pentium consists of three register groups. These are the general-purpose registers, the segment registers, and the instruction pointer (program counter) and the flag register.

Figure 3.2 shows the three sets of registers. The first set consists of general purpose registers A, B, C, D, SI (source index), DI (destination index), SP (stack pointer), and BP (base pointer). The second set of registers consists of CS (code segment), SS (stack segment), and four data segment registers DS, ES, FS, and GS. The third set of registers consists of the instruction pointer (program counter) and the flags (status) register. Among the status bits, the first five are identical to those bits introduced as early as in the 8085 8-bit microprocessor. The next 6 – 11 bits are identical to those introduced in the 8086. The flags in the bits 12 – 14 were introduced in the 80286 while the 16 – 17 bits were introduced in the 80386. The flag in bit 18 was introduced in the 80486.

3-2-6 MIPS Registers

The MIPS CPU contains 32 general-purpose registers that are numbered 0 – 31. Register x is designated by $x. Register $zero always contains the hardwired value 0. Table 3.1 lists the registers and describes their intended use. Registers $at (1), $k0 (26), and $k1 (27) are reserved for use by the assembler and operating system. Registers $a0 – $a3 (4 – 7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack). Registers $v0 and $v1 (2, 3) are used to return values from functions. Registers $t0 – $t9 (8 – 15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers $s0 – $s7 (16 – 23) are caller saved registers that hold long-lived values that should be preserved across calls.



31 16	15 8	7 0
EAX	AH	AL
EBX	BH	BL
ECX	CH	CL
EDX	DH	DL
ESI	SI	
EDI	DI	
ESP	SP	
EBP	BP	

General Purpose Registers

Segment Registers

15 0
CS (Code segment pointer)
SS (Stack segment pointer – top)
DS (Data segment pointer 0)
ES (Data segment pointer 1)
FS (Data segment pointer 2)
GS (Data segment pointer 3)

IP

E Flags	Flags H	Flags L

Flags

Figure 3-2 the main register sets in 80 86 (80386 and above extended all 16 bit registers Except segment registers)

TABLE 3.1 MIPS General-Purpose Registers

Name	Number	Usage	Name	Number	Usage
zero	0	Constant 0	s0	16	Saved temporary (preserved across call)
at	1	Reserved for assembler	s1	17	Saved temporary (preserved across call)
v0	2	Expression evaluation and	s2	18	Saved temporary (preserved across call)
v1	3	results of a function	s3	19	Saved temporary (preserved across call)
a0	4	Argument 1	s4	20	Saved temporary (preserved across call)
a1	5	Argument 2	s5	21	Saved temporary (preserved across call)
a2	6	Argument 3	s6	22	Saved temporary (preserved across call)
a3	7	Argument 4	s7	23	Saved temporary (preserved across call)
t0	8	Temporary (not preserved across call)	t8	24	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)	t9	25	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)	k0	26	Reserved for OS kernel
t3	11	Temporary (not preserved across call)	k1	27	Reserved for OS kernel
t4	12	Temporary (not preserved across call)	gp	28	Pointer to global area
t5	13	Temporary (not preserved across call)	sp	29	Stack pointer
t6	14	Temporary (not preserved across call)	fp	30	Frame pointer
t7	15	Temporary (not preserved across call)	ra	31	Return address (used by function call)

3-3 Datapath

The CPU can be divided into a data section and a control section. The data section, which is also called the datapath, contains the registers and the ALU. The datapath is capable of performing certain operations on data items. The control section is basically the control unit, which issues control signals to the datapath. Internal to the CPU, data move from one register to another and between ALU and registers. Internal data movements are performed via local buses, which may carry data, instructions, and addresses. Externally, data move from registers to memory and I/O devices, often by means of a system bus. Internal data movement among registers and between the ALU and registers may be carried out using different organizations including one-bus, two-bus, or three-bus organizations. Dedicated datapaths may also be used between components that transfer data between them- selves more frequently. For example, the contents of the PC are transferred to the MAR to fetch a new instruction at the beginning of each instruction cycle. Hence, a dedicated datapath from the PC to the MAR could be useful in speeding up this part of instruction execution.

3-3-1 One-Bus Organization

Using one bus, the CPU registers and the ALU use a single bus to move outgoing and incoming data. Since a bus can handle only a single data movement within one clock cycle, two-operand operations will need two cycles to fetch the operands for the ALU. Additional registers may also be needed to buffer data for the ALU. This bus organization is the simplest and least expensive, but it limits the amount of data transfer that can be done in the same clock cycle, which will slow down the overall performance. Figure 3.3 shows a one-bus datapath consisting of a set of general-purpose registers, a memory address register (MAR), a memory data register (MDR), an instruction register (IR), a program counter (PC), and an ALU.

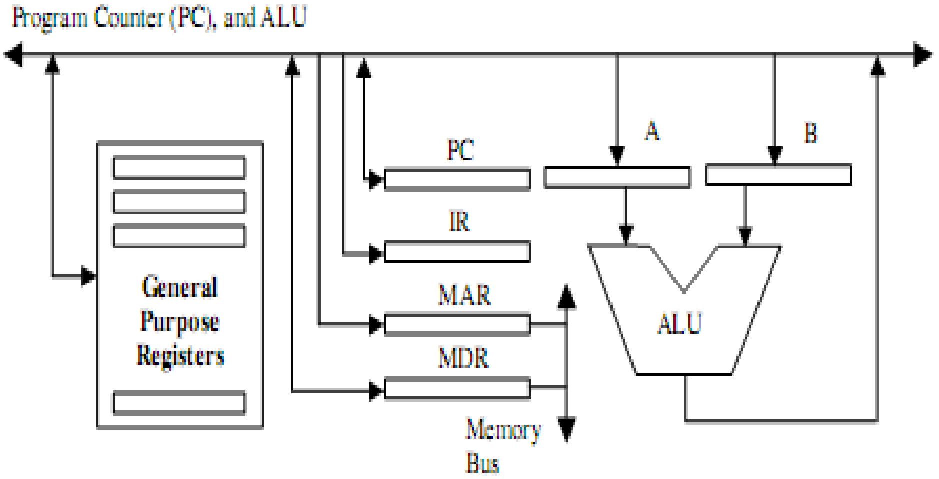


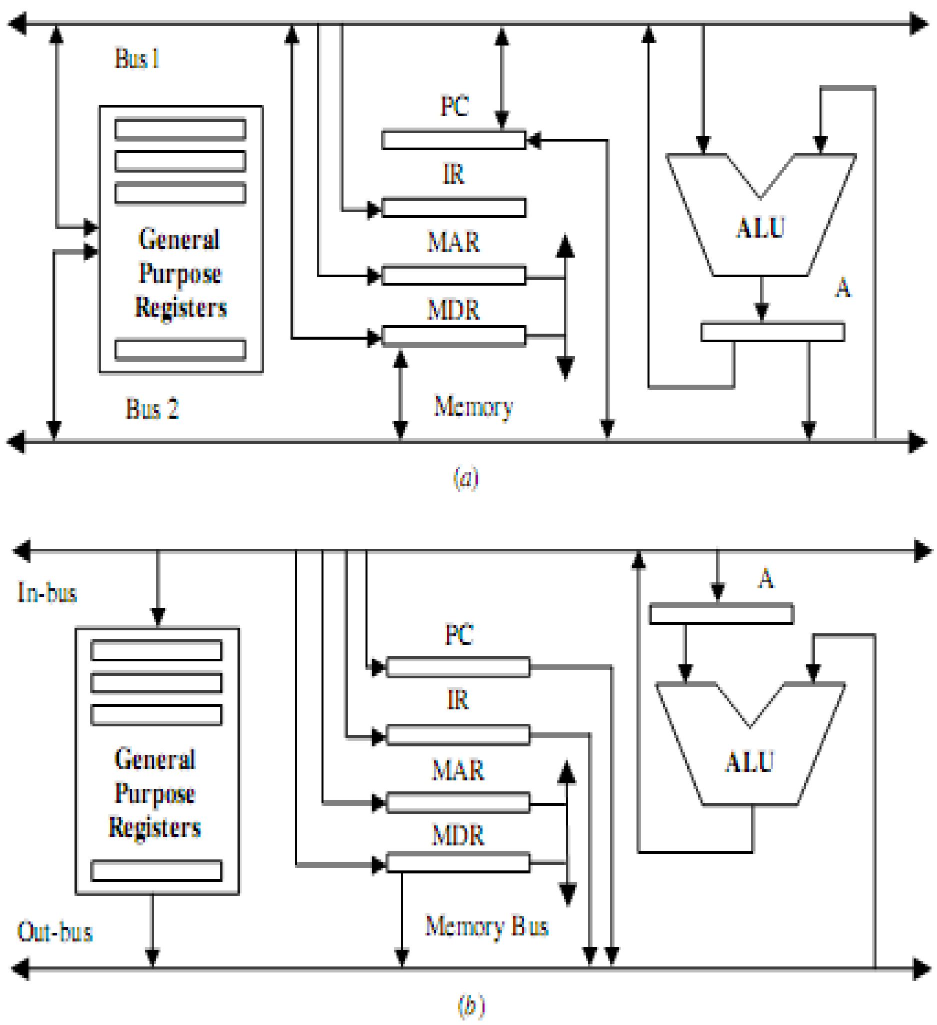
Figure 3.3 One-bus datapath

3-3-2 Two-Bus Organization

Using two buses is a faster solution than the one-bus organization. In this case, general purpose registers are connected to both buses. Data can be transferred from two different registers to the input point of the ALU at the same time. Therefore, a two- operand operation can fetch both operands in the same clock cycle. An additional buffer register may be needed to hold the output of the ALU when the two buses are busy carrying the two operands. Figure 3.4a shows a two-bus organization. In some cases, one of the buses may be dedicated for moving data into registers (in-bus), while the other is dedicated for transferring data out of the registers (out-bus). In this case, the additional buffer register may be used, as one of the ALU inputs, to hold one of the operands. The ALU output can be connected directly to the in-bus, which will transfer the result into one of the registers. Figure 3.4b shows a two-bus organization with in-bus and out-bus.

3-3-3 Three-Bus Organization

In a three-bus organization, two buses may be used as source buses while the third is used as destination. The source buses move data out of registers (out-bus), and the destination bus may move data into a register (in-bus). Each of the two out-buses is connected to an ALU input point. The output of the ALU is connected directly to the in-bus. As can be expected, the more buses we have, the more data we can move within a single clock cycle. However, increasing the number of buses will also increase the complexity of the hardware. Figure 3.5 shows an example of a three-bus datapath.



(a)

(b)

Figure 3.4 Two-bus organizations. (a) An Example of Two-Bus Datapath. (b) Another Example of Two-Bus Datapath with in-bus and out-bus

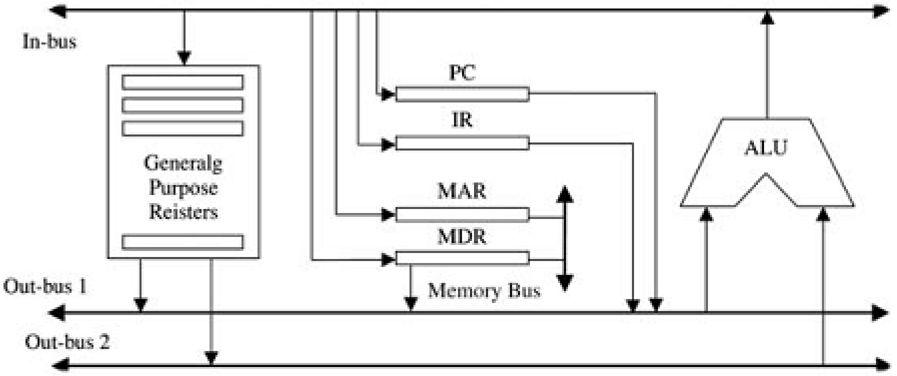


Figure 3.5 Three-bus datapath

3-4 CPU INSTRUCTION CYCLE

The sequence of operations performed by the CPU during its execution of instructions is presented in Fig. 3.6. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the opcode field of the instruction. At the completion of the instruction execution, a test is made to determine whether an interrupt has occurred. An interrupt handling routine needs to be invoked in case of an interrupt.

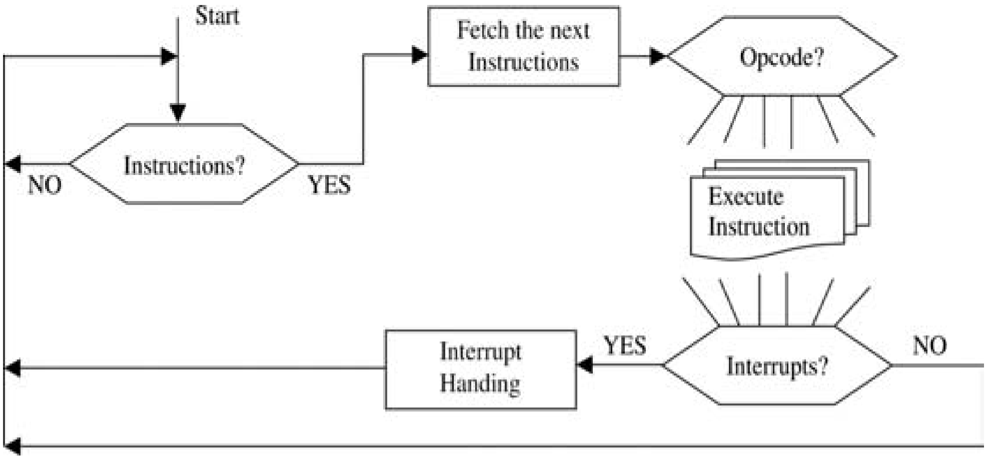


Figure 3.6 CPU Functions

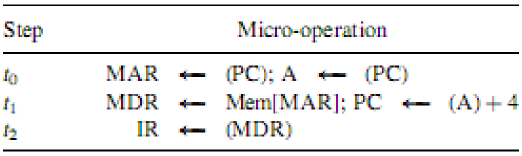
The basic actions during fetching an instruction, executing an instruction, or handling an interrupt are defined by a sequence of micro-operations. A group of control signals must be enabled in a prescribed sequence to trigger the execution of a micro- operation. In this section, we show the micro-operations that implement instruction fetch, execution of simple arithmetic instructions, and interrupt handling.

3-4-1 Fetch Instructions

The sequence of events in fetching an instruction can be summarized as follows:

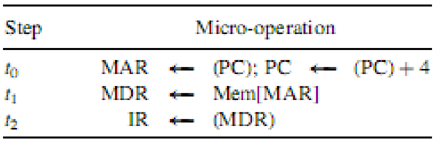
1. The contents of the PC are loaded into the MAR.
2. The value in the PC is incremented. (This operation can be done in parallel with a memory access.)
3. As a result of a memory read operation, the instruction is loaded into the MDR.
4. The contents of the MDR are loaded into the IR.

Let us consider the one-bus datapath organization shown in Fig. 3.3. We will see that the fetch operation can be accomplished in three steps as shown in the table below, where $t_0 < t_1 < t_2$ Note that multiple operations separated by “;” imply that they are accomplished in parallel.



Step	Micro-operation
t_0	MAR ← (PC); A ← (PC)
t_1	MDR ← Mem[MAR]; PC ← (A) + 4
t_2	IR ← (MDR)

Using the three-bus datapath shown in Figure 3.5, the following table shows the steps needed.



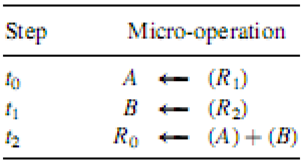
Step	Micro-operation
t_0	MAR ← (PC); PC ← (PC) + 4
t_1	MDR ← Mem[MAR]
t_2	IR ← (MDR)

3-4-2 Execute Simple Arithmetic Operation

Add R1, R2, R0: This instruction adds the contents of source registers R1 and R2, and stores the results in destination register R0. This addition can be executed as follows:

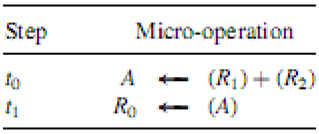
1. The registers R0, R1, R2, are extracted from the IR.
2. The contents of R1 and R2 are passed to the ALU for addition.
3. The output of the ALU is transferred to R0.

Using the one-bus datapath shown in Figure 3.3, this addition will take three steps as shown in the following table, where $t_0 < t_1 < t_2$.



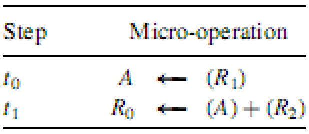
Step	Micro-operation
t_0	$A \leftarrow (R_1)$
t_1	$B \leftarrow (R_2)$
t_2	$R_0 \leftarrow (A) + (B)$

Using the two-bus datapath shown in Figure 3.4a, this addition will take two steps as shown in the following table, where $t_0 < t_1$.



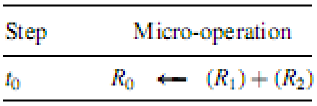
Step	Micro-operation
t_0	$A \leftarrow (R_1) + (R_2)$
t_1	$R_0 \leftarrow (A)$

Using the two-bus datapath with in-bus and out-bus shown in Figure 3.4b, this addition will take two steps as shown below, where $t_0 < t_1$.



Step	Micro-operation
t_0	$A \leftarrow (R_1)$
t_1	$R_0 \leftarrow (A) + (R_2)$

Using the three-bus datapath shown in Figure 3.5, this addition will take only one step as shown in the following table.

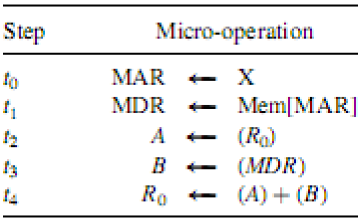


Step	Micro-operation
t_0	$R_0 \leftarrow (R_1) + (R_2)$

Add X, R0 This instruction adds the contents of memory location X to register R0 and stores the result in R0. This addition can be executed as follows:

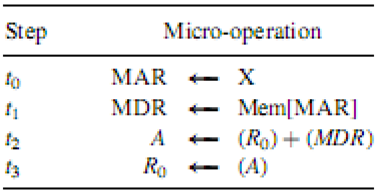
1. The memory location X is extracted from IR and loaded into MAR.
2. As a result of memory read operation, the contents of X are loaded into MDR.
3. The contents of MDR are added to the contents of R0.

Using the one-bus datapath shown this addition will take five steps as below, where $t_0 < t_1 < t_2 < t_3 < t_4$.



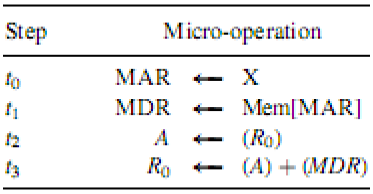
Step	Micro-operation
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	$A \leftarrow (R_0)$
t_3	$B \leftarrow (MDR)$
t_4	$R_0 \leftarrow (A) + (B)$

Using the two-bus datapath shown in this addition will take four Steps, where $t_0 < t_1 < t_2 < t_3$.



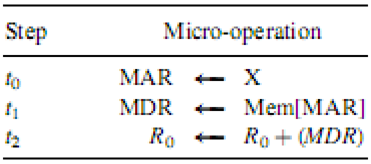
Step	Micro-operation
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	$A \leftarrow (R_0) + (MDR)$
t_3	$R_0 \leftarrow (A)$

Using the two-bus datapath with in-bus and out-bus shown this addition will take four steps, where $t_0 < t_1 < t_2 < t_3$.



Step	Micro-operation
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	$A \leftarrow (R_0)$
t_3	$R_0 \leftarrow (A) + (MDR)$

Using the three-bus datapath shown this addition will take three steps, where $t_0 < t_1 < t_2$.



Step	Micro-operation
t_0	MAR ← X
t_1	MDR ← Mem[MAR]
t_2	$R_0 \leftarrow R_0 + (MDR)$

3-4-3 Interrupt Handling

After the execution of an instruction, a test is performed to check for pending interrupts. If there is an interrupt request waiting, the following steps take place:

1. The contents of PC are loaded into MDR (to be saved).
2. The MAR is loaded with the address at which the PC contents are to be saved.
3. The PC is loaded with the address of the first instruction of the interrupt hand ling routine.
4. The contents of MDR (old value of the PC) are stored in memory. The following table shows the sequence of events, where $t_1 < t_2 < t_3$.

3-5 Control unit

The control unit is the main component that directs the system operations by sending control signals to the datapath. These signals control the flow of data within the CPU and between the CPU and external units such as memory and I/O. Control buses generally carry signals between the control unit and other computer components in a clock-driven manner. The system clock produces a continuous sequence of pulses in a specified duration and frequency. A sequence of steps t_0, t_1, t_2... ($t_1 < t_2 < t_3$…..) used to execute certain instruction. The op-code field of a fetched instruction is decoded to provide the control signal generator with information about the instruction to be executed. Step information generated by a logic circuit module is used with other inputs to generate control signals. The signal generator can be specified simply by a set of Boolean equations for its output in terms of its inputs. Figure 3.7 shows a block diagram that describes how timing is used in generating control signals.

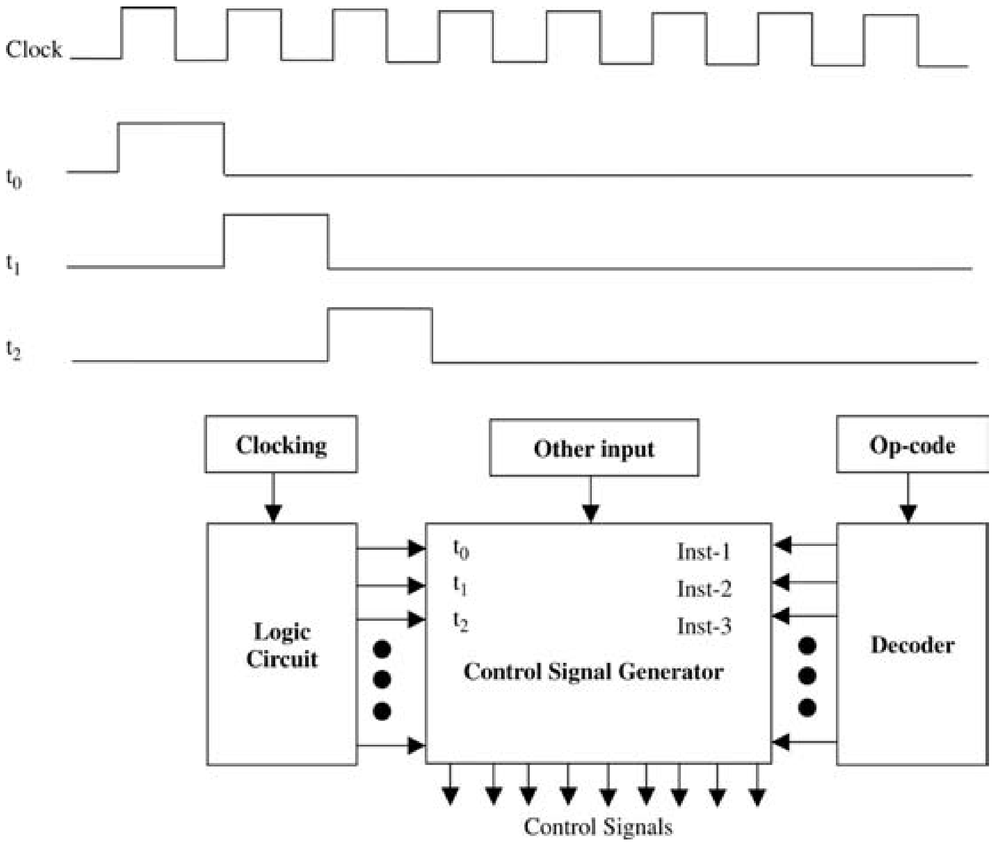


Figure 3.7 Timing of control signals

There are mainly two different types of control units: microprogrammed and hardwired. In microprogrammed control, the control signals associated with operations are stored in special memory units inaccessible by the programmer as control words. A control word is a microinstruction that specifies one or more micro-operations. A sequence of microinstructions is called a microprogram, which is stored in a ROM or RAM called a control memory CM.

In hardwired control, fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals. Clearly hardwired control is faster than microprogrammed control. However, hardwired control could be very expensive and complicated for complex systems. Hardwired control is more economical for small control units. It should also be noted that microprogrammed control could adapt easily to changes in the system design. We can easily add new instructions without changing hardware. Hardwired control will require a redesign of the entire systems in the case of any change.

Example 1 Let us revisit the add operation in which we add the contents of source registers R_1 , R_2 , and store the results in destination register R0 . We have shown earlier that this operation can be done in one step using the three-bus datapath shown in Figure 3.5.

Let us try to examine the control sequence needed to accomplish this addition at step t_0 . Suppose that the op-code field of the current instruction was decoded to Inst-x type. First we need to select the source registers and the destination register, then we select Add as the ALU function to be performed. The following table shows the needed step and the control sequence.

Step	Instruction type	Micro-operation	Control
t_0	Inst-x	$R_0 \leftarrow (R_1) + (R_2)$	Select R_1 as source 1 on out-bus1 (R_1 out-bus1) Select R_2 as source 2 on out-bus2 (R_2 out-bus2) Select R_0 as destination on in-bus (R_0 in-bus) Select the ALU function Add (Add)

Figure 3.8 shows the signals generated to execute Inst-x during time period t_0. The AND gate ensures that these signals will be issued when the op-code is decoded into Inst-x and during time period t_0. The signals (R_1 out-bus 1), (R_2 out-bus2), (R_0 in-bus), and (Add) will select R_1 as a source on out-bus1, R_2 as a source on out-bus2, R_0 as destination on in-bus, and select the ALUs add function, respectively.

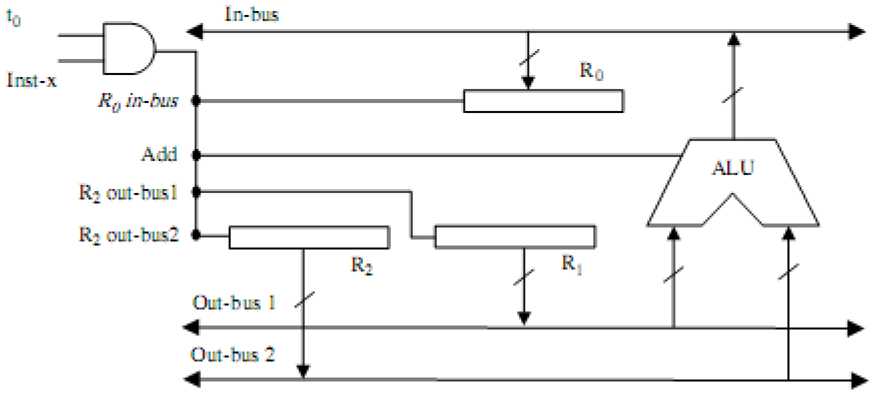
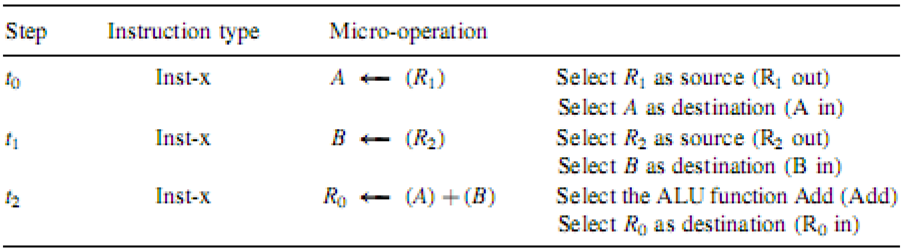


Figure 3.8 Signals generated to execute Inst-x on three-bus datapath during time period t_0

Example 2 Let us repeat the operation in the previous example using the one-bus datapath shown in Fig. 3.3. We have shown earlier that this operation can be carried out in three steps using the one-bus datapath.

Suppose that the op-code field of the current instruction was decoded to Inst-x type. The following table shows the needed steps and the control sequence.

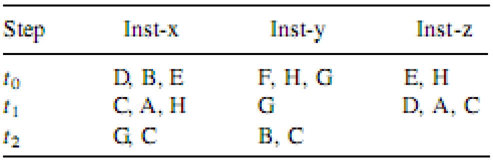


Step	Instruction type	Micro-operation	
t_0	Inst-x	$A \longleftarrow (R_1)$	Select R_1 as source (R_1 out) Select A as destination (A in)
t_1	Inst-x	$B \longleftarrow (R_2)$	Select R_2 as source (R_2 out) Select B as destination (B in)
t_2	Inst-x	$R_0 \longleftarrow (A) + (B)$	Select the ALU function Add (Add) Select R_0 as destination (R_0 in)

Figure 3.9 shows the signals generated to execute Inst-x during time periods t_0 , t_1 , and t_2 . The AND gates ensure that the appropriate signals will be issued when the op-code is decoded into Inst-x and during the appropriate time period. During t_0, the signals (R_1 out) and (A in) will be issued to move the contents of R_1 into A. Similarly during t_1, the signals (R_2 out) and (B in) will be issued to move the contents of R_2 into B. Finally, the signals (R_0 in) and (Add) will be issued during t_2 to add the contents of A and B and move the results into R_0 .

3-5-1. Hardwired Implementation

In hardwired control, a direct implementation is accomplished using logic circuits. For each control line, one must find the Boolean expression in terms of the input to the control signal generator as shown in Figure 3.7. Let us explain the implementation using a simple example. Assume that the instruction set of a machine has the three instructions: Inst-x, Inst-y, and Inst-z; and A, B, C, D, E, F, G, and H are control lines. The following table shows the control lines that should be activated for the three instructions at the three steps t_0 , t_1 , and t_2 .



Step	Inst-x	Inst-y	Inst-z
t_0	D, B, E	F, H, G	E, H
t_1	C, A, H	G	D, A, C
t_2	G, C	B, C	

The Boolean expressions for control lines A, B, and C can be obtained as follows:

$$\begin{aligned} \text{A} &= \text{Inst-x} \cdot t_1 + \text{Inst-z} \cdot t_1 = (\text{Inst-x} + \text{Inst-z}) \cdot t_1 \\ \text{B} &= \text{Inst-x} \cdot t_0 + \text{Inst-y} \cdot t_2 \\ \text{C} &= \text{Inst-x} \cdot t_1 + \text{Inst-x} \cdot t_2 + \text{Inst-y} \cdot t_2 + \text{Inst-z} \cdot t_1 \\ &= (\text{Inst-x} + \text{Inst-z}) \cdot t_1 + (\text{Inst-x} + \text{Inst-y}) \cdot t_2 \end{aligned}$$

Figure 3.10 shows the logic circuits for these control lines. Boolean expressions for the rest of the control lines can be obtained in a similar way. Figure 3.11 shows the state diagram in the execution cycle of these instructions.

3-5-2. Microprogrammed Control Unit

The idea of microprogrammed control units was introduced by M. V. Wilkes in the early 1950s.Microprogramming was motivated by the desire to reduce the complexities involved with hardwired control. As we studied earlier, an instruction is implemented using a set of microoperations. Associated with each micro-operation is a set of control lines that must be activated to carry out the corresponding micro-operation. The idea of microprogrammed control is to store the control signals associated with the implementation of a certain instruction as a microprogram in a special memory called a control memory (CM). A microprogram consists of a sequence of microinstructions. A microinstruction is a vector of bits, where each bit is a control signal, condition code, or the address of the next microinstruction Microinstructions are fetched from CM the same way program instructions are fetched from main memory (Fig. 3.12).

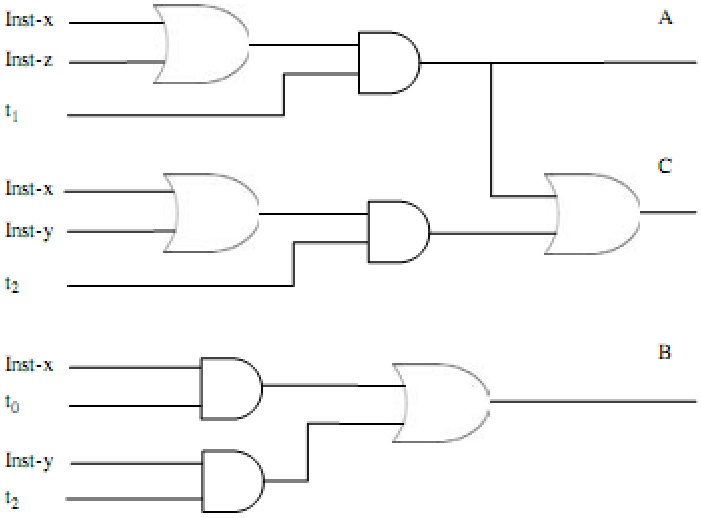


Figure 3.10 Logic circuits for control lines A, B, and C

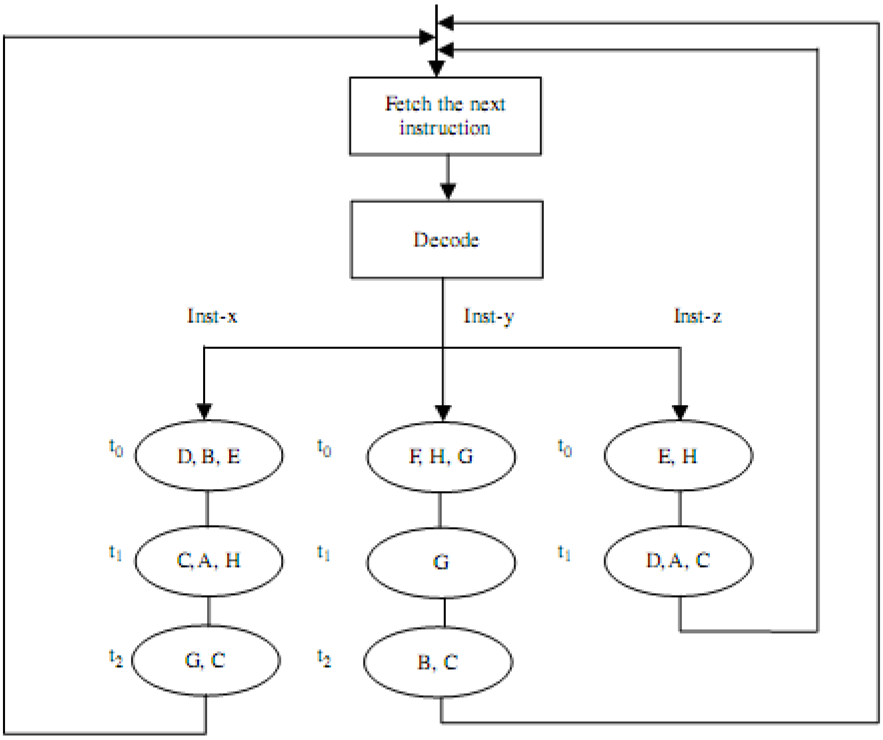


Figure 3.11 Instruction execution state diagram.

When an instruction is fetched from memory, the op-code field of the instruction will determine which microprogram is to be executed. In other words, the op-code is mapped to a microinstruction address in the control memory. The microinstruction processor uses that address to fetch the first microinstruction in the microprogram. After fetching each microinstruction, the appropriate control lines will be enabled. Every control line that corresponds to a “1” bit should be turned on. Every control line that corresponds to a “0” bit should be left off. After completing the execution of one microinstruction, a new microinstruction will be fetched and executed. If the condition code bits indicate that a branch must be taken, the next microinstruction is specified in the address bits of the current microinstruction. Otherwise, the next microinstruction in the sequence will be fetched and executed. The length of a microinstruction is determined based on the number of micro-operations specified in the microinstructions, the way the control bits will be interpreted, and the way the address of the next microinstruction is obtained. A microinstruction may specify one or more micro-operations that will be activated simultaneously.

The length of the microinstruction will increase as the number of parallel micro-operations per microinstruction increases. Furthermore, when each control bit in the microinstruction corresponds to exactly one control line, the length of microinstruction could get bigger. The length of a microinstruction could be reduced if control lines are coded in specific fields in the microinstruction. Decoders will be needed to map each field into the individual control lines. Clearly, using the decoders will reduce the number of control lines that can be activated simultaneously. There is a tradeoff between the length of the microinstructions and the amount of parallelism. It is important that we reduce the length of microinstructions to reduce the cost and access time of the control memory. It may also be desirable that more micro-operations be performed in parallel and more control lines can be activated simultaneously.

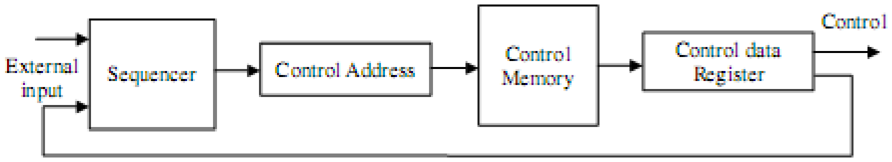


Figure 3.12 Fetching microinstructions (control words)

Horizontal Versus Vertical Microinstructions can be classified as horizontal or vertical. Individual bits in horizontal microinstructions correspond to individual control lines. Horizontal microinstructions are long and allow maximum parallelism since each bit controls a single control line. In vertical microinstructions, control lines are coded into specific fields within a microinstruction. Decoders are needed to map a field of k bits to 2k possible combinations of control lines. For example, a 3-bit field in a microinstruction could be used to specify any one of eight possible lines. Because of the encoding, vertical microinstructions are much shorter than horizontal ones. Control lines encoded in the same field cannot be activated simultaneously. Therefore, vertical micro-instructions allow only limited parallelism. It should be noted that no decoding is needed in horizontal microinstructions while decoding is necessary in the vertical case.

3-6 Summary

In this lesson, we have discussed in detail about register organisation and simple structure of CPU. We have also discussed in detail about the data path and their implementation in hardware using simple circuits and CPU instruction cycle and control unit.

3-7 Keywords

Arithmetic logic unit: An arithmetic-logic unit (*ALU*) is the part of a computer processor (CPU) that carries out arithmetic and logic operations on the operands in computer.

Memory Data Register (MDR): The memory data register (MDR) is the register of a computer's control unit that contains the data to be stored in the computer storage (e.g. RAM), or the data after a fetch from the computer storage.

Memory Address Register (MAR): The memory address register holds the address of the next memory location where the next instruction is to be executed.

NOTES

MIPS: Million instructions per second

Datapath: The CPU can be divided into a data section and a control section. The data section, which is also called the datapath, contains the registers and the ALU.

Instruction cycle: The time period during which one instruction is fetched from memory and executed when a computer is given an instruction in machine language.

3-8 Exercise

1. How many instruction bits are required to specify the following:

(a) Two operand registers and one result register in a machine that has 64 general-purpose registers?

(b) Three memory addresses in a machine with 64 KB of main memory?

2. Show the micro-operations of the load, store, and jump instructions using:

(a) One-bus system

(b) Two-bus system

(c) Three-bus system

3. Data movement within the CPU can be performed in several different ways. Contrast the following methods in terms of their advantages and disadvantages:

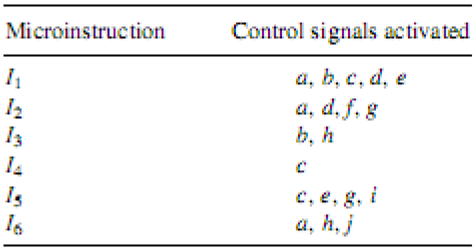
(a) Dedicated connections

(b) One-bus datapath

(c) Two-bus datapath

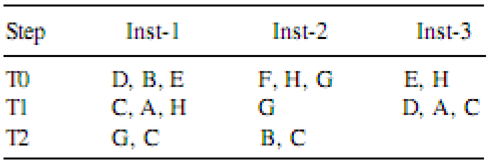
(d) Three-bus datapath

4. Find a method of encoding the microinstructions described by the following table so that the minimum number of control bits is used and all inherent parallelism among the microoperations is preserved.



Microinstruction	Control signals activated
I_1	a, b, c, d, e
I_2	a, d, f, g
I_3	b, h
I_4	c
I_5	c, e, g, i
I_6	a, h, j

5. Suppose that the instruction set of a machine has three instructions: Inst-1, Inst-2, and Inst-3; and A, B, C, D, E, F, G, and H are the control lines. The following table shows the control lines that should be activated for the three instructions at the three steps T_0, T_1, and T_2.



Step	Inst-1	Inst-2	Inst-3
T0	D, B, E	F, H, G	E, H
T1	C, A, H	G	D, A, C
T2	G, C	B, C	

(a) Hardwired approach:

(i) Write Boolean expressions for all the control lines A–G.

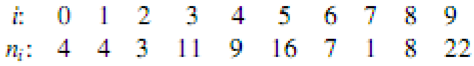
(ii) Draw the logic circuit for each control line.

(b) Microprogramming approach:

(i) Assuming a horizontal representation, write down the microprogram for instructions Inst-1. Indicate the microinstruction size.

(ii) If we allow both horizontal and vertical representation, what would be the best grouping? What is the microinstruction size? Write the microprogram of Inst-1

6. A certain processor has a microinstruction format containing 10 separate control fields $C_0 : C_9$. Each C_i can activate any one of n_i distinct control lines, where n_i is specified as follows:



i:	0	1	2	3	4	5	6	7	8	9
n_i:	4	4	3	11	9	16	7	1	8	22

(a) What is the minimum number of control bits needed to represent the 10 control fields?

(b) What is the maximum number of control bits needed if a purely horizontal format is used for all control information?

3-9 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

4. INSTRUCTION SET ARCHITECTURE

4.1 Introduction

In this lesson we tackle a central topic in computer architecture: the language understood by the computer's hardware, referred to as its machine language. The machine language is usually discussed in terms of its assembly language, which is functionally equivalent to the corresponding machine language except that the assembly language uses more intuitive names such as Move, Add, and Jump instead of the actual binary words of the language. (Programmers find con-structs such as “Add r0, r1, r2” to be more easily understood and rendered with-out error than 0110101110101101.)

In order to describe the nature of assembly language and assembly language programming, we choose as a model architecture the ARC machine, which is a simplification of the commercial SPARC architecture common to Sun computers. (Additional architectural models are covered in The Computer Architecture Companion volume.)

We illustrate the utility of the various instruction classes with practical examples of assembly language programming, and we conclude with a Case Study of the Java bytecodes as an example of a common, portable assembly language that can be implemented using the native language of another machine.

4.2 Hardware Components of the Instruction Set Architecture

The ISA of a computer presents the assembly language programmer with a view of the machine that includes all the programmer-accessible hardware, and the instructions that manipulate data within the hardware. In this section we look at the hardware components as viewed by the assembly language programmer. We begin with a discussion of the system as a whole: the CPU interacting with its internal (main) memory and performing input and output with the outside world.

4.2.1 THE SYSTEM BUS MODEL REVISITED

Figure 4-1 revisits the system bus model. The purpose of the bus is to reduce the number of interconnections between the CPU and its subsystems. Rather than have separate communication paths between memory and each I/O device, the CPU is interconnected with its memory and I/O systems via a shared system bus. In more complex systems there may be separate busses between the CPU and memory and CPU and I/O.

Not all of the components are connected to the system bus in the same way. The CPU generates addresses that are placed onto the address bus, and the memory receives addresses from the address bus. The memory never generates addresses, and the CPU never receives addresses, and so there are no corresponding connections in those directions.

In a typical scenario, a user writes a high level program, which a compiler translates into assembly language. An assembler then translates the assembly language program into machine code, which is stored on a disk. Prior to execution, the machine code program is loaded from the disk into the main memory by an operating system.

During program execution, each instruction is brought into the ALU from the memory, one instruction at a time, along with any data that is needed to execute the instruction. The output of the program is placed on a device such as a video display, or a disk. Communication among the three compoents (CPU, Memory, and I/O) is handled with busses.

An important consideration is that the instructions are executed inside of the ALU, even though all of the instructions and data are initially stored in the memory. This means that instructions and data must be loaded from the memory into the ALU registers, and results must be stored back to the memory from the ALU registers.

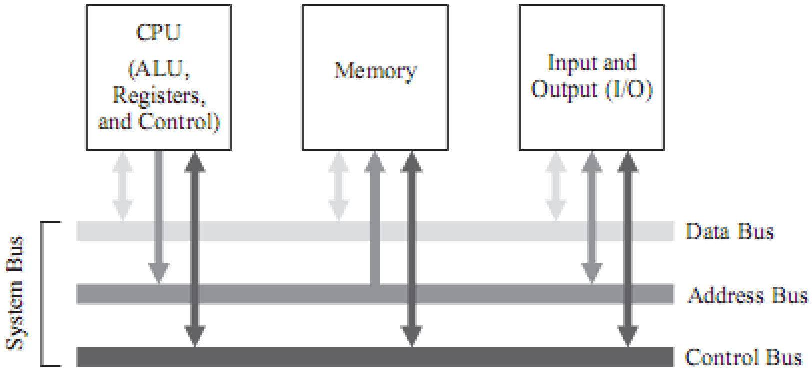


Figure 4-1 The system bus model of a computer system.

4-3 CPU

Now that we are familiar with the basic components of the system bus and memory, we are ready to explore the internals of the CPU. At a minimum, the CPU consists of a data section that contains registers and an ALU, and a control section, which interprets instructions and effects register transfers, as illustrated in Figure 4-2. The data section is also referred to as the datapath.

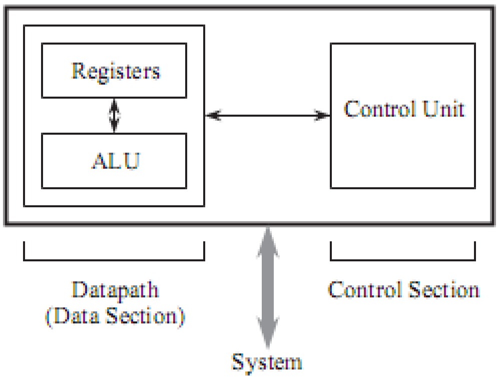


Figure 4-2 High level view of a CPU.

The control unit of a computer is responsible for executing the program instructions, which are stored in the main memory. There are two registers that form the interface between the control unit and the data unit, known as the program counter (PC) and the instruction register (IR). The PC contains the address of the instruction being executed. The instruction that is pointed to by the PC is fetched from the memory, and is stored in the IR where it is interpreted. The steps that the control unit carries out in executing a program are:

1) Fetch the next instruction to be executed from memory.

2) Decode the opcode.

3) Read operand(s) from main memory, if any.

4) Execute the instruction and store results.

5) Go to step 1.

This is known as the fetch-execute cycle.

The control unit is responsible for coordinating these different units in the execution of a computer program. It can be thought of as a form of a “computer within a computer” in the sense that it makes decisions as to how the rest of the machine behaves. The datapath is made up of a collection of registers known as the register file and the arithmetic and logic unit (ALU), as shown in Figure 4-3.

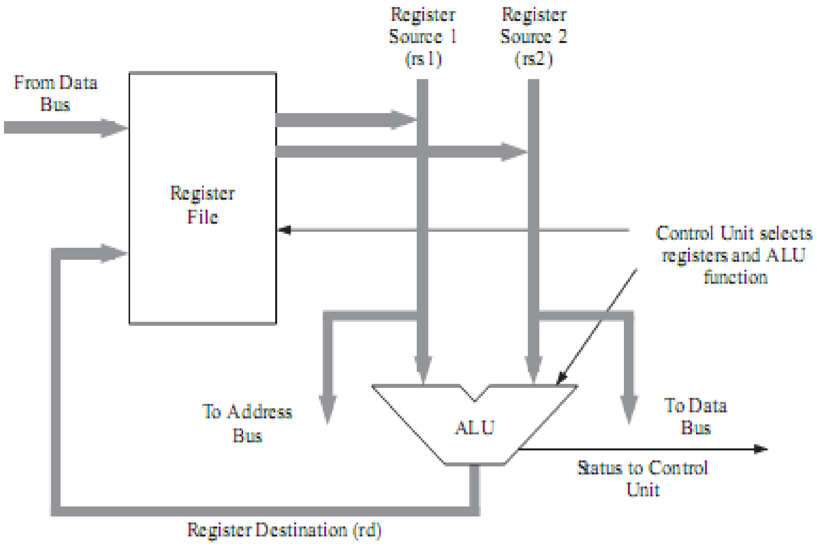


Figure 4-3 An example datapath

The register file in the figure can be thought of as a small, fast memory, separate from the system memory, which is used for temporary storage during computation. Typical sizes for a register file range from a few to a few thousand registers. Like the system memory, each register in the register file is assigned an address in sequence starting from zero. These register "addresses" are much smaller than main memory addresses: a register file containing 32 registers would have only a 5-bit address, for example. The major differences between the register file and the system memory is that the register file is contained within the CPU, and is therefore much faster. An instruction that operates on data from the register file can often run ten times faster than the same instruction that operates on data in memory. For this reason, register-intensive programs are faster than the equivalent memory intensive programs, even if it takes more register operations to do the same tasks that would require fewer operations with the operands located in memory.

Notice that there are several busses inside the datapath of Figure 4-3. Three busses connect the datapath to the system bus. This allows data to be transferred to and from main memory and the register file. Three additional busses connect the register file to the ALU. These busses allow two operands to be fetched from the register file simultaneously, which are operated on by the ALU, with the results returned to the register file.

The ALU implements a variety of binary (two-operand) and unary (one-operand) operations. Examples include add, and, not, or, and multiply. Operations and operands to be used during the operations are selected by the Control Unit. The two source operands are fetched from the register file onto busses labeled “Register Source 1 (rs1)” and “Register Source 2 (rs2).” The output from the ALU is placed on the bus labeled “Register Destination (rd),” where the results are conveyed back to the register file. In most systems these connections also include a path to the System Bus so that memory and devices can be accessed. This is shown as the three connections labeled “From Data Bus”, “To Data Bus”, and “To Address Bus.”

4-4 Instruction Set

The instruction set is the collection of instructions that a processor can execute, and in effect, it defines the processor. The instruction sets for each processor type are completely different one from the other. They differ in the sizes of instructions, the kind of operations they allow, the type of operands they operate on, and the types of results they provide.This incompatibility in instruction sets is in stark contrast to the compatibility of higher level languages such as C, Pascal, and Ada. Programs written in these higher level languages can run almost unchanged on many different processors if they are re-compiled for the target processor.

Because of this incompatibility among instruction sets, computer systems are often identified by the type of CPU that is incorporated into the computer system. The instruction set determines the programs the system can execute and has a significant impact on performance. Programs compiled for an IBM PC (or compatible) system use the instruction set of an 80x86 CPU, where the ‘x’ is replaced with a digit that corresponds to the version, such as 80586, more commonly referred to as a Pentium processor. These programs will not run on an Apple Macintosh or an IBM RS6000 computer, since the Macintosh and IBM machines execute the instruction set of the Motorola PowerPC CPU. This does not mean that all computer systems that use the same CPU can execute the same programs, however. A PowerPC program written for the IBM RS6000 will not execute on the Macintosh without extensive modifications, however, because of differences in operating systems and I/O conventions.

4-5 Software for generating machine language programs

A compiler is a computer program that transforms programs written in a high-level language such as C, Pascal, or Fortran into machine language. Compilers for the same high level language generally have the same “front end,” the part that recognizes statements in the high-level language. They will have different “back ends,” however, one for each target processor. The compiler’s back end is responsible for generating machine code for a specific target processor. On the other hand, the same program, compiled by different C compilers for the same machine can produce different compiled programs for the same source code, as we will see. In the process of compiling a program (referred to as the translation process), a high-level source program is transformed into assembly language, and the assembly language is then translated into machine code for the target machine by an assembler. These translations take place at compile time and assembly time, respectively. The resulting object program can be linked with other object programs, at link time. The linked program, usually stored on a disk, is loaded into main memory, at load time, and executed by the CPU, at run time. Although most code is written in high level languages, programmers may use assembly language for programs or fragments of programs that are time or space-critical. In addition, compilers may not be available for some special purpose processors, or their compilers may be inadequate to express the special operations which are required. In these cases also, the programmer may need to resort to programming in assembly language.

High level languages allow us to ignore the target computer architecture during coding. At the machine language level, however, the underlying architecture is the primary consideration. A program written in a high level language like C, Pascal, or Fortran may look the same and execute correctly after compilation on several different computer systems. The object code that the compiler produces for each machine, however, will be very different for each computer system, even if the systems use the same instruction set, such as programs compiled for the PowerPC but running on a Macintosh vs. running on an IBM RS6000. Having discussed the system bus, main memory, and the CPU, we now examine details of a model instruction set, the ARC.

4-6 ARC, RISC ,CISC Computer

A model architecture that is based on the commercial Scalable Processor Architecture (SPARC) processor that was developed at Sun Microsystems in the mid-1980’s. The SPARC has become a popular architecture since its introduction, which is partly due to its “open” nature: the full definition of the SPARC architecture is made readily available to the public (SPARC, 1992). In this chapter, we will look at just a subset of the SPARC, which we call “A RISC Computer” (ARC). “RISC” is yet another acronym, for reduced instruction set computer. “CISC “for Complex instruction set computer

The ARC has most of the important features of the SPARC architecture, but without some of the more complex features that are present in a commercial processor.

4-7 CISC to RISC

Historically, when memory cycle times were very long and when memory prices were high, fewer, complicated instructions held an advantage over more, simpler instructions. There came a point, however, when memory became inexpensive enough and memory hierarchies became fast and large enough, that computer architects began reexamining this advantage. One technology that affected this examination was pipelining that is, keeping the execution unit more or less the same, but allowing different instructions (which each require several clock cycles to execute) to use different parts of the execution unit on each clock cycle. For example, one instruction might be accessing operands in the register file while another is using the ALU.

We will cover pipelining in more detail later in the chapter, but the important point to make here is that computer architects learned that CISC instructions do not fit pipelined architectures very well. For pipelining to work effectively, each instruction needs to have similarities to other instructions, at least in terms of relative instruction complexity. The reason can be viewed in analogy to an assembly line that produces different models of an automobile. For efficiency, each “station” of the assembly line should do approximately the same amount and kind of work. If the amount or kind of work done at each station is radically different for different models, then periodically the assembly line will have to “stall” to accommodate the requirements of the given model.

CISC instruction sets have the disadvantage that some instructions, such as register-to-register moves, are inherently simple, whereas others, such as the MVC instruction and others like it are complex, and take many more clock cycles to execute.

The main philosophical underpinnings of the RISC approach are:

• Prefetch instructions into an instruction queue in the CPU before they are needed. This has the effect of hiding the latency associated with the instruction fetch.

• With instruction fetch times no longer a penalty, and with cheap memory to hold a greater number of instructions, there is no real advantage to

CISC instructions. All instructions should be composed of sequences of RISC in structions, even though the number of instructions needed may increase (typically by as much as 1/3 over a CISC approach).

- Moving operands between registers and memory is expensive, and should be minimized.

- The RISC instruction set should be designed with pipelined architectures in mind.

- There is no requirement that CISC instructions be maintained as integrated wholes; they can be decomposed into sequences of simpler RISC instructions.

The result is that RISC architectures have characteristics that distinguish them from CISC architectures:

- All instructions are of fixed length, one machine word in size.

- All instructions perform simple operations that can be issued into the pipeline at a rate of one per clock cycle. Complex operations are now composed of simple instructions by the compiler.

- All operands must be in registers before being operated upon. There is a separate class of memory access instructions: LOAD and STORE. This is referred to as a LOAD-STORE architecture.

- Addressing modes are limited to simple ones. Complex addressing calculations are built up using sequences of simple operations.

- There should be a large number of general registers for arithmetic operations so that temporary variables can be stored in registers rather than on a stack in memory.

4-8 Pipelining the Datapath

The flow of instructions through a pipeline follows the steps normally taken when an instruction is executed. In the discussion below we consider how three classes of instructions: arithmetic, branch, and load-store, are executed, and then we relate this to how the instructions are pipelined.

4-8-1 ARITHMETIC, BRANCH, AND LOAD-STORE INSTRUCTIONS

Consider the “normal” sequence of events when an arithmetic instruction is executed in a load-store machine:

1) Fetch the instruction from memory;

2) Decode the instruction (it is an arithmetic instruction, but the CPU has to find that out through a decode operation);

3) Fetch the operands from the register file;

4) Apply the operands to the ALU;

5) Write the result back to the register file.

There are similar patterns for other instruction classes. For branch instructions the sequence is:

1) Fetch the instruction from memory;

2) Decode the instruction (it is a branch instruction);

3) Fetch the components of the address from the instruction or register file;

4) Apply the components of the address to the ALU (address arithmetic);

5) Copy the resulting effective address into the PC, thus accomplishing the branch.

The sequence for load and store instructions is:

1) Fetch the instruction from memory;

2) Decode the instruction (it is a load or store instruction);

3) Fetch the components of the address from the instruction or register file;

4) Apply the components of the address to the ALU (address arithmetic);

5) Apply the resulting effective address to memory along with a read (load) or write (store) signal. If it is a write signal, the data item to be written must also be retrieved from the register file.

The three sequences above show a high degree of similarity in what is done at each stage: (1) fetch, (2) decode, (3) operand fetch, (4) ALU operation, (5) result writeback. These five phases are similar to the four phases discussed in chapters 4 and 6 except that we have refined the fourth phase, “execute,” into two subphases: ALU operation and writeback, as illustrated in Figure 4-4. A result writeback is not always needed, and one way to deal with this is to have two separate subphases (ALU and writeback) with a bypass path for situations when a writeback is not needed. For this discussion, we take a simpler approach, and force all

NOTES

instructions to go entirely through each phase, whether or not that is actually needed.

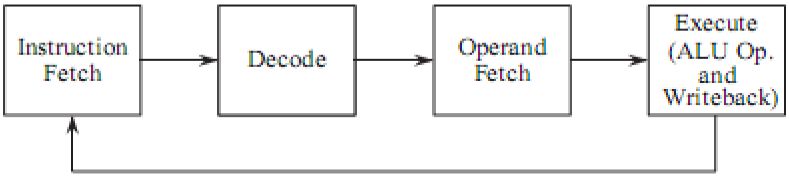


Figure 4-4 Four-stage instruction pipeline.

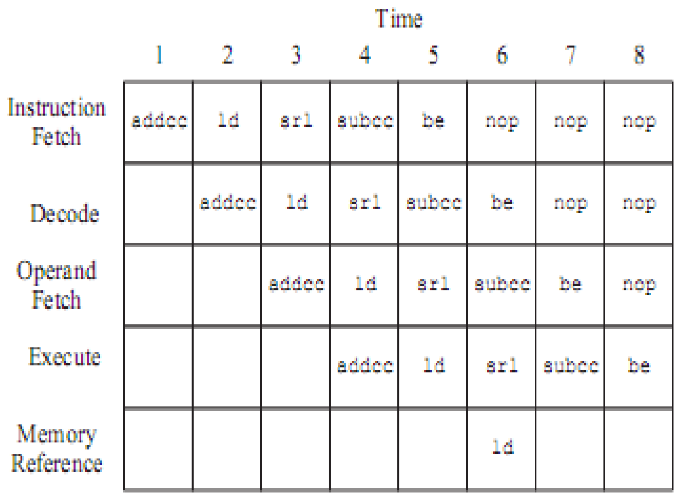
4-9 PIPELINING INSTRUCTIONS

In practice, each CPU designer approaches the design of the pipeline from a different perspective, depending upon the particular design goals and instruction set. For example the original SPARC implementation had only four pipeline stages, while some floating point pipelines may have a dozen or more stages.

Each of the execution units performs a different operation in the fetch-execute cycle. After the Instruction Fetch unit finishes its task, the fetched instruction is handed off to the Decode unit. At this point, the Instruction Fetch unit can begin fetching the next instruction, which overlaps with the decoding of the previous instruction. When the Instruction Fetch and Decode units complete their tasks, they hand off the remaining tasks to the next units (Operand Fetch is the next unit for Decode). The flow of control continues until all units are filled.

4-10 KEEPING THE PIPELINE FILLED

Notice an important point: although it takes multiple steps to execute an instruction in this model, on average, one instruction can be executed per cycle as long as the pipeline stays filled. The pipeline does not stay filled, however, unless we are careful as to how instructions are ordered. We know from Figure 4-4 that approximately one in every four instructions is a branch. We cannot fetch the instruction that follows a branch until the branch completes execution. Thus, as soon as the pipeline fills, a branch is encountered, and then the pipeline has to be flushed by filling it with no-operations (NOPs). These NOPs are sometimes referred to as pipeline bubbles. A similar situation arises with the LOAD and STORE instructions. They generally require an additional clock cycle in which to access memory, which has the effect of expanding the Execute phase from one cycle to two cycles at times. The “wait” cycles are filled with NOPs. Figure 4-5 illustrates the pipeline behavior during a memory reference and also during a branch for the ARC instruction set. The addcc instruction enters the pipeline on time step (cycle) 1. On cycle 2, the ld instruction, which references memory, enters the pipeline and addcc moves to the Decode stage. The pipeline continues filling with the srl and subcc instructions on cycles 3 and 4, respectively. On cycle 4, the addcc instruction is executed and leaves the pipeline. On cycle 5, the ld instruction reaches the Execute level, but does not finish execution because an additional cycle is needed for memory references. The ld instruction finishes execution during cycle 6.



	Time							
	1	2	3	4	5	6	7	8
Instruction Fetch	addcc	ld	srl	subcc	be	nop	nop	nop
Decode		addcc	ld	srl	subcc	be	nop	nop
Operand Fetch			addcc	ld	srl	subcc	be	nop
Execute				addcc	ld	srl	subcc	be
Memory Reference						ld		

Figure 4-5 Pipeline behavior during a memory reference and a branch.

4-11 References:

1. Computer System Architecture   By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha   —BPB Publications

3. Computer Organisation And Architecture By: William Stallings   — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

UNIT - III

1. NUMBER SYSTEMS

Structure

Objectives

At the end of lesson you will be able to:

- Describe the decimal number system
- Describe the binary number system
- Discuss the binary arithmetic operations
- Describe the floating-point number

1-1 Introduction

This lesson is dedicated to a discussion on computer arithmetic. Our goal is to introduce the reader to the fundamental issues related to the arithmetic operations and circuits used to support computation in computers. Our coverage starts with an introduction to number systems. In particular, we introduce issues such as number representations and base conversion. This is followed by a discussion on integer arithmetic. In this regard, we introduce a number of algorithms together with hardware schemes that are used in performing integer addition, subtraction, multiplication, and division. We end this chapter with a discussion on floating point arithmetic. In particular, we introduce issues such as floating-point representation, floating-point operations, and floating-point hardware schemes.

1.2 Decimal Number System

The decimal number system has ten unique digits 0, 1, 2, 3… 9. Using these single digits, ten different values can be represented. Values greater than ten can be represented by using the same digits in different combinations. Thus ten is represented by the number 10; two hundred seventy five is represented by 275 etc. Thus same set of numbers 0, 1, 2… 9 are repeated in a specific order to represent larger numbers.

The decimal number system is a positional number system as the position of a digit represents its true magnitude. For example, 2 is less than 7, however 2 in 275 represents 200, whereas 7 represents 70. The left most digit has the highest weight and the right most digit has the lowest weight. 275 can be written in the form of an expression in terms of the base value of the number system and weights.

$2 \text{ x } 10^2 + 7 \text{ x } 10^1 + 5 \text{ x } 10^0 = 200 + 70 + 5 = 275$

Where, 10 represent the base or radix

10^2, 10^1, 10^0 represent the weights 100, 10 and 1 of the numbers 2, 7 and 5

1-3 Binary Number System

The Caveman Number system is a hypothetical number system introduced to explain that number system other than the Decimal Number system can exist and can be used to represent and count numbers. Digital systems use a Binary number system. Binary as the name indicates is a Base-2 number system having only two numbers 0 and 1. The Binary digit 0 or 1 is known as a 'Bit'. Table

Decimal Number	Binary Number	Decimal Number	Binary Number
0	0	10	1010
1	1	11	1011
2	10	12	1100

3	11	13	1101
4	100	14	1110
5	101	15	1111
6	110	16	10000
7	111	17	10001
8	1000	18	10010
9	1001	19	10011
		20	10100

Counting in Binary Number system is similar to counting in Decimal or Caveman Number systems. In a decimal Number system a value larger than 9 has to be represented by 2, 3, 4 or more digits. In the Caveman Number System a value larger than 4 has to be represented by 2, 3, 4 or more digits of the Caveman Number System. Similarly, in the Binary Number System a Binary number larger than 1 has to be represented by 2, 3, 4 or more binary digits.

1-4 Binary Arithmetic

Digital systems use the Binary number system to represent numbers. Therefore these systems should be capable of performing standard arithmetic operations on binary numbers.

1-4-1 Binary Addition

Binary Addition is identical to Decimal Addition. By adding two binary bits a Sum bit and a Carry bit are generated. The only difference between the two additions is the range of numbers used. In Binary Addition, four possibilities exist when two single bits are added together. The four possible input combinations of two single bit binary numbers and their corresponding Sum and Carry Outputs are specified in table.

First Number	Second Number	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The first three additions give a result 0, 1 and 1 respectively which can be represented by a single binary digit (bit). The fourth addition results in the number 2, which can be represented in binary as 10_2. Thus two digits (bits) are required. This is similar to the addition of 9 + 3 in decimal. The answer is 12 which can not be represented by a single digit, thus two digits are required. The number 2 is the sum part and 1 is the carry part.

Any number of binary numbers having any number of digits can be added together. Thus the number 1011, 110, 1000 and 11 can be added together. Most significant digits (bits) of second and fourth numbers are assumed to be zero.

Carry		1	10	1		Decimal Equivalent
1st Number		1	0	1	1	(11)
2nd Number			1	1	0	(06)
3rd Number		1	0	0	0	(08)
4th Number				1	1	(03)
Result	1	1	1	0	0	(28)

1-4-2 Binary Subtraction

Binary Subtraction is identical to Decimal Subtraction. The only difference between the two is the range of numbers. Subtracting two single bit binary numbers results in a difference bit and a borrow bit. The four possible input combinations of two single bit binary numbers and their corresponding Difference and Borrow Outputs are specified in table. It is assumed that the second number is subtracted from the first number.

First Number	Second Number	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The second subtraction subtracts 1 from 0 for which a Borrow is required to make the first digit equal to 2. The Difference is 1. This is similar to decimal subtraction when 17 is subtracted from 21. The first digit 7 can not be subtracted from 1, therefore 10 is borrowed from the next significant digit. Borrowing a 10 allows subtraction of 7 from 11 resulting in a Difference of 4.

1-4-3 Binary Multiplication

Binary Multiplication is similar to the Decimal multiplication except for the range of numbers. Four possible combinations of two single bit binary numbers and their products are listed in table

First Number	Second Number	Product
0	0	0
0	1	0
1	0	0
1	1	1

Multiplying two binary numbers such as 1101 x 101 is performed by a shift and add operation. The binary multiplication shifts and adds partial product terms.

```
       1101
   X    101
      1101         1st product term
     0000          2nd product term
    1101           3rd product term
   1000001
```

1-4-4. Binary Multiplication by shifting left

Binary Multiplication can be performed by shifting the binary number towards left. A left shift by a single bit is equivalent to multiplication by 2. A left shift by two bits is equivalent to multiplication by 4. Generally, the multiplication factor is determined by 2^n where n is the number of bit shifts.

00011	(3)	original binary number
00110	(6)	binary number shifted left by 1 bit
01100	(12)	binary number shifted left by 2 bits
11000	(24)	binary number shifted left by 3 bits

1-4-5. Binary Division

Division in binary follows the same procedure as in the division of decimal numbers. An example illustrates the division of binary numbers

```
    10
101|1101
    101
     011
     000
      11
```

1-4-6. Binary Division by shifting right

Binary Division can be performed by shifting the binary number towards right. A right shift by a single bit is equivalent to division by 2. A right shift by two bits is equivalent to division by 4. Generally, the division factor is determined by 2^n where n is the number of bit shifts.

10100	(20)	original binary number
01010	(10)	binary number shifted right by 1 bit
00101	(5)	binary number shifted right by 2 bits

1-4-7 Signed and Unsigned Binary Numbers

Digital systems not only handle positive numbers but both positive and negative numbers. In the decimal number system positive numbers are identified by the + sign and negative numbers are represented by the – sign.

In a digital system which uses the Binary number system, the positive and negative signs can not be represented as + and -. As mentioned in the Overview all forms of numbers, text, punctuation marks etc. are represented in the form of 1s and 0s. Thus the positive and negative signs are also presented in terms of binary 0 and 1.

To handle positive and negative binary numbers, the digital system sets aside the most significant digit (bit) to represent the sign

- MSB set to 1 indicates a negative number
- MSB set to 0 indicates a positive number

Thus +13 and -13 are represented as 01101and 11101 respectively. The bits 1101 represent the number 13 and the MSBs 0 and 1 represent positive and negative signs respectively. Thus binary numbers having the MSB signifying the Sign bit are treated as Signed Binary Numbers. This representation is known as the Signed Magnitude representation.

Digital systems also handle binary numbers which are assumed to be positive and therefore do not have the most significant sign bit. Such numbers are known as unsigned numbers. Digital system thus have to handle two different types of binary numbers, signed and unsigned. Thus 11101_2 represent -13 in signed binary and 29 in unsigned binary. How should a Digital System treat a binary number? Should it consider it as a signed or unsigned number? A digital system on its own can not decide how to handle a binary number. The digital system has to be notified beforehand to deal with a certain binary representation as signed or unsigned.

1’s & 2’s complement

Informing the digital system how to treat a binary number is not very efficient. A better way is to represent negative signed numbers in their 2’s complement form. Using 2’s Complement form to represent signed numbers, allows direct manipulation of positive as well as negative numbers without having to worry about setting the most significant sign bit to indicate positive and negative numbers.

A 2’s complement of a number is obtained by first taking the 1’s complement of a number and then adding a 1 to change the 1’s complement to 2’s complement. 1’s complement of a number is obtained by simply inverting all its bits. Obtaining the 2’s complement of 13 is described in the example below.

```
01101    The number 13
10010    1’s complement of 13 is obtained by inverting all the five bits.
   +1
10011    2’s complement of 13 is obtained by adding a 1 to its 1’s
complement.
```

In a 2’s complement number system all negative numbers are represented in their 2’s complement form and all positive numbers are represented in their actual form. Negative numbers can be readily identified by their MSBs which are set to 1. Thus in a 2’s complement representation +13 is represented as 01101 and -13 is represented as 10011.

By having numbers represented in their 2’s complement form addition and subtraction operations can easily be performed without having to worry about the sign bits. Thus +13 added to -13 should result in a zero value. This can be verified by directly adding the +13 and -13 in their 2’s complement forms.

```
 01101
 10011
100000
```

The most significant carry bit is discarded; retaining only the first 5 bits proves that adding +13 and -13 results in a zero value. Similarly it can be shown that adding the numbers +7 and -13 results in -6.

```
10011        (-13)
00111        (+7)
11010        (-6)
```

The binary 2’s complement number 11010 has its most significant bit set to 1 indicating that the number is negative. The actual magnitude of the negative number is determined by taking the 2’s complement of 11010.

```
11010   Original number
00101   1’s complement of Original number
  + 1
00110   2’s complement of Original number is equal to 6.
```

1-4-8 Range of Signed and Unsigned Binary numbers

Three different types of Binary representations have been discussed. The Unsigned Binary representation can only represent positive binary numbers. The Sign- Magnitude can represent both positive and negative numbers. The 2's complement signed representation also allows positive and negative numbers to be handled.

Each of the three binary number representations can represent certain range of binary numbers determined by the total number of bits used.

The maximum range of values that can be represented in any number system depends upon the number of digits assigned to represent the value. A 5-digit car odometer can only count up to 99,999 and then it rolls back to 00000. Similarly an 8-digit calculator can only handle integer numbers of the magnitude 99,999,999. A calculator that reserves the most significant digit to write + or – can only handle a maximum range of integer numbers from -9,999,999 to +9,999,999.

A 3-bit unsigned binary number can have values ranging between 000 and 111. Adding 100 and 111 unsigned numbers results in 1011, this result is considered to be out of range as 4 bits are required. Similarly a 4-bit sign magnitude number can handle a number range between -7 and +7. -8 can not be represented as 5-bits are required 11000. A 4-bit 2's complement based signed number range is between -8 to +7.

The table shows the range of values that can be represented by the three Binary representations using 4-bits.

Decimal Number	Sign-Magnitude form	2's complement form	Unsigned form
-8		1000	
-7	1111	1001	
-6	1110	1010	
-5	1101	1011	
-4	1100	1100	
-3	1011	1101	
-2	1010	1110	
-1	1001	1111	
0	0000	0000	000
1	0001	0001	001
2	0010	0010	010
3	0011	0011	011
4	0100	0100	100
5	0101	0101	101
6	0110	0110	110
7	0111	0111	111

- Signed Magnitude representation can represent positive and negative numbers in the range (2^{n-1}-1) and - (2^{n-1}-1) where n represents the number of bits.
- 2's complement signed representation can represent positive and negative numbers in the range (2^{n-1}-1) and - (2^{n-1}) where n represents the number of bits.
- The unsigned representation can represent positive numbers in the range 0 to 2^{n}-1, where n represents the number of bits.

1-4-9 Range of Numbers and Overflow

When arithmetic operation such as Addition, Subtraction, Multiplication and Division are performed on numbers the results generated may exceed the range of values specified by the Binary representations. The values that exceed the specified range can not be correctly represented and are considered as Overflow values.

For example, a 3-bit Unsigned representation can correctly represent Unsigned Binary values in the range 0 to 2^{3}-1 (0 to 7). Adding 3-bit Unsigned 010 (2) to another 3- bit Unsigned 111 (7) results in 1001 (9) which exceeds the 3-bit unsigned range and is considered to be an Overflow. Similarly, 1011 (-5) and 1100 (-4) values represented in 4- bit 2's complement form when added together result in 10111 (-9) which exceeds the 4- bit 2's complement range of values (2^{4-1}-1) and - (2^{4-1}) (7 to -8) and is considered as an Overflow.

1-5 Floating-point numbers

The floating-point number system, based on scientific notation is capable of representing very large and very small numbers without having to increase the number of bits. Numbers having an integer part and a fraction part are also easily represented using the Floating-Point representation.

Floating point numbers are defined using certain standards. The ANSI/IEEE Standard 754 defines a 32-bit Single-Precision Floating Point format for binary numbers. The 32-bit Single-Precision F.P. format is shown in Figure

S	Exponent	Mantissa

- The single Sign (S) bit represents the sign of the number (0=positive 1=negative)
- The Exponent (E) 8 bits represent the exponent

- The Mantissa 23 bits represent the magnitude of the number

1-5-1 Decimal Number Floating-Point Format

To help understand how numbers are represented in the 32-bit Single Precision Floating Point format. Consider a similar 15 digit Decimal Number format to represent very large and very small decimal numbers. The 15-digit floating point format to represent decimal numbers is shown in Figure 3.2.

S	E	E	M	M	M	M	M	M	M	M	M	M	M	M

- The Sign (S) 1 digit represents the sign of the number (+/–)
- The Exponent (E) 2 digits represent the exponent
- The Mantissa 12 digits represent the magnitude of the number

The number 6918.3125 can be written as 6.9183125 x 10^3.

- 69183125 represents the magnitude of the number (mantissa)
- 3 represents the exponent
- The decimal point is moved to the extreme left of the number (normalized) so that the magnitude is represented by a fraction part.

The number 0.69183125 x 10^4 is represented in decimal floating point. notation as

+	0	4	6	9	1	8	3	1	2	5	0	0	0	0

- Using this 15 digit (including the sign digit) notation the largest number that can be represented is 0.999,999,999,999 x 10^{99}

Representing Negative Exponent Values

The 15-digit decimal floating-point format does not allow negative exponents to be represented. There are two options available

- Increase the Exponent field by one digit to allow for the sign to represent positive and negative exponents. The total number of digits increases to 16.
- Used a Biased Exponent scheme. Instead of writing the exponent value directly add the value 50 to the exponent and write the result in the exponent field. Using this biased scheme the

maximum positive exponent value that can be represented is 49 (49 + 50 = 99). The smallest exponent that can be represented is -50 (-50 + 50 = 0).

After allowing positive and negative exponent values to be represented, the range of positive and negative decimal numbers that can be represented using the decimal floating point notation is 0.999,999,999,999 x 10^{49} to 0.999,999,999,999 x 10^{-50}

Representing Zero and Infinity Values

How should the number Zero and the value Infinity be represented using the 15- digit decimal floating point format?

- The number zero can be represented by setting al the Mantissa digits to 0. The Biased exponent field can be set to any number and the sign field can be set to + or –

- The number infinity can not be represented.

The solution to represent infinity is to set aside a biased exponent value to represent infinity. There are two options available

- Allow numbers having the maximum and minimum exponent values to be 48 and -49 instead of 49 and -50. Thus the Biased exponent values would range between 98 (50 + 48 = 98) and 01 (-49 + 50 = 1). The biased exponent value 00 can be used to represent the number zero whatever the value of the mantissa. The biased exponent value 99 can be used to represent the number infinity what ever the value of mantissa.

- Allow numbers having the maximum and minimum exponent values to be 49 and -48 instead of 49 and -50 and selecting 49 as the biased number. Thus the Biased exponent values would range between 98 (49 + 49 = 98) and 01 (-48 + 49 = 1). The biased exponent value 00 can be used to represent the number zero whatever the value of the mantissa. The biased exponent value 99 can be used to represent the number infinity what ever the value of mantissa. This approach is perhaps better as the range of maximum positive exponent remains 49 and the range of values having a negative exponent have been reduced to -48.

1-5-2 Floating number in 32-bit Single-Precision

Floating Point format

The 32-bit Single Precision Floating Point format represents the Exponent value as a Biased Number, reserving the exponent values 0 and 255 to represent the value zero and infinity respectively. The range of exponent value is from +127 to -126.

The step wise representation of a decimal number 6918.3125 in 32-bit Floating
Point format

- Convert Decimal number into equivalent Binary representation: Binary equivalent of
Decimal number 6918.3125 is 1101100000110.0101
- Normalizing the binary number: 1.1011000001100101 x 2^{12}
- Representing the exponent in Biased 127: exponent is 12 + 127 =139 = 10001011

0	10001011	10110000011001010000000

- The Mantissa is 10110000011001010000000 instead of 110110000011001010000000 as all binary numbers that are normalized always have a leading 1. In the floating point format the leading 1 is not written, however it is taken into account in all calculations. The leading 1 which is not written is known as a hidden 1.

1-5-3 64-bit Double-Precision Floating Point format

The 32-bit Single precision floating point representation can represent largest positive or negative number of the order of 2^{127} and the smallest positive or negative number of the order of 2^{-126}. To represent numbers larger than 2^{127} and numbers smaller than 2^{-126}, 64- bit Double Precision floating point format is used.

The 64-bit Double-Precision format sets aside 11 bits to represent the exponent as Biased-1023 and a mantissa of 52 bits. A single bit, the most significant bit, is set aside for the sign.

1-6 Summary

A number of issues related to computer arithmetic. Our discussion started with an introduction to number representation and radix conversion techniques. We then discussed integer arithmetic and, in particular, we discussed the four main operations, that is, addition, subtraction, multiplication, and division. In each case, we have shown basic architectures and organization. The last topic discussed in this lesson has been floating-point representation.

1-7 Keywords

Decimal Number: The decimal (base ten or occasionally denary) numeral system has ten as its base. It is the most widely used numeral system, perhaps because humans have ten digits over both hands.

NOTES

Binary Number: The binary numeral system, or base-2 number system, is a numeral system that represents numeric values using two symbols, usually 0 and 1.

Binary Arithmetic: Arithmetic in binary is much like arithmetic in other numeral systems. Addition, subtraction, multiplication, and division can be performed on binary numerals.

Floating-point number: The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float": that is, it can be placed anywhere relative to the significant digits of the number.

Radix : In mathematical numeral systems, the base or radix is usually the number of unique digits, including zero, that a positional numeral system uses to represent numbers.

1-8 Exercise

1. Show the results of adding the following pairs of five-bit (i.e. one sign bit and four data bits) two's complement numbers and indicate whether or not overflow occurs for each case:

```
  10110          11110          11111
+ 10111        + 11101        + 01111
-------        -------        -------
```

2. One way to determine that overflow has occurred when adding two numbers is to detect that the result of adding two positive numbers is negative, or that the result of adding two negative numbers is positive. The overflow rules are different for subtraction: there is overflow if the result of subtracting a negative number from a positive number is negative or the result of subtracting a positive number from a negative number is positive. Subtract the numbers shown below and determine whether or not an overflow has occurred. Do not form the two's complement of the subtrahend and add: perform the subtraction bit by bit; showing borrows generated at each position:

```
  0 1 0 1
- 0 1 1 0
__________
```

3. Add the following two's complement and one's complement binary numbers as indicated. For each case, indicate if there is overflow.

```
Two's complement          One's complement
  1 0 1 1.1 0 1             1 0 1 1.1 0 1
+ 0 1 1 1.0 1 1           + 0 1 1 1.0 1 1
_______________           _______________
```

NOTES

2. COMPUTER ARITHMETIC

Structure

Objectives

At the end of lesson you will be able to:

- Discuss about hardware implementation of adder and subtractor
- Discuss about Fixed Point Multiplication and Division
- Discuss about Signed Multiplication and Division
- Discuss about Floating Point Arithmetic

2-1Introduction

In the previous lesson we explored a few ways that numbers can be represented in a digital computer, but we only briefly touched upon arithmetic operations that can be performed on those numbers. In this chapter we cover four basic arithmetic operations: addition, subtraction, multiplication, and division. We begin by describing how these four operations can be performed on fixed point numbers, and continue with a description of how these four operations can be performed on floating point numbers.

2-2 Hardware implementation of Adder and Subtractor

Up until now we have focused on algorithms for addition and subtraction. Now we will take a look at implementations of simple adders and subtractors.

2-2-1 Ripple-Carry Addition and Ripple-Borrow Subtraction

The adder is modeled after the way that we normally perform decimal addition by hand, by summing digits in one column at a time while moving from right to left. In this section, we review the ripple-carry adder, and then take a look at a ripple-borrow subtractor. We then combine the two into a single addition/subtraction unit.

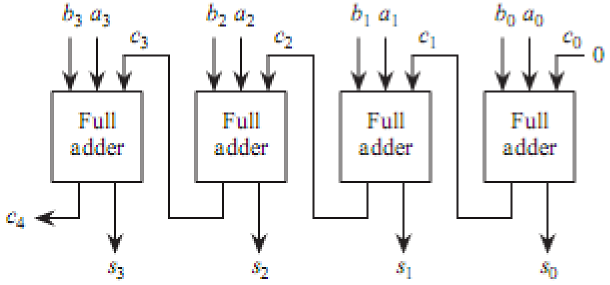


Figure 2-1 4-bit ripple-carry adder

Figure 2-1 shows a 4-bit ripple-carry adder that is developed in Appendix A. Two binary numbers *A* and *B* are added from right to left, creating a sum and a carry at the outputs of each full adder for each bit position.

Four 4-bit ripple-carry adders are cascaded in Figure 2-2 to add two 16-bit numbers. The rightmost full adder has a carry-in of 0. Although the rightmost full adder can be simplified as a result of the carry-in of 0, we will use the more general form and force c_0 to 0 in order to simplify subtraction later on.

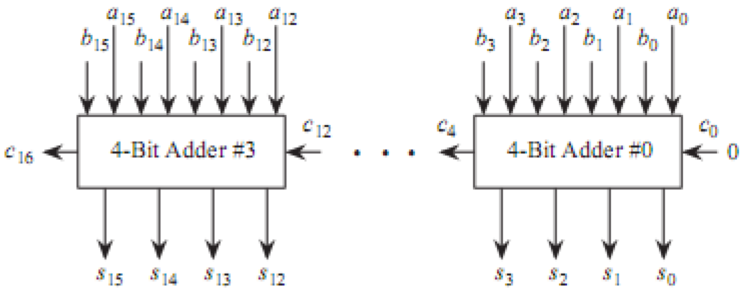


Figure 2-2 A 16-bit adder is made up of a cascade of four 4-bit ripple-carry adders.

Subtraction of binary numbers proceeds in a fashion analogous to addition. We can subtract one number from another by working in a single column at a time, subtracting digits of the subtrahend b_i, from the minuend a_i, as we move from right to left. As in decimal subtraction, if the subtrahend is larger than the minuend or there is a borrow from a previous digit then a borrow must be propagated to the next most significant bit. Figure 2-3 shows the truth table and a “black-box” circuit for subtraction.

a_i	b_i	bor_i	$diff_i$	bor_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

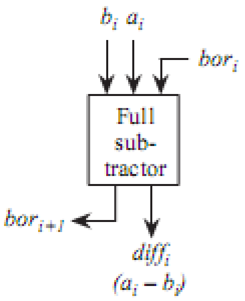


Figure 2-3 Truth table and schematic symbol for a ripple-borrow subtractor.

Full subtractors can be cascaded to form ripple-borrow subtractors in the same manner that full adders are cascaded to form ripple-carry adders. Figure 2--4 illustrates a four-bit ripple-borrow subtractor that is made up of four full subtractors.

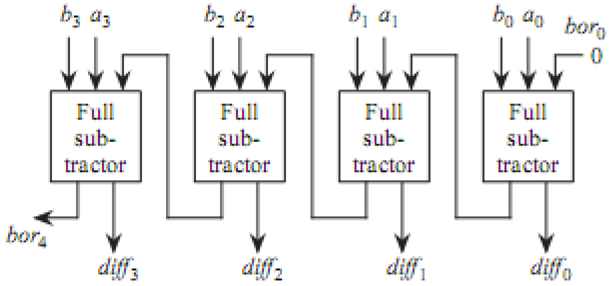


Figure 2-4 Ripple-borrow subtractor.

As discussed above, an alternative method of implementing subtraction is to form the two's complemented negative of the subtrahend and *add* it to the minuend. The circuit that is shown in Figure 2-5 performs both addition and subtraction on four-bit two's complement numbers by allowing the b_i inputs to be complemented when subtraction is desired. An (ADD)' /SUBTRACT control line determines which function is performed. The bar over the ADD symbol indicates the ADD operation is active when the signal is low. That is, if the control line is 0, then the a_i and b_i inputs are passed through to the adder, and the sum is generated at the s_i outputs. If the control line is 1, then the a_i inputs are passed through to the adder, but the b_i inputs are one's complemented by the XOR gates before they are passed on to the adder. In order to form the two's complement negative, we must add 1 to the one's complement negative, which is accomplished by setting the carry - in line (c_0) to 1 with the control input. In this way, we can share the adder hardware among both the adder and the subtractor.

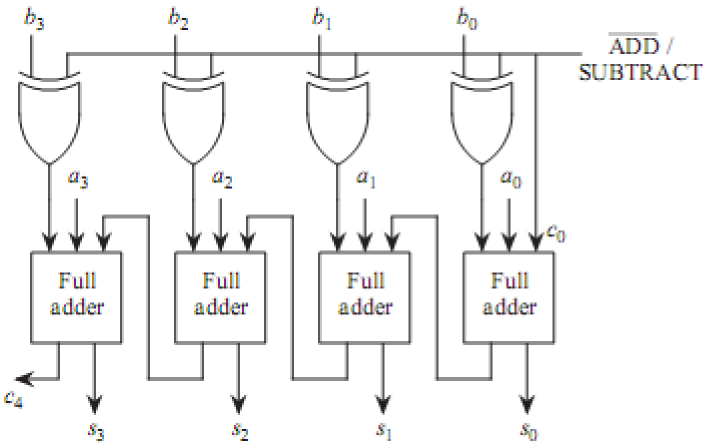


Figure 2-5 Addition / subtraction unit.

2-3 Fixed Point Multiplication and Division

Multiplication and division of fixed point numbers can be accomplished with addition, subtraction, and shift operations. The sections that follow describe methods for performing multiplication and division of fixed point numbers in both unsigned and signed forms using these basic operations. We will first cover unsigned multiplication and division, and then we will cover signed multiplication and division.

2-3-1Unsigned Multiplication

Multiplication of unsigned binary integers is handled similar to the way it is carried out by hand for decimal numbers. Figure 2-7 illustrates the multiplication process for two unsigned binary integers. Each bit of the multiplier determines whether or not the multiplicand, shifted left according to the position of the multiplier bit, is added into the product. When two unsigned n-bit numbers are multiplied, the result can be as large as 2n bits. The example shown in Figure 2-7 multiplication of two four bit operands results in an eight-bit product. When two signed n-bit numbers are multiplied, the result can be as large as only 2(n-1) +1 = (2n-1) bits, because this is equivalent to multiplying two (n-1)-bit unsigned numbers and then introducing the sign bit.

```
              1 1 0 1      (13)10   Multiplicand M
            × 1 0 1 1      (11)10   Multiplier Q
            ---------
              1 1 0 1      ┐
            1 1 0 1        │
          0 0 0 0          │  Partial products
        1 1 0 1            ┘
        ---------------
        1 0 0 0 1 1 1 1    (143)10  Product P
```

Figure 2-7 Multiplication of two unsigned binary integers.

A hardware implementation of integer multiplication can take a similar form to the manual method. Figure 2-8 shows a layout of a multiplication unit for four-bit numbers, in which there is a four-bit adder, a control unit, three four-bit registers, and a one-bit carry register. In order to multiply two numbers, the multiplicand is placed in the M register, the multiplier is placed in the Q register, and the A and C registers are cleared to zero. During multiplication, the rightmost bit of the multiplier determines whether the multiplicand is added into the product at each step. After the multiplicand is added into the product, the multiplier and the A register are simultaneously shifted to the right. This has the effect of shifting the multiplicand to the left (as for the manual process) and exposing the next bit of the multiplier in position q_0.

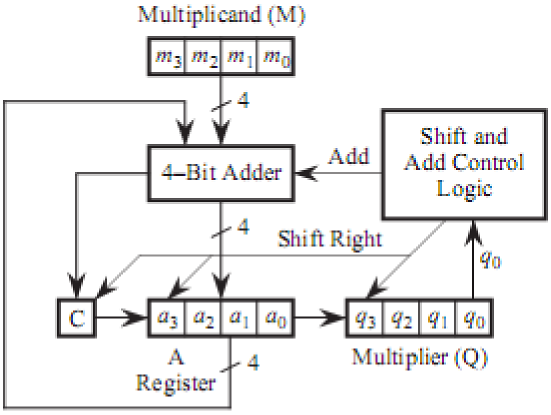
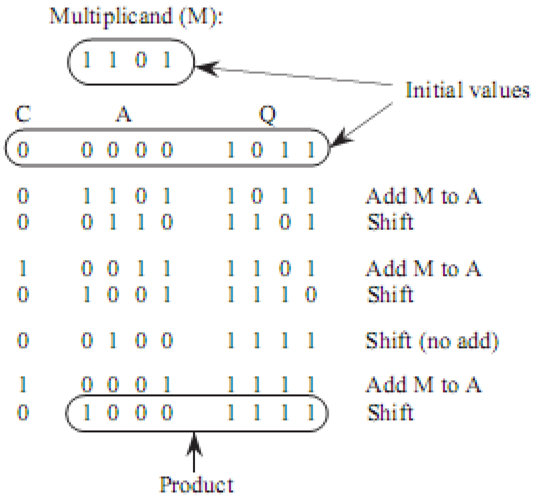


Figure 2-8 A serial multiplier

Figure 2-9 illustrates the multiplication process. Initially, C and A are cleared, and M and Q hold the multiplicand and multiplier, respectively. The rightmost bit of Q is 1, and so the multiplier M is added into the product in the A register. The A and Q registers together make up the eight-bit product, but the A register is where the multiplicand is added. After M is added to A, the A and Q registers are shifted to the right. Since the A and Q registers are linked as a pair to form the eight-bit product, the rightmost bit of A is shifted into the leftmost bit of Q. The rightmost bit of Q is then dropped, C is shifted into the leftmost bit of A, and a 0 is shifted into C.



C	A	Q	
0	0 0 0 0	1 0 1 1	
0	1 1 0 1	1 0 1 1	Add M to A
0	0 1 1 0	1 1 0 1	Shift
1	0 0 1 1	1 1 0 1	Add M to A
0	1 0 0 1	1 1 1 0	Shift
0	0 1 0 0	1 1 1 1	Shift (no add)
1	0 0 0 1	1 1 1 1	Add M to A
0	1 0 0 0	1 1 1 1	Shift

Figure 3-12 An example of multiplication using the serial multiplier

The process continues for as many steps as there are bits in the multiplier. On the second iteration, the rightmost bit of Q is again 1, and so the multiplicand is added to A and the C/A/Q combination is shifted to the right. On the third iteration, the rightmost bit of Q is 0 so M is not added to A, but the C/A/Q combination is still shifted to the right. Finally, on the fourth iteration, the rightmost bit of Q is again 1, and so M is added to A and the C/A/Q combination is shifted to the right. The product is now contained in the A and Q registers, in which A holds the high-order bits and Q holds the low-order bits.

2-3-2 Unsigned Division

In longhand binary division, we must successively attempt to subtract the divisor from the dividend, using the fewest number of bits in the dividend as we can. Figure 2-10 illustrates this point by showing that $(11)_2$ does not “fit” in 0 or 01, but does fit in 011 as indicated by the pattern 001 that starts the quotient.

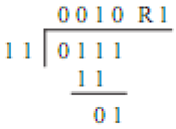


Figure 2-10 Example of base 2 division.

Computer-based division of binary integers can be handled similar to the way that binary integer multiplication is carried out, but with the complication that the only way to tell if the dividend does not “fit” is to actually do the subtraction and test if the remainder is negative. If the remainder is negative then the subtraction must be “backed out” by adding the divisor back in, as described below. In the division algorithm, instead of shifting the product to the right as we did for multiplication, we now shift the quotient to the left, and we subtract instead of adding. When two n-bit unsigned numbers are being divided, the result is no larger than n bits.

Figure 2-11 shows a layout of a division unit for four-bit numbers in which there is a five bit adder, a control unit, a four-bit register for the dividend Q, and two five-bit registers for the divisor M and the remainder A. Five-bit registers are used for A and M, instead of 4-bit registers as we might expect, because an extra bit is needed to indicate the sign of the intermediate result. Although this division method is for unsigned numbers, subtraction is used in the process and negative partial results sometimes arise, which extends the range from −16 through +15, thus there is a need for 5 bits to store intermediate results.

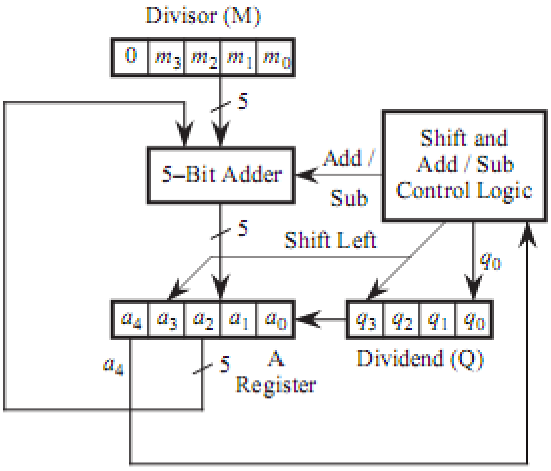


Figure 2-11 A serial divider

In order to divide two four-bit numbers, the dividend is placed in the Q register, the divisor is placed in the M register, and the A register and the high order bit of M are cleared to zero. The leftmost bit of the A register determines whether the divisor is added back into the dividend at each step. This is necessary in order to restore the dividend when the result of subtracting the divisor is negative, as described above. This is referred to as restoring division, because the dividend is restored to its former value when the remainder is negative. When the result is not negative, then the least significant bit of Q is set to 1, which indicates that the divisor “fits” in the dividend at that point.

Figure 2-12 illustrates the division process. Initially, A and the high order bit of M are cleared, and Q and the low order bits of M are loaded with the dividend and divisor, respectively. The A and Q registers are shifted to the left as a pair and the divisor M is subtracted from A. Since the result is negative, the divisor is added back to restore the dividend, and q_0 is cleared to 0. The process repeats by shifting A and Q to the left, and by subtracting M from A. Again, the result is negative, so the dividend is restored and q_0 is cleared to 0. On the third iteration, A and Q are shifted to the left and M is again subtracted from A, but now the result of the subtraction is not negative, so q_0 is set to 1. The process continues for one final iteration, in which A and Q is shifted to the left and M is subtracted from A, which produces a negative result. The dividend is restored and q_0 is cleared to 0. The quotient is now contained in the Q register and the remainder is contained in the A register.

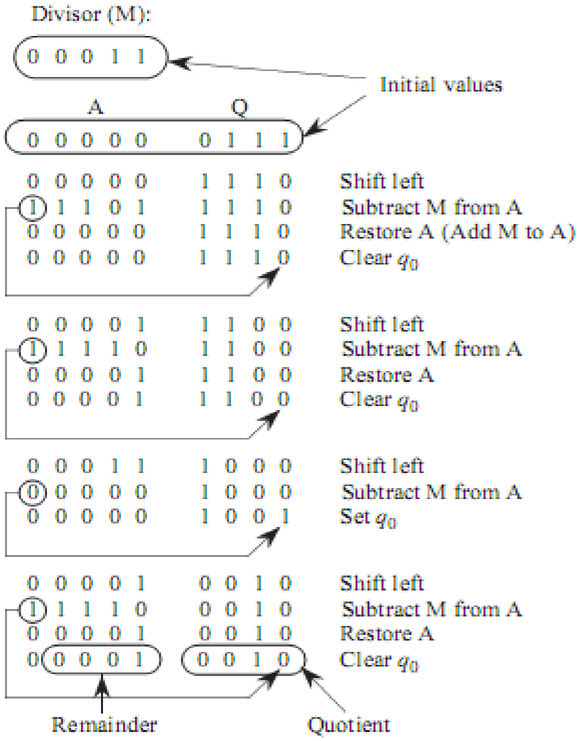

Divisor (M):
0 0 0 1 1
Initial values
A
Q
0 0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 1 0 Shift left
1 1 1 0 1 1 1 1 0 Subtract M from A
0 0 0 0 0 1 1 1 0 Restore A (Add M to A)
0 0 0 0 0 1 1 1 0 Clear q_0
0 0 0 0 1 1 1 0 0 Shift left
1 1 1 1 0 1 1 0 0 Subtract M from A
0 0 0 0 1 1 1 0 0 Restore A
0 0 0 0 1 1 1 0 0 Clear q_0
0 0 0 1 1 1 0 0 0 Shift left
0 0 0 0 0 1 0 0 0 Subtract M from A
0 0 0 0 0 1 0 0 1 Set q_0
0 0 0 0 1 0 0 1 0 Shift left
1 1 1 1 0 0 0 1 0 Subtract M from A
0 0 0 0 1 0 0 1 0 Restore A
0 0 0 0 1 0 0 1 0 Clear q_0
Remainder
Quotient

Figure 2-12 an example of division using the serial divider.

2- 4 Signed Multiplication and Division

If we apply the multiplication and division methods described in the previous sections to signed integers, then we will run into some trouble. Consider multiplying −1 by +1 using four-bit words, as shown in the left side of Figure 2-13 The eight-bit equivalent of +15 is produced instead of −1. What went wrong is that the sign bit did not get extended to the left of the result. This is not a problem for a positive result because the high order bits default to 0, producing the correct sign bit 0.

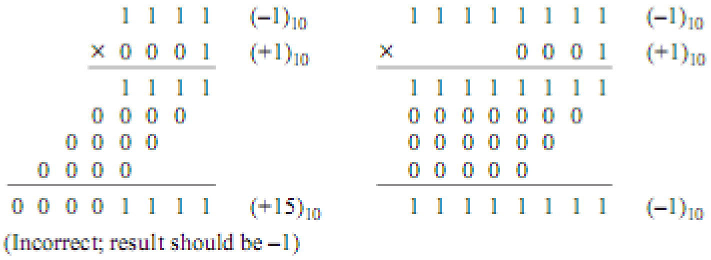


Figure 2-13 Multiplication of signed integers.

A solution is shown in the right side of Figure 2-13, in which each partial product is extended to the width of the result, and only the rightmost eight bits of the result are retained. If both operands are negative, then the signs are extended for both operands, again retaining only the rightmost eight bits of the result.

Signed division is more difficult. We will not explore the methods here, but as a general technique, we can convert the operands into their positive forms, perform the division, and then convert the result into its true signed form as a final step.

2- 5 Floating Point Arithmetic

Arithmetic operations on floating point numbers can be carried out using the fixed point arithmetic operations described in the previous sections, with attention given to maintaining aspects of the floating point representation. In the sections that follow, we explore floating point arithmetic in base 2 and base 10, keeping the requirements of the floating point representation in mind.

2- 5 - 1 Floating Point Addition subtractor

Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands. As in ordinary base 10 arithmetic using scientific notation, the exponents of the operands must be made equal for addition and subtraction. The fractions are then added or subtracted as appropriate, and the result is normalized.

This process of adjusting the fractional part and also rounding the result can lead to a loss of precision in the result. Consider the unsigned floating point addition ($.101 \times 2^3 + .111 \times 2^4$) in which the fractions have three significant digits. We start by adjusting the smaller exponent to be equal to the larger exponent and adjusting the fraction accordingly. Thus we have $.101 \times 2^3 = .010 \times 2^4$, losing $.001 \times 2^3$ of precision in the process. The resulting sum is

$$(.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5,$$

and rounding to three significant digits, $.100 \times 2^5$, and we have lost another 0.001×2^4 in the rounding process.

Why do floating point numbers have such complicated formats?

We may wonder why floating point numbers have such a complicated structure, with the mantissa being stored in signed magnitude representation, the exponent stored in excess notation, and the sign bit separated from the rest of the magnitude by the intervening exponent field. There is a simple explanation for this structure. Consider the complexity of performing floating point arithmetic in a computer. Before any arithmetic can be done, the number must be unpacked from the form it takes in storage. The exponent and mantissa must be extracted from the packed bit pattern before an

NOTES

arithmetic operation can be performed; after the arithmetic operation(s) are performed, the result must be renormalized and rounded, and then the bit patterns are re-packed into the requisite format.

The virtue of a floating point format that contains a sign bit followed by an exponent in excess notation, followed by the magnitude of the mantissa, is that two floating point numbers can be compared for >, <, and = without unpacking. The sign bit is most important in such a comparison, and it appropriately is the MSB in the floating point format. Next most important in comparing two numbers is the exponent, since a change of ± 1 in the exponent changes the value by a factor of 2 (for a base 2 format), whereas a change in even the MSB of the fractional part will change the value of the floating point number by less than that.

In order to account for the sign bit, the signed magnitude fractions are represented as integers and are converted into two’s complement form. After the addition or subtraction operation takes place in two’s complement, there may be a need to normalize the result and adjust the sign bit. The result is then converted back to signed magnitude form.

2-5-2 Floating point multiplication and division

Floating point multiplication and division are performed in a manner similar to floating point addition and subtraction, except that the sign, exponent, and fraction of the result can be computed separately. If the operands have the same sign, then the sign of the result is positive. Unlike signs produce a negative result. The exponent of the result before normalization is obtained by adding the exponents of the source operands for multiplication, or by subtracting the divisor exponent from the dividend exponent for division. The fractions are multiplied or divided according to the operation, followed by normalization.

Consider using three-bit fractions in performing the base 2 computation: $(+.101\times 2^{2}) \times (-.110 \times 2^{-3})$. The source operand signs differ, which means that the result will have a negative sign. We add exponents for multiplication, and so the exponent of the result is $2 + -3 = -1$. We multiply the fractions, which produces the product .01111. Normalizing the product and retaining only three bits in the fraction produces $-.111 \times 2^{-2}$. Now consider using three-bit fractions in performing the base 2 computation: $(+.110 \times 2^{5}) / (+.100 \times 2^{4})$. The source operand signs are the same, which means that the result will have a positive sign. We subtract exponents for division, and so the exponent of the result is $5 – 4 = 1$. We divide fractions, which can be done in a number of ways. If we treat the fractions as unsigned integers, then we will have 110/100 = 1 with a remainder of 10. What we really want is a contiguous set of bits representing the fraction instead of a separate result and remainder, and so we can scale the dividend to the left by two positions, producing the result: 11000/100 = 110. We then scale the result to the right by two positions to restore the original scale factor, producing 1.1 Putting it all together, the result of dividing $(+.110 \times 2^{5})$ by $(+.100 \times 2^{4})$ produces $(+1.10 \times 2^{1})$. After normalization, the final result is $(+.110 \times 2^{2})$.

2-6 Summary

In this unit we have discuss about the hardware implementation of adder and subtractor by different techniques. We also discuss multiplication and division in fixed and signed numbers. Finally we focus on floating point arithmetic operations.

2-7 Keywords

Ripple-Carry: It is possible to create a logical circuit using multiple full adders to add *N*-bit numbers. Each full adder inputs a C_{in}, which is the C_{out} of the previous adder. This kind of adder is a *ripple carry adder*, since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder.

Serial adder: The serial binary adder is a digital circuit that performs binary addition bit by bit. The serial full adder has three single bit inputs for the numbers to be added and the carry in.

Floating point numbers: The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float": that is, it can be placed anywhere relative to the significant digits of the number.

2-8 Exercise

1. Show the results of adding the following pairs of five-bit (i.e. one sign bit and four data bits) two's complement numbers and indicate whether or not overflow occurs for each case:

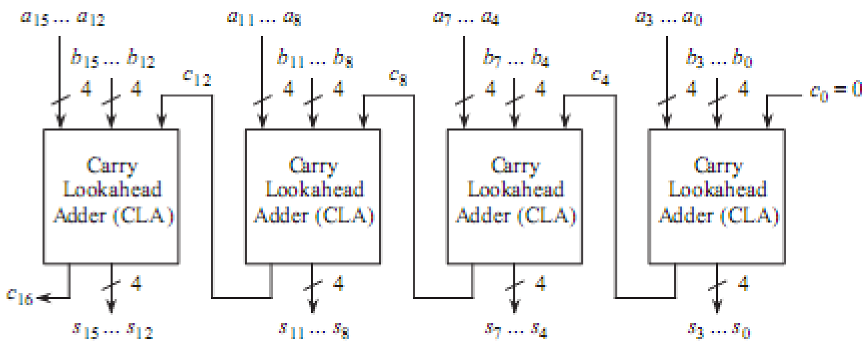
$$\begin{array}{r} 1\,0\,1\,1\,0 \\ +\,1\,0\,1\,1\,1 \\ \hline \end{array} \qquad \begin{array}{r} 1\,1\,1\,1\,0 \\ +\,1\,1\,1\,0\,1 \\ \hline \end{array} \qquad \begin{array}{r} 1\,1\,1\,1\,1 \\ +\,0\,1\,1\,1\,1 \\ \hline \end{array}$$

2. Show the process of serial unsigned multiplication for 1010 (multiplicand) multiplied by 0101 (multiplier). Use the form shown in Figure 2-9.

3. Show the process of serial unsigned division for 1010 divided by 0101. Use the form shown in Figure 2-12.

4. Show the process of serial unsigned division for 1010 divided by 0100, but instead of generating a remainder, compute the fraction by continuing the process. That is, the result should be 10.1_2.

5. The 16-bit adder shown below uses a ripple carry among four-bit carry lookahead adders.



(a) What is the longest gate delay through this adder?

(b) What is the shortest gate delay through this adder, from any input to any output?

(c) What is the gate delay for s_{12}?

NOTES

3. HIGH PERFORMANCE ARITHMETIC

Structure

Objectives

At the end of this lesson you should be able to:

- Describe the high performance addition, multiplication, and division
- Discuss about Residue Arithmetic

3-1 Introduction

For many applications, the speed of arithmetic operations is the bottleneck to performance. Most supercomputers, such as the Cray, the Tera, and the Intel Hypercube are considered “super” because they excel at performing fixed and floating point arithmetic. In this section we discuss a number of ways to improve the speed of addition, subtraction, multiplication, and division.

3-2 High Performance Addition

The ripple-carry adder that we reviewed in lesson 2 may introduce too much delay into a system. The longest path through the adder is from the inputs of the least significant full adder to the outputs of the most significant full adder. The process of summing the inputs at each bit position is relatively fast (a small two-level circuit suffices) but the carry propagation takes a long time to work its way through the circuit. In fact, the propagation time is proportional to the number of bits in the operands. This is unfortunate, since more significant in an addition translates to more time to perform the addition. In this section, we look at a method of speeding the carry propagation in what is known as a carry lookahead adder.

In reduced Boolean expressions for the sum (s_i) and carry outputs (c_{i+1}) of a full adder are created. These expressions are repeated below, with subscripts added to denote the relative position of a full adder in a ripple-carry adder:

$$s_i = \overline{a}_i\overline{b}_ic_i + \overline{a}_ib_i\overline{c}_i + a_i\overline{b}_i\overline{c}_i + a_ib_ic_i$$

$$c_{i+1} = b_ic_i + a_ic_i + a_ib_i$$

We can factor the second equation and obtain:

$$c_{i+1} = a_ib_i + (a_i + b_i)c_i$$

which can be rewritten as:

$$c_{i+1} = G_i + P_ic_i$$

where: $G_i = a_ib_i$ and $P_i = a_i + b_i$

The G_i and P_i terms are referred to as generate and propagate functions, respectively, for the effect they have on the carry. When $G_i = 1$, a carry is generated at stage i. When $P_i = 1$, then a carry is propagated through stage i if either a_i or b_i is a 1. The G_i and P_i terms can be created in one level of logic since they only depend on an AND or an OR of the input variables, respectively.

The carries again take the most time. The carry c_1 out of stage 0 is $G_0 + P_0c_0$, and since $c_0 = 0$ for addition, we can rewrite this as $c_1 = G_0$. The carry c_2 out of stage 1 is $G_1 + P_1c_1$, and since $c_1 = G_0$, we can rewrite this as: $c_2 = G_1 + P_1G_0$. The carry c_3 out of stage 2 is $G_2 + P_2c_2$, and since $c_2 = G_1 + P_1G_0$, we can rewrite this as: $c_3 = G_2 + P_2G_1 + P_2P_1G_0$. Continuing one more time for a four-bit adder, the carry out of stage 3 is $G_3 + P_3c_3$, and since $c_3 = G_2 + P_2G_1 + P_2P_1G_0$, we can rewrite this as: $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$. We can now create a four-bit carry lookahead adder as shown in Figure 3-1. We still have the delay through the full adders as before, but now the carry chain is broken into independent pieces that require one gate delay for G_i and P_i and two more gate delays to generate c_{i+1}. Thus, a depth of three gate delays is added, but the ripple-carry chain is removed. If we assume that each full adder introduces a gate delay of two, then a four-bit carry lookahead adder will have a maximum gate delay of five, whereas a four-bit ripple-carry adder will have a maximum gate delay of eight. The difference between the two approaches is more pronounced for wider operands. This process is limited to about eight bits of carry-lookahead, because of gate fan-in limitations. For additions of numbers having more than eight bits, the carry-lookahead circuits can be cascaded to compute the carry in and carry out of each carry-lookahead unit. (See the EXAMPLE at the end of the lesson.)

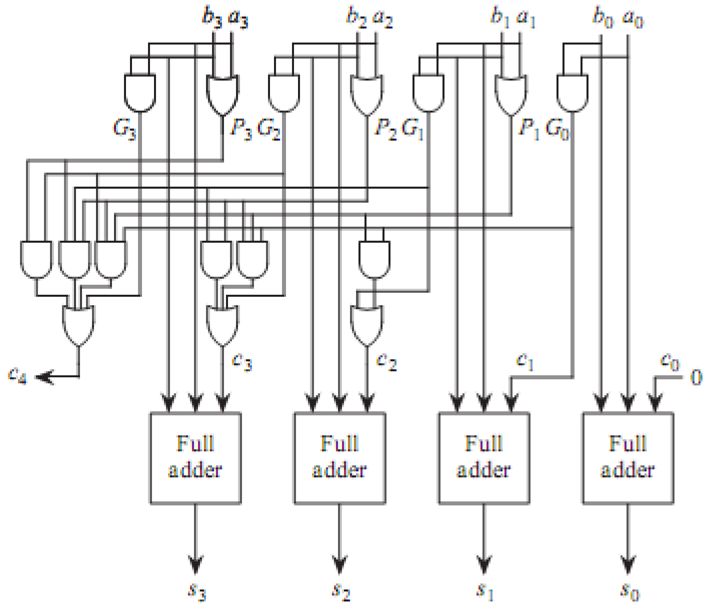


Figure 3-1 Carry-lookahead adder.

3-3 High Performance Multiplication

A number of methods exist for speeding the process of multiplication. Two methods are described in the sections below. The first approach gains performance by skipping over blocks of 1's, which eliminates addition steps. A parallel multiplier is described next, in which a cross product among all pairs of multiplier and multiplicand bits is formed. The result of the cross product is summed by rows to produce the final product.

3-3-1 Booth Algorithm

The Booth algorithm treats positive and negative numbers uniformly. It operates on the fact that strings of 0's or 1's in the multiplier require no additions just shifting. Additions or subtractions take place at the boundaries of the strings, where transitions take place from 0 to 1 or from 1 to 0. A string of 1's in the multiplier from bit positions with weights 2^u to 2^v can be treated as $2^{u+1} – 2^v$. For example, if the multiplier is 001110 $(+14)_{10}$, then u = 3 and v = 1, so $2^4 – 2^1 = 14$.

In a hardware implementation, the multiplier is scanned from right to left. The first transition is observed going from 0 to 1, and so 2^1 is subtracted from the initial value (0). On the next transition, from 1 to 0, 2^4 is added, which results in +14. A 0 is considered to be appended to the right side of the multiplier in order to defined the situation in which a 1 is in the rightmost digit of the multiplier. If the multiplier is recoded according to the Booth algorithm, then fewer steps may be needed in the multiplication process. Consider the multiplication example shown in Figure 3-2. The multiplier $(14)_{10}$ contains three 1's, which means that three addition operations are required for the shift/add multiplication procedure that is described in lesson 2. The Booth recoded multiplier is obtained by scanning the original multiplier from right to left, and placing a −1 in the position where the first 1 in a string is encountered, and placing a +1 in the position where the next 0 is seen. The multiplier 001110 thus becomes 0 +1 0 0 −1 0. The Booth recoded multiplier contains just two nonzero digits: +1 and −1, which means that only one addition operation and one subtraction operation are needed, and so a savings is realized for this example.

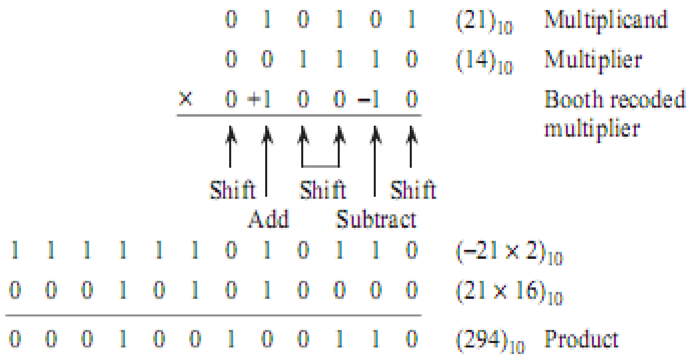

0 1 0 1 0 1 $(21)_{10}$ Multiplicand
0 0 1 1 1 0 $(14)_{10}$ Multiplier
× 0 +1 0 0 −1 0 Booth recoded multiplier
Shift
Add
Shift
Subtract
Shift
1 1 1 1 1 1 0 1 0 1 1 0 $(-21 \times 2)_{10}$
0 0 0 1 0 1 0 1 0 0 0 0 $(21 \times 16)_{10}$
0 0 0 1 0 0 1 0 0 1 1 0 $(294)_{10}$ Product

Figure 3-2 Multiplication of signed integers.

A savings is not always realized, however, and in some cases the Booth algorithm may cause more operations to take place than if it is not used at all. Consider the example shown in Figure 3-3, in which the multiplier consists of alternating 1's and 0's. This is the same example shown in Figure 3-2 but with the multiplicand and multiplier swapped. Without Booth recoding of the multiplier, three addition operations are required for the three 1's in the multiplier. The Booth recoded multiplier, however, requires six addition and subtraction operations, which is clearly worse. We improve on this in the next section.

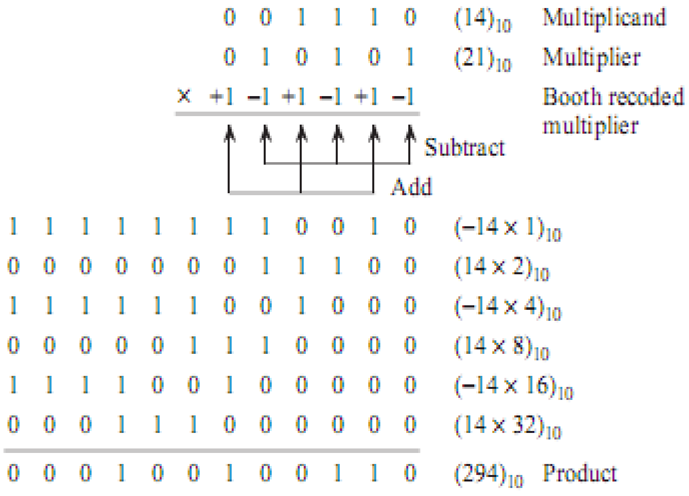
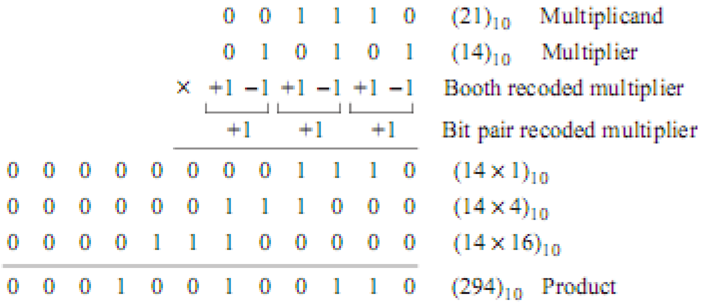

0 0 1 1 1 0 $(14)_{10}$ Multiplicand
0 1 0 1 0 1 $(21)_{10}$ Multiplier
× +1 −1 +1 −1 +1 −1 Booth recoded multiplier
Subtract
Add
1 1 1 1 1 1 1 1 0 0 1 0 $(-14 \times 1)_{10}$
0 0 0 0 0 0 0 1 1 1 0 0 $(14 \times 2)_{10}$
1 1 1 1 1 1 0 0 1 0 0 0 $(-14 \times 4)_{10}$
0 0 0 0 0 1 1 1 0 0 0 0 $(14 \times 8)_{10}$
1 1 1 1 0 0 1 0 0 0 0 0 $(-14 \times 16)_{10}$
0 0 0 1 1 1 0 0 0 0 0 0 $(14 \times 32)_{10}$
0 0 0 1 0 0 1 0 0 1 1 0 $(294)_{10}$ Product

Figure 3-3 A worst case Booth recoded multiplication example

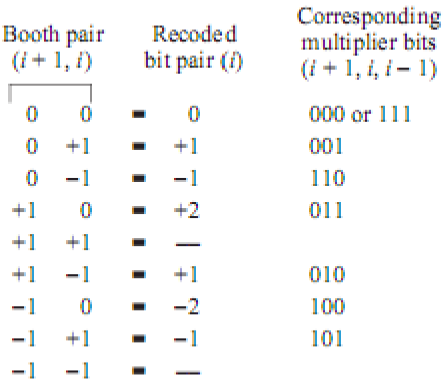
3-3-2 Modified Booth Algorithm

One solution to this problem is to group the recoded multiplier bits in pairs, known as bit pair recoding, which is also known as the modified Booth algorithm. Grouping bit pairs from right to left produces three “+1, −1” pairs as shown in Figure 3-4. Since the +1 term is to the left of the −1 term, it has a In a similar manner, the pair −1, +1 is equivalent to −2 + 1 = −1. The pairs +1, +1 and −1,−1 cannot occur. There are a total of seven pairs that can occur, which are shown in Figure 3-5. For each case, the value of the recoded bit pair is multiplied by the multiplicand and is added to the product. In an implementation of bit pair recoding, the Booth recoding and bit pair recoding steps are collapsed into a single step, by observing three multiplier bits at a time, as shown in the corresponding multiplier bit table.



						0	0	1	1	1	0	$(21)_{10}$	Multiplicand
						0	1	0	1	0	1	$(14)_{10}$	Multiplier
					×	+1	−1	+1	−1	+1	−1	Booth recoded multiplier	
							+1		+1		+1	Bit pair recoded multiplier	
0	0	0	0	0	0	0	0	1	1	1	0	$(14 \times 1)_{10}$	
0	0	0	0	0	0	1	1	1	0	0	0	$(14 \times 4)_{10}$	
0	0	0	0	1	1	1	0	0	0	0	0	$(14 \times 16)_{10}$	
0	0	0	1	0	0	1	0	0	1	1	0	$(294)_{10}$	Product

Figure 3-4 Multiplication with bit-pair recoding of the multiplier.



Booth pair $(i+1, i)$			Recoded bit pair (i)	Corresponding multiplier bits $(i+1, i, i-1)$
0	0	■	0	000 or 111
0	+1	■	+1	001
0	−1	■	−1	110
+1	0	■	+2	011
+1	+1	■	—	
+1	−1	■	+1	010
−1	0	■	−2	100
−1	+1	■	−1	101
−1	−1	■	—	

Figure 3-5 Recoded bit pairs.

The process of bit pair recoding of a multiplier guarantees that in the worst case, only w/2 additions (or subtractions) will take place for a w-bit multiplier.

3-3-3 Array Multipliers

The serial method we used for multiplying two unsigned integers in lesson 2(2-2-1) requires only a small amount of hardware, but the time required to multiply two numbers of length w grows as w^2. We can speed the multiplication process so that it completes in just 2w steps by implementing the manual process shown in Figure 2-7 in parallel. The general idea is to form a one-bit product between each multiplier bit and each multiplicand bit, and then sum each row of partial product elements from the top to the bottom in systolic (row by row) fashion. The structure of a systolic array multiplier is shown in Figure 3-6. A partial product (PP) element is shown at the bottom of the figure. A multiplicand bit (m_i) and a multiplier bit (q_j) are multiplied by the AND gate, which forms a partial product at position (i,j) in the array. This partial product is added with the partial product from the previous stage (b_j) and any carry that is generated in the previous stage (a_j). The result has a width of 2w, and appears at the bottom of the array (the high order w bits) and at the right of the array (the low order w bits).

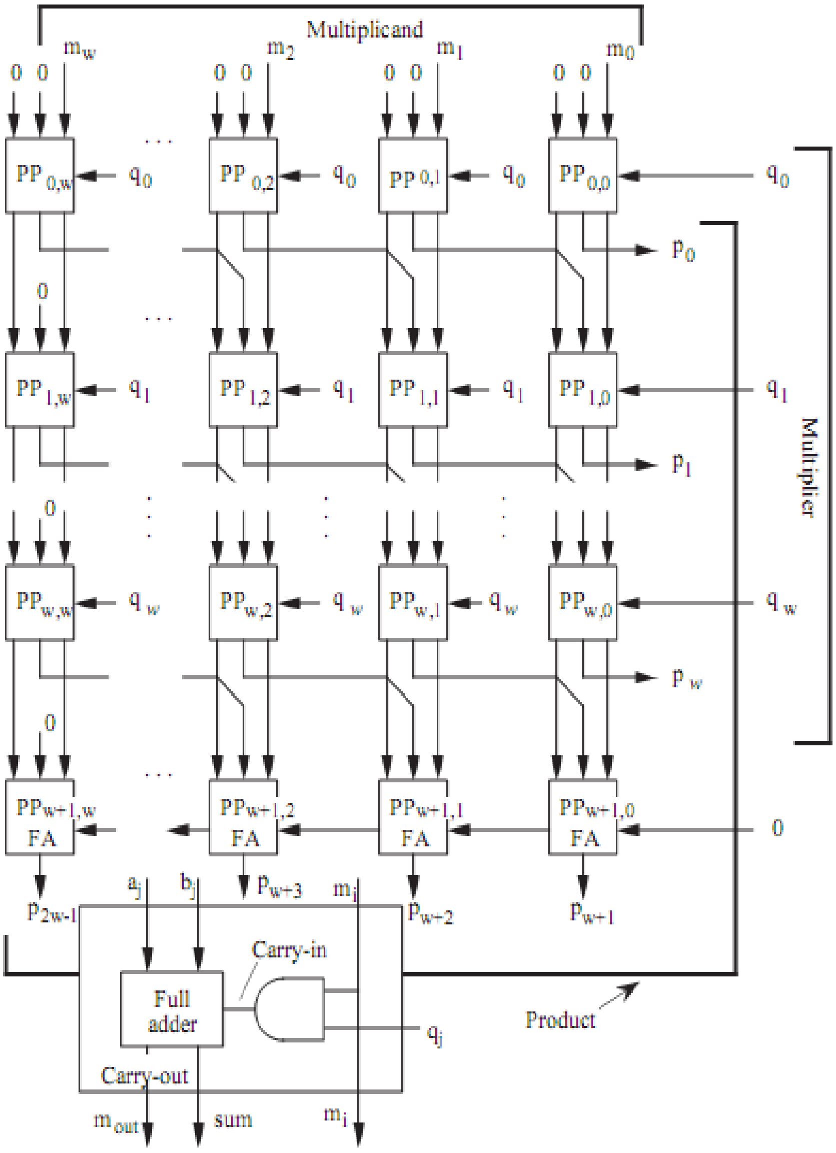


Figure 3-6 Parallel pipelined array multiplier.

3-4 High Performance Division

We can extend the unsigned integer division technique of lesson to produce a fractional result in computing a/b. The general idea is to scale a and b to look like integers, perform the division process, and then scale the quotient to correspond to the actual result of dividing a by b.

A faster method of division makes use of a lookup table and iteration. An iterative method of finding a root of a polynomial is called Newton's iteration, which is illustrated in Figure 3-7. The goal is to find where the function f(x) crosses the x axis by starting with a guess x_i and then using the error between f (x_i) and zero to refine the guess.

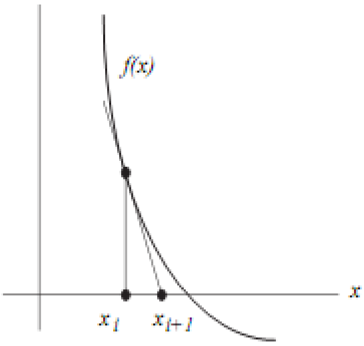


Figure 3-7 Newton's iteration for zero finding.

The tangent line at $f(x_i)$ can be represented by the equation:

$$y - f(x_i) = f'(x_i)(x - x_i).$$

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The tangent line crosses the x axis at:

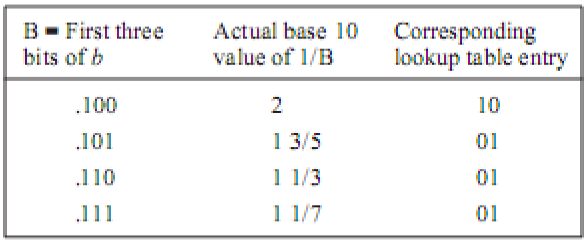
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

The process repeats while f(x) approaches zero.

The number of bits of precision doubles on each iteration, and so if we are looking to obtain 32 bits of precision and we start with a single bit of precision, then five iterations are required to reach our target precision. The problem now is to cast division in the form of finding a zero for f(x). Consider the function 1/x − b which has a zero at 1/b. If we start with b, then we can compute 1/b by iteratively applying Newton's method. Since f '(x) = $-1/x^2$, we now have:

$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

Thus, we only need to perform multiplication and subtraction in order to perform division. Further, if our initial guess for x0 is good enough, then we may only need to perform the iteration a few times. Before using this method on an example, we need to consider how we will obtain our initial guess. If we are working with normalized fractions, then it is relatively easy to make use of a lookup table for the first few digits. Consider computing 1/.101101 using a 16-bit normalized base 2 fraction in which the leading 1 is not hidden. The first three bits for any binary fraction will be one of the patterns: .100, .101, .110, or .111. These fractions correspond to the base 10 numbers 1/2, 5/8, 3/4, and 7/8, respectively. The reciprocals of these numbers are 2, 8/5, 4/3, and 8/7, respectively. We can store the binary equivalents in a lookup table, and then retrieve x0 based on the first three bits of b. The leading 1 in the fraction does not contribute to the precision, and so the leading three bits of the fraction only provide two bits of precision. Thus, the lookup table only needs two bits for each entry, as shown in Figure 3-8



B = First three bits of *b*	Actual base 10 value of 1/B	Corresponding lookup table entry
.100	2	10
.101	1 3/5	01
.110	1 1/3	01
.111	1 1/7	01

Figure 3-8 A three-bit lookup table

Figure 3-8 A three-bit lookup table for computing x_0 now consider computing 1/.1011011 using this floating point representation. We start by finding x_0 using the table shown in Figure 3-8. The first three bits of the fraction b are 101, which corresponds to x_0 = 01. We compute $x_1 = x_0(2 - x_0b)$ and obtain, in unsigned base 2 arithmetic: $x_1 = 01(10 - (01)(.1011011)) = 1.0100101$. Our two bits of precision have now become four bits of precision. For this example, we will retain as much intermediate precision as we can. In general, we only need to retain at most 2p bits of intermediate precision for a p-bit result. We iterate again, obtaining eight bits of precision:

$$x_2=x_1(2 - x_1b) = 1.0100101(10 - (1.0100101)(.1011011))$$

$$= 1.011001011001001011101.$$

We iterate again, obtaining our target 16 bits of precision:

$$x_3 = x_2(2 - x_2b)$$

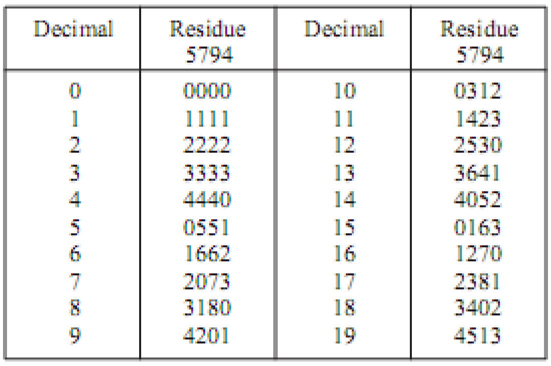
$$=(1.011001011001001011101)(2-(1.011001011001001011101)(.1011011))$$

$$= 1.011010000001001 = (1.40652466)_{10}.$$

The precise value is $(1.40659341)_{10}$, but our 16-bit value is as close to the precise value as it can be.

3-5 Residue Arithmetic

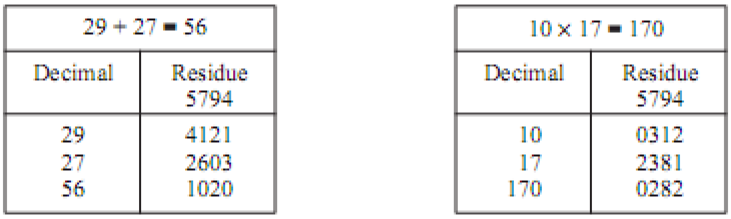
Addition, subtraction, and multiplication can all be performed in a single, carry-less step using residue arithmetic. The residue number system is based on relatively prime integers called module. The residue of an integer with respect to a particular modulus is the least positive integer remainder of the division of the integer by the modulus. A set of possible moduli are 5, 7, 9, and 4. With these moduli, 5 × 7 × 9 × 4 = 1260 integers can be uniquely represented. A table showing the representation of the first twenty decimal integers using moduli 5, 7, 9, and 4 is shown in Figure 3-9.



Decimal	Residue 5794	Decimal	Residue 5794
0	0000	10	0312
1	1111	11	1423
2	2222	12	2530
3	3333	13	3641
4	4440	14	4052
5	0551	15	0163
6	1662	16	1270
7	2073	17	2381
8	3180	18	3402
9	4201	19	4513

Figure 3-9 First twenty decimal integers in the residue number system for the given moduli

Addition and multiplication in the residue number system result in valid residue numbers provided the size of the chosen number space is large enough to contain the results. Subtraction requires each residue digit of the subtrahend to be complemented with respect to its modulus before performing addition. Addition and multiplication examples are shown in Figure 3-10. For these examples, the moduli used are 5, 7, 9, and 4. Addition is performed in parallel for each column, with no carry propagation. Multiplication is also performed in parallel for each column, independent of the other columns.



29 + 27 = 56	
Decimal	Residue 5794
29	4121
27	2603
56	1020

10 × 17 = 170	
Decimal	Residue 5794
10	0312
17	2381
170	0282

Figure 3-10 Examples of addition and multiplication in the residue number system.

Although residue arithmetic operations can be very fast, there are a number of disadvantages to the system. Division and sign detection are difficult, and a representation for fractions is also difficult. Conversions between the residue number system and weighted number systems are complex, and often require involved methods such as the Chinese remainder theorem. The conversion problem is important because the residue number system is not very useful without being translated to a weighted number system so that magnitude comparisons can be made. However, for integer applications in which the time spent in addition, subtraction, and multiplication outweighs the time spent in division, conversion, etc., the residue number system may be a practical approach. An important application area is matrix-vector multiplication, which is used extensively in signal processing.

3-6 Summary

Performance can be improved by skipping over 1's in the Booth and bit-pair recoding techniques. An alternative method of improving performance is to use carryless addition, such as in residue arithmetic. Although carryless addition may be the fastest approach in terms of time complexity and circuit complexity, the more common weighted position codes are normally used in practice in order to simplify comparisons and represent fractions.

3-7 Keywords

Booth algorithm: Booth's algorithm involves repeatedly adding one of two predetermined values *A* and *S* to a product *P*, then performing a rightward arithmetic shift on *P*. Let x and y be the multiplicand and multiplier, respectively; and let *x* and *y* represent the number of bits in x and y.

High performance arithmetic: The goal is to improve the speed of arithmetic operation.

Residue arithmetic: The residue number system is based on relatively prime integers called module. The residue of an integer with respect to a particular modulus is the least positive integer remainder of the division of the integer by the modulus.

3-8 Exercise

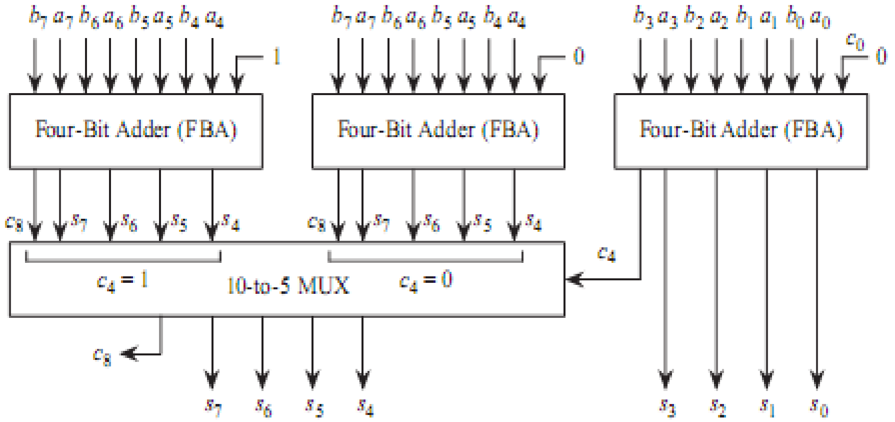
1. Use the Booth algorithm (not bit pair recoding) to multiply 010011 (multiplicand) by 011011 (multiplier).

2. Use bit pair recoding to multiply 010011 (multiplicand) by 011011 (multiplier).

3. Compute the maximum gate delay through a 32-bit carry lookahead adder.

4. In a carry-select adder a carry is propagated from one adder stage to the next, similar to but not exactly the same as a carry lookahead adder. As with many other adders, the carry out of a carry-select adder stage is

either 0 or 1. In a carry-select adder, two sums are computed in parallel for each adder stage: one sum assumes a carry-in of 0, and the other sum assumes a carry-in of 1. The actual carry-in selects which of the two sums to use (with a MUX, for example). The basic layout is shown below for an eight-bit carry-select adder:


$b_7\ a_7\ b_6\ a_6\ b_5\ a_5\ b_4\ a_4$
1
$b_7\ a_7\ b_6\ a_6\ b_5\ a_5\ b_4\ a_4$
0
$b_3\ a_3\ b_2\ a_2\ b_1\ a_1\ b_0\ a_0$
c_0
0
Four-Bit Adder (FBA)
Four-Bit Adder (FBA)
Four-Bit Adder (FBA)
c_8 s_7 s_6 s_5 s_4
c_8 s_7 s_6 s_5 s_4
c_4
$c_4 = 1$
10-to-5 MUX
$c_4 = 0$
c_8
s_7 s_6 s_5 s_4
s_3 s_2 s_1 s_0

Assume that each four-bit adder (FBA) unit uses carry lookahead internally. Compare the number of gate delays needed to add two eight-bit numbers using FBA units in a carry-select configuration vs. using FBA units in which the carry is rippled from one FBA to the next.

(a) Draw a diagram of a functionally equivalent eight-bit carry lookahead configuration using the FBAs shown above.

(b) Show the number of gate delays for each adder configuration, by both the 8-bit carry-select adder shown above and the adder designed in part (a) above.

5. The path with the maximum gate delay through the array multiplier shown in Figure 3-6 starts in the top right PP element then travels to the bottom row, then across to the left. The maximum gate delay through a PP element is three. How many gate delays are on the maximum gate delay path through an array multiplier that produces a p-bit result?

6. Given multiplication units that each produce a 16-bit unsigned product on two unsigned 8-bit inputs, and 16-bit adders that produce a 16-bit sum and a carry-out on two 16-bit inputs and a carry-in, connect these units so that the overall unit multiplies 16-bit unsigned numbers, producing a 32-bit result.

7. Using Newton's iteration for division, we would like to obtain 32 bits of precision. If we use a lookup table that provides eight bits of precision for the initial guess, how much iteration need to be applied?

.

NOTES

4. MEMORY ORGANIZATION

Structure

Objectives

At the end of this lesson you should be able to:

- Discuss about Memory Hierarchy
- Discuss about Main Memory
- Define the Auxiliary Memory
- Define the Associative Memory

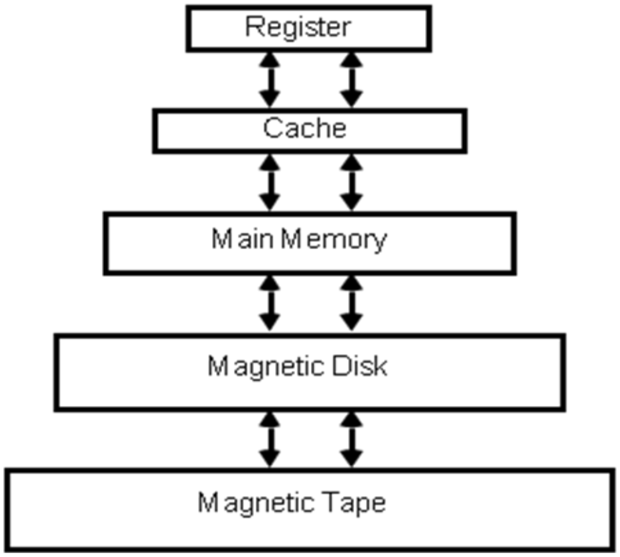
4-1 Memory Hierarchy

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software. Not all accumulated information is needed by the

NOTES

processor at the same time. Therefore, it is more economical to use low-cost storage devices to serve as a backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the *main memory.* Devices that provide backup storage are called *auxiliary memory.* The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high-speed processing logic. Figure 4-1 illustrates the components in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.



NOTES

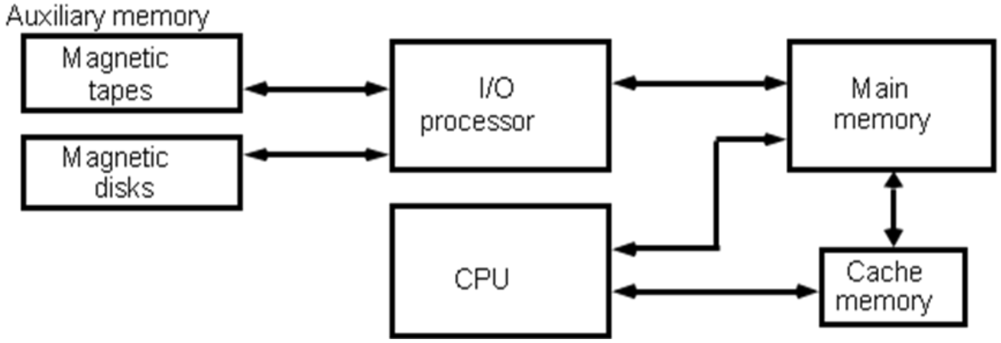


Figure 4-1 Memory hierarchy in a computer system

A special very-high-speed memory called a *cache* is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations. By making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of me memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared t0 main memory. The cache memory is very small, relatively expensive and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100 ns, while main memory access time may be 700 ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called *multiprogramming,* refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to utilize the CPU.

With multiprogramming the need arises for running partial programs, for varying the amount of main memory in use by a given program, and for moving programs around the memory hierarchy. Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU. Thus one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the How of information between auxiliary memory and main memory is called the *memory management system.*

4-2 Main Memory

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, *static* and *dynamic.* The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charges on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic

RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles. One of the major applications of the static RAM is in implementing the cache memories. The dynamic RAMs are used for implementing the main memory. Most of the desktop personnel computer systems are dynamic RAMs with improved performance characteristics such as multibank DRAM, extended dataout DRAM, synchronous DRAM, and Direct RAM bus DRAM.

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

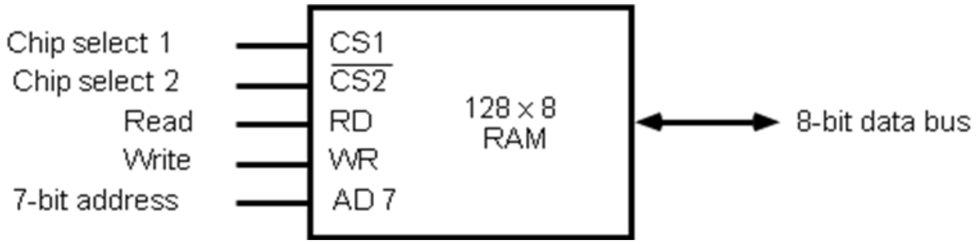
Among other things, the ROM portion of main memory is needed for storing an initial program called a *bootstrap loader.* The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 X 8 memory constructed with 128 X 8 RAM chips and 512 X 8 ROM chips.

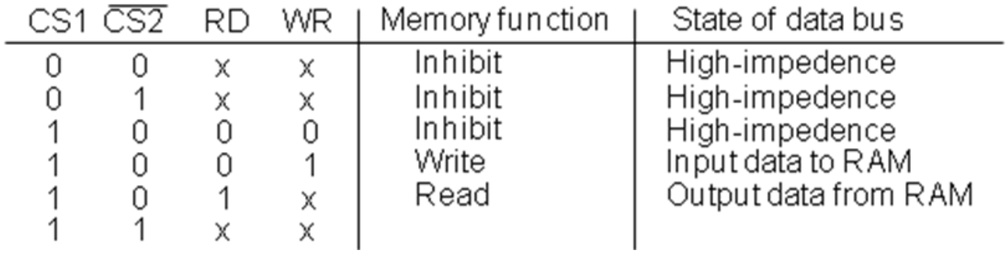
4-2 -1 RAM and ROM Chips

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is shown in Fig. 4-2. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations of read or write.



(a) Typical RAM chip



CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	x	x	Inhibit	High-impedence
0	1	x	x	Inhibit	High-impedence
1	0	0	0	Inhibit	High-impedence
1	0	0	1	Write	Input data to RAM
1	0	1	x	Read	Output data from RAM
1	1	x	x		

(b) Function table

Figure 4-2 Typical RAM chip

The function table listed in Fig. 4-2 (b) specifies the operation of the RAM chip. The unit is in operation only when CS1 = 1 and $\overline{CS2}$ = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When CS1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 4-3. For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

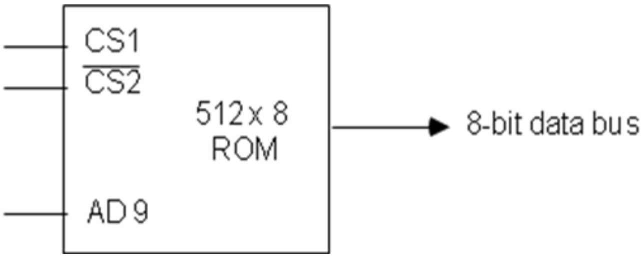


Figure 4-3 Typical ROM chip

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and $\overline{CS2}$ = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

4-2 -1 Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a *memory address map,* is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips to be used are specified in Figs. 4-2 and 4-3. The memory address map for this configuration is shown in Table 4-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines'. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to 2^9 = 512 bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained from the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-I's value.

4-2 -2 Memory Connection to CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in Fig. 4-4. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 4-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 X 4 decoder whose outputs go to the CS1 inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

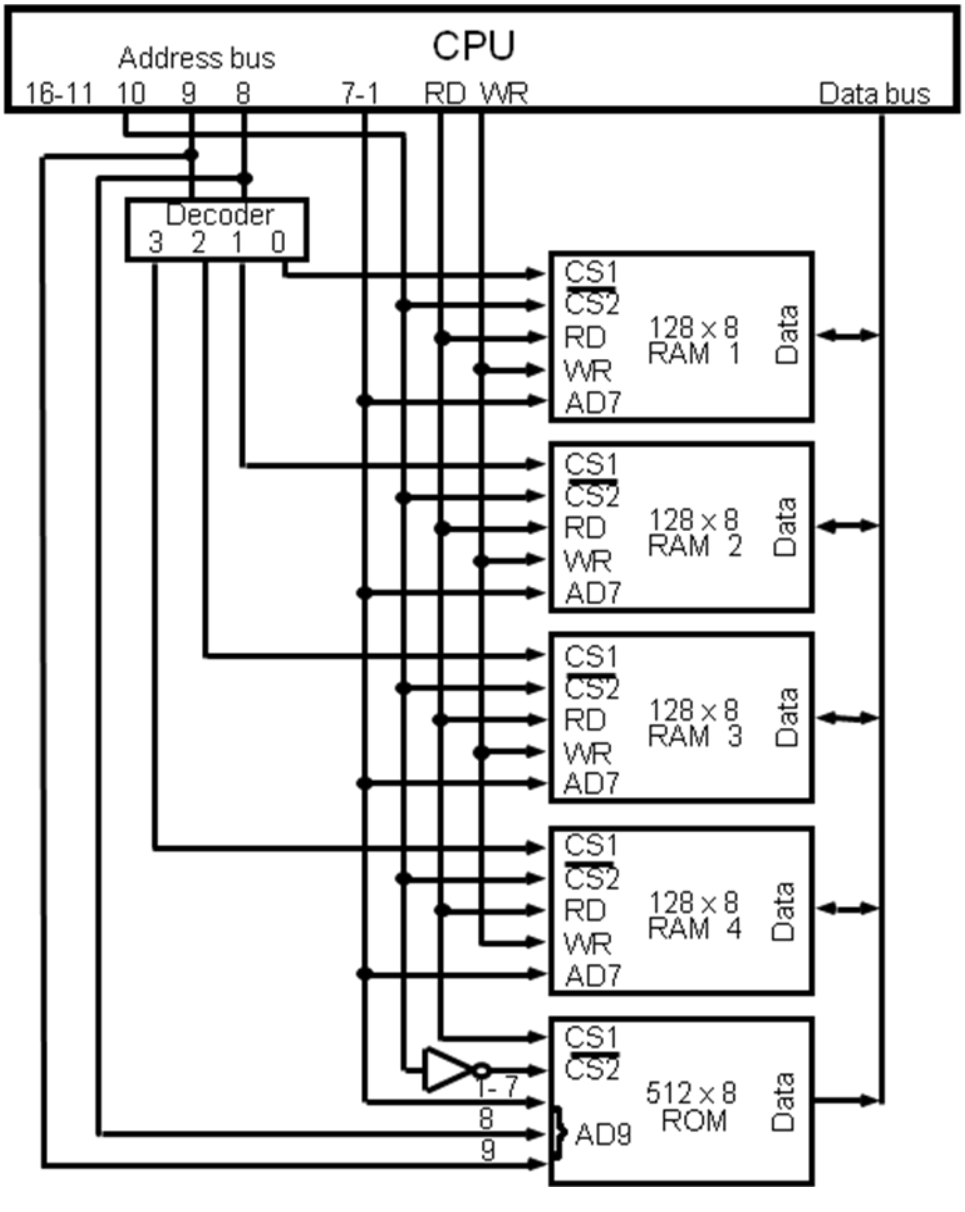


Figure 4-4 Memory connection to the CPU

4-3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have knowledge of magnetic, electronics, and electromechanical systems. Although the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a *seek* time required to position the read-write head to a location and a *transfer* time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. The recording surface rotates at uniform speed and is not started or stopped during access operations. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a *write head.* Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a *read head.* The amount of surface available for recording in a disk is greater than in a drum of equal physical size. Therefore, more information can be stored on a disk than on a drum of comparable size. For this reason, disks have replaced drums in more recent computers.

4-3 -1 Magnetic Disks

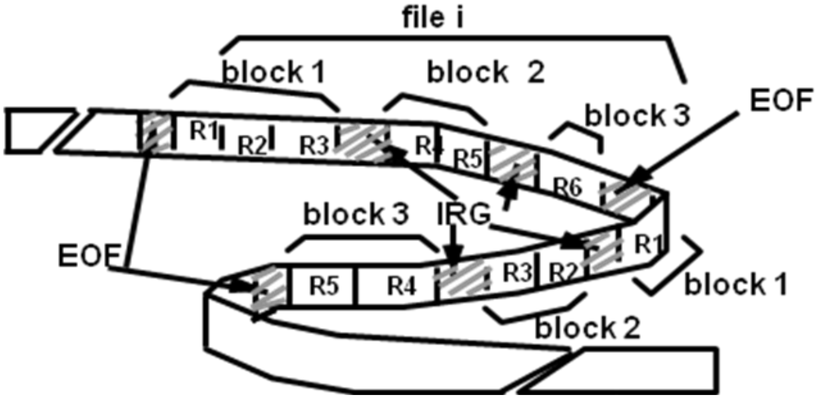
A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available, on each surface. All disks rotate together at high speed and are not stopped or started for access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector. The subdivision of one disk surface into tracks and sectors is shown in Fig. 4-5.

Some units use a single read/write head for each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

Permanent timing tracks are used in disks to synchronize the bits and recognize the sectors. A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.

A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called *hard disks.* A disk drive with removable disks is called a *floppy disk.* The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.



(a) Information Organization on Magnetic Tapes

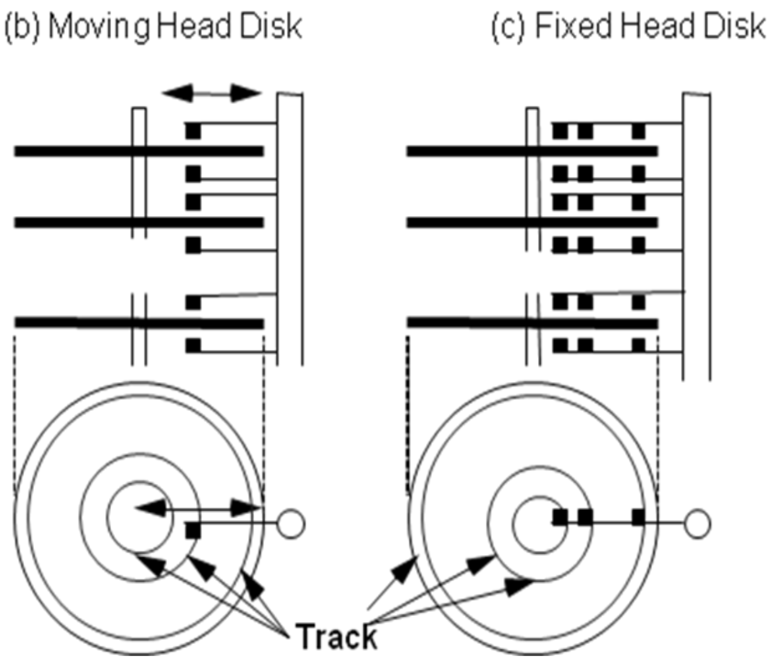


Figure 4-5 magnetic disks

4-3 -1 Magnetic Tape

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a, magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number and the number of characters in the record. Records may be of fixed or variable length.

4- 4 Associative Memory

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an *associative memory or content addressable memory* (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates all words which match the specified content and marks them for reading.

Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

4-4-1 Hardware Organization

The block diagram of an associative memory is shown in Fig. 4-6. It consists of a memory array and logic for *m* words with *n* bits per word. The argument register *A* and key register *K* each have *n* bits, one for each bit of a word. The match register M has *m* bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

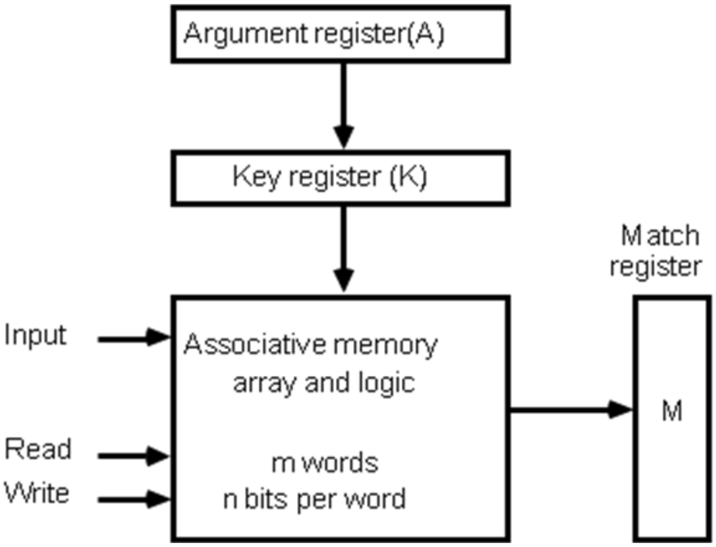


Figure 4-6 Block diagram of associative memory

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made. To illustrate with a numerical example, suppose that the argument register *A* and the key register *K have* the bit configuration shown below. Only the three leftmost bits *of A* are compared with memory words because *K has* 1's in these positions.

A	101 111100	
K	111 000000	
Word1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in Fig. 4-7. The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} *is* the cell for bit *j* in word *i.* A bit *A.* in the argument register is compared with all the bits in column *j* of the array provided that K_j= 1. This is done for all columns *j* = 1, 2, . . . , *n.* If a match occurs between all the unmasked bits of the argument and the bits in word *i,* the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i *is* cleared to 0.

The internal organization of a typical cell C_{ij} *is* shown in Fig. 4-8. It consists of a flip-flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

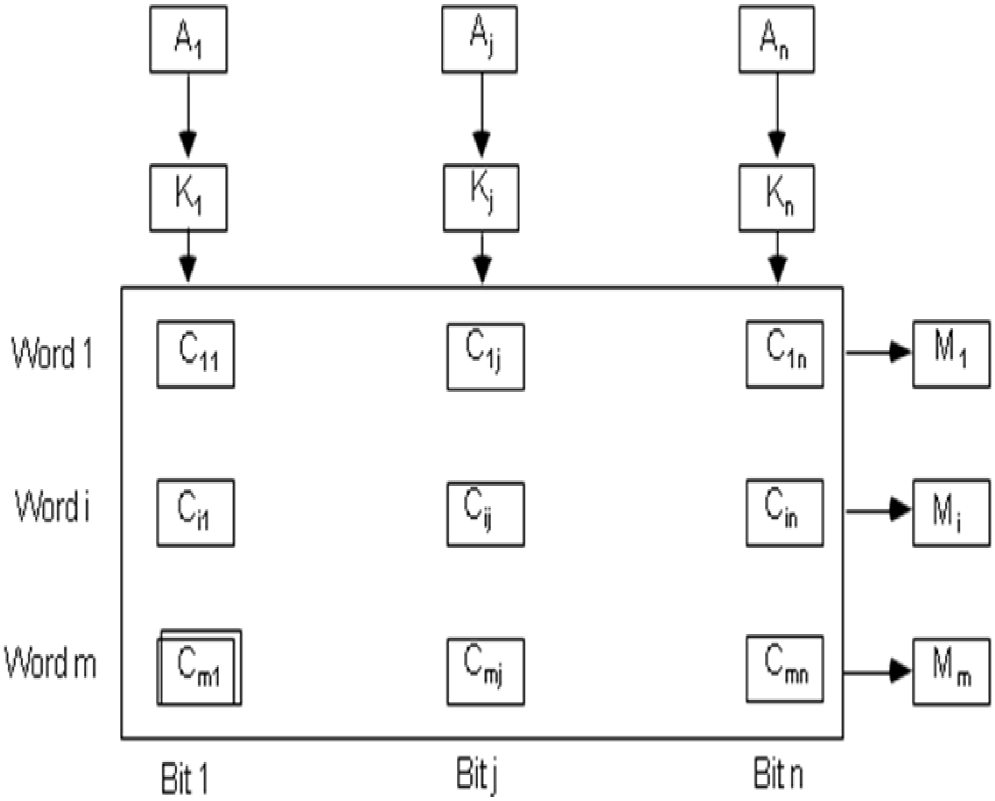


Figure 4-7 Associative memory of m word, n cells per word

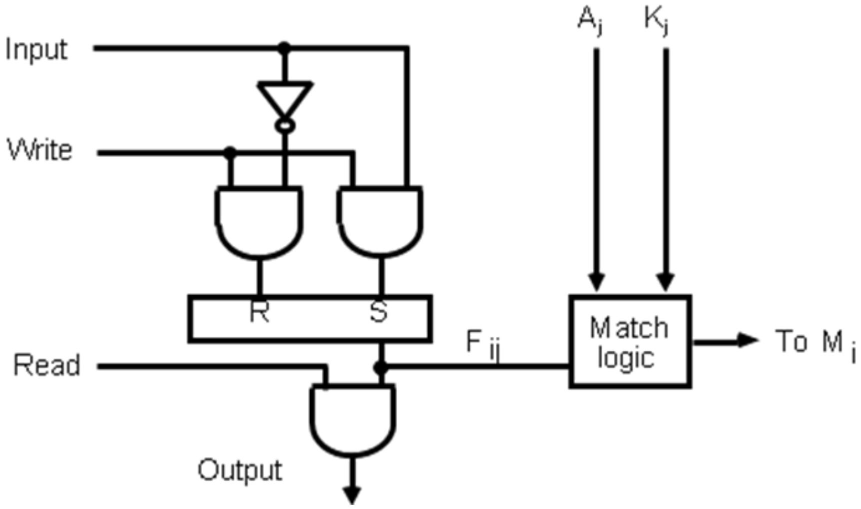


Figure 4-8 One cell of associative memory

4-4-1 Match Logic

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we *neglect* the key bits and compare the argument in *A* with the bits stored in the cells of the words. Word i is equal to the argument in *A* if A_j = F_{ij} *for j* = 1, 2, . . . ,*n.* Two bits are equal if they are both 1 and 0. The equality of two bits can be expressed logically by the Boolean function

$$X_j = A_jF_{ij} + A'_jF'_{ij}$$

where X_j = 1 if the pair of bits in position *j* are equal; otherwise X_j = 0.

For a word *i* to be equal to the argument in *A* we must have all X_j variables equal to 1. This is the condition for setting the corresponding match bit *Mi to* 1. The Boolean function for this condition is

$$M_i = x_1\, x_2\, x_3 \ldots x_n$$

and constitutes the AND operation of all pairs of matched bits in a word.

We now include the key bit K_j in the comparison logic. The requirement is that if K_j = 0, the corresponding bits of A_j, and F_{ij} need no comparison. Only when K, = 1 must they be compared. This requirement is achieved by ORing each term with K'_j thus:

$$X_j + k'_j = x_j \qquad \text{if } K_j = 1$$

$$= 1 \qquad \text{if } K_j = 1$$

When K_j = 1, we have K_j = 0 and X_j+ 0 = x_j When K_j = 0, then K'_j = 1and X_j+ 1 = 1. A term (X_j+ k'_j) will be in the 1 state if its pair of bits is not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when K_j = 1.

The match logic for word *i* in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + k'_1)\ (x_2 + k'_2)\ (x_3 + k'_3)\ldots\ldots\ (\ x_n + k'_n)$$

Each term in the expression will be equal to 1 if its corresponding K_j = 0. If K_j = 1, the term will be either 0 or 1 depending on the value of x_j. A match will occur and M_i will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of *x.,* the Boolean function above can be expressed as follows:

$$M_i= \prod_{j=1}^{n}(\ A_jF_{ij} + A'_jF'_{ij})$$

where $\prod$ is a product symbol designating the AND operation of all *n* terms. We need *m* such functions, one for each word *i* = 1, 2, 3, . . . , *m.*

The circuit for matching one word is shown in Fig. 4-9. Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j. are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i. M_i will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i, will be a 1 irrespective of the value of *A* or the word. This occurrence must be avoided during normal operation.

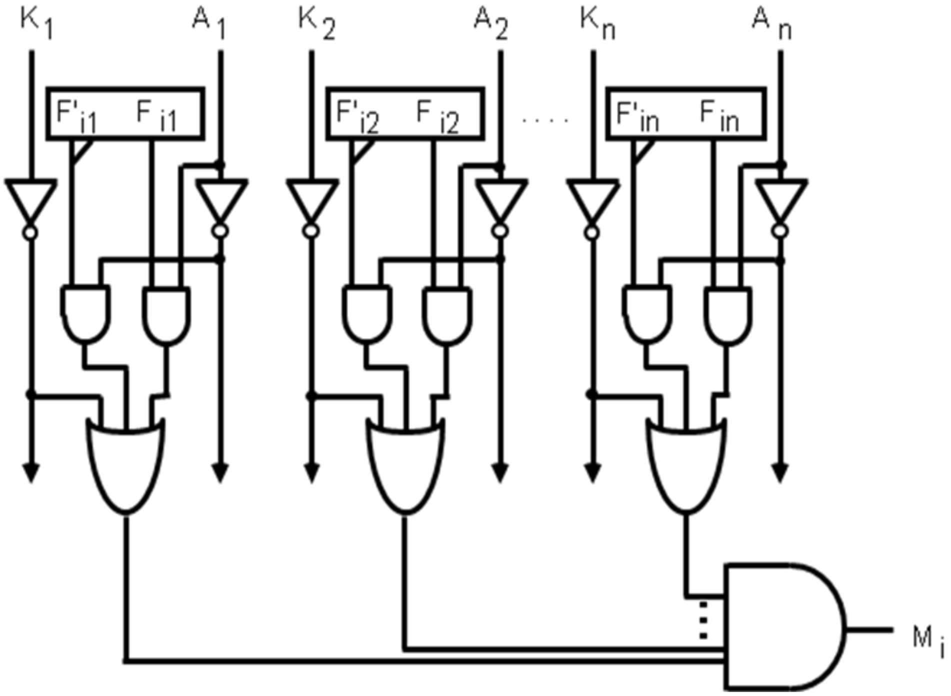


Figure 4-9 Match logic for one word of associative memory

4-4-2 Read Operation

If more than one word in memory matches the unmasked argument field, all the matched words will have 1's in the corresponding bit position of the match register. It is then necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a 1.

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the *M* register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

4-4-3 Write Operation

An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded

NOTES

with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having *m* address lines, one for each word in memory, the number of address lines can be reduced by the decoder to *d* lines, where $m = 2^d$.

If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a *tag register,* would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location. Moreover, the words that have a tag bit of 0 must be masked (together with the *K.* bits) with the argument word so that only active words are compared.

4-5 Summary

Thus, we have taken a complete view of the memory system of computer along with the various technologies. The unit has outlined the importance of memory system, the memory hierarchy, the main memory and its technologies, the secondary memories and its technologies and high-speed memories. We have also discussed the key characteristic of these memories and the technologies, which are used for constructing these memories.

4-6 Keywords

Auxiliary memory: A high-speed memory that is in a large main frame or supercomputer is not directly.

Cache memory: A special very high speed memory called cache.

Multiprogramming: Multiprogramming refers to the existence of two or more programs in different parts of the memory hierarchy at the same time.

Bootstrap loader: The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.

4-7 Exercise

1. a. How many 128 X 8 RAM chips are needed 10 provide a memory capacity of 2048 bytes?

b. How many lines of the address bus must be used to access 2048 bytes of memory? How many of these lines will be common to all chips? c. How many lines must be decoded for chip select? Specify the size of the decoders.

2. A computer uses RAM chips of 1024 X 1 capacities.

a. How many chips are needed, and how should their address lines be connected to provide a memory capacity of 1024 bytes?

b. How many chips are needed to provide a memory capacity of 16K bytes? Explain in words how the chips are to be connected to the address bus.

3. A ROM chip of 1024 X 8 bits has four select inputs and operates from a 5 volt power supply. How many pins are needed for the 1C package? Draw a block diagram and label all input and output terminals in the ROM.

4. Extend the memory system of Fig. 4-4 to 4096 bytes of RAM and 4096 bytes of ROM. List the memory-address map and indicate what size decoders are needed.

5. A computer employs RAM chips of 256 X 8 and ROM chips of 1024 X 8. The computer system needs 2K bytes of RAM, 4K bytes of ROM, and four interface units, each with four registers. A memory-mapped I/O configuration is used. The two highest-order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface registers.

a. How many RAM and ROM chips are needed?

b. Draw a memory-address map for the system. c. Give the address range in hexadecimal for RAM, ROM, and interface.

6. An 8-bit computer has a 16-bit address bus. The first 15 lines of the address are used to select a bank of 3 2K bytes of memory. The high-order bit of the address is used to select a register which receives the contents of the data bus. Explain how this configuration can be used to extend the memory capacity of the system to eight banks of 32K bytes each, for a total of 256K bytes of memory.

7**.** a. Draw the logic diagram of all the cells of one word in an associative memory. Include the read and write logic of Fig. 2-8 and the match logic of Fig. 4-9.

b. Draw the logic diagram of all cells along one vertical column (column *f)* in an associative memory. Include a common output line for all bits in the same column.

c. From the diagrams in (a) and (b) show that if output M_j *is* connected to the *read* line of the same word, then the matched word will be read out, provided that only one word matches the masked argument.

8. What additional logic is required to give a no-match result for a word in an associative memory when all key bits are zeros?

5. INTERNAL MEMORY

Structure

Objectives

At the end of this lesson you should be able to:

- Describe the importance of Cache Memory and other high speed memory.
- Discuss about Memory mapping
- Describe the Virtual Memory
- Describe the Memory Management Hardware

5-1 Cache Memory

Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of *locality reference.* The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions is fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table-lookup procedures repeatedly refer to that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively infrequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as a *cache memory.* It is placed between the CPU and main memory as illustrated in Fig. 4-1(lesson 4). The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the average memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is frequently measured in terms of a quantity called *hit ratio.* When the CPU refers to memory and finds the word in cache, it is said to produce a *hit.* If the word is not found in cache, it is in main memory and it counts as a *miss.* The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is best measured experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100ns, a main memory access time of 1000ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a *mapping* process. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping

2. Direct mapping

3. Set-associative mapping

To help in the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in Fig. 5-1. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

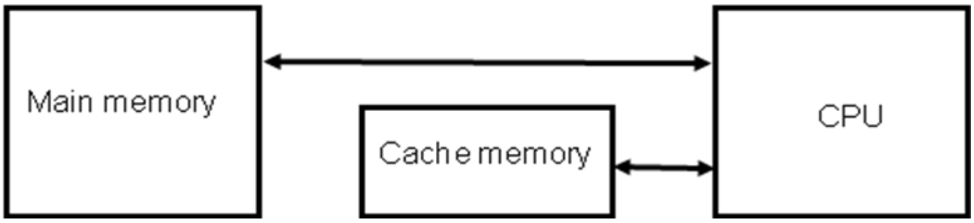


Figure 5-1 Example of cache memory

5-1-1 Associative Mapping

The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in Fig. 5-2. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address.

If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

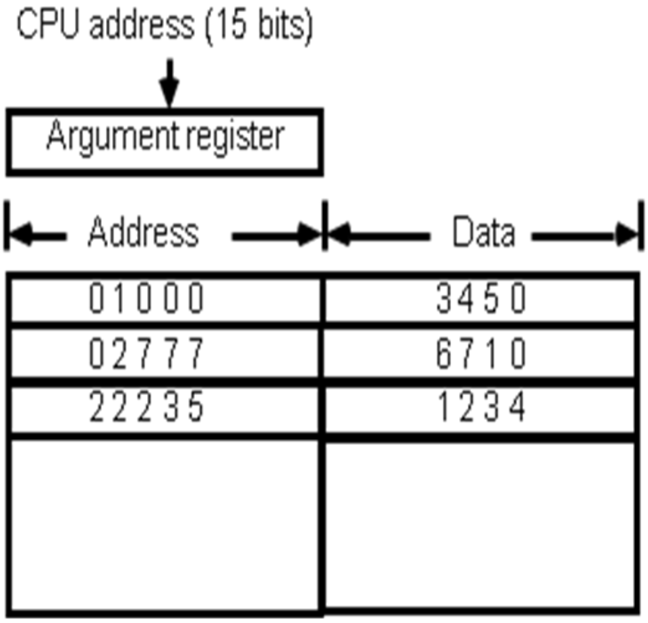


Figure 5-2 Associative mapping cache

5-1-2 Direct Mapping

Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig. 5-3. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the *index* field and the remaining six bits form the *tag* field. The figure shows that main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: *k* bits for the index field and *n - k* bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the *k-bit* index to access the cache. The internal organization of the words in the cache memory is as shown in Fig. 5-4 (b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

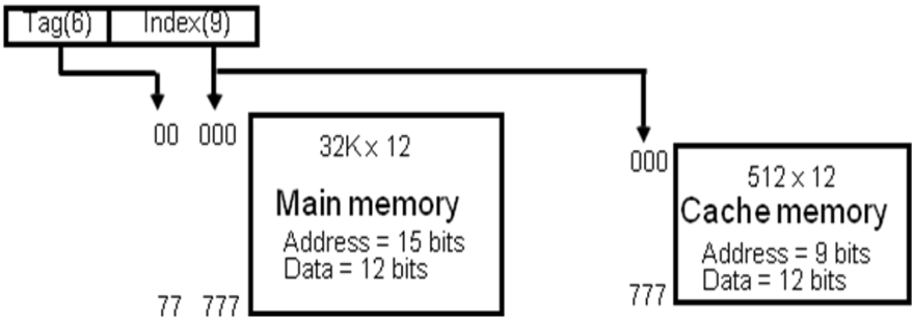
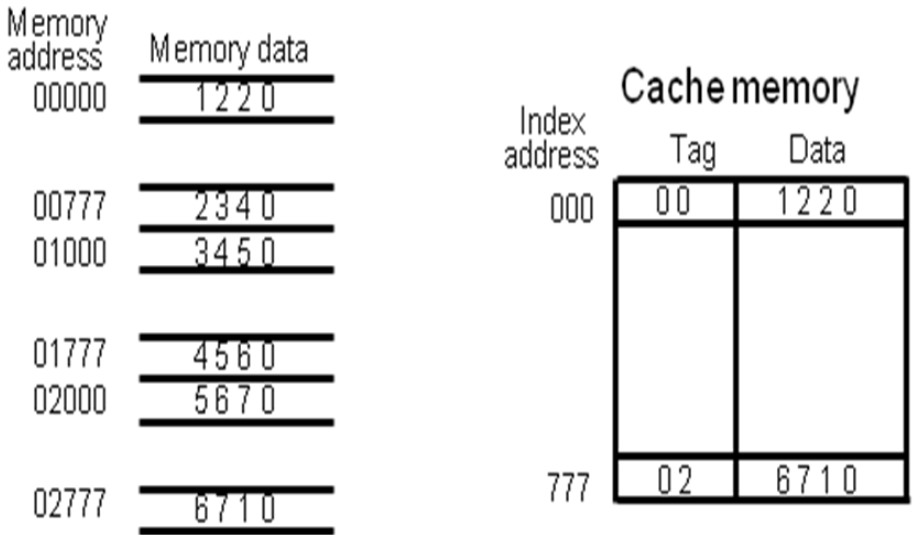


Figure 5-3 Addressing Relationships between main memory and cache memories



(a) Main Memory (b) Cache memory

Figure 5-4 direct mapping cache organization

To see how the direct-mapping organization operates, consider the numerical example shown in Fig. 5-4. The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670

The direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig. 5-5

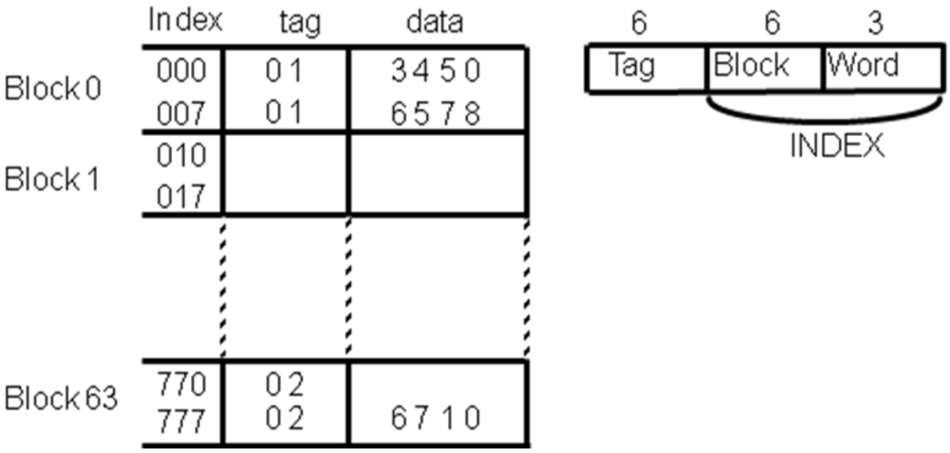
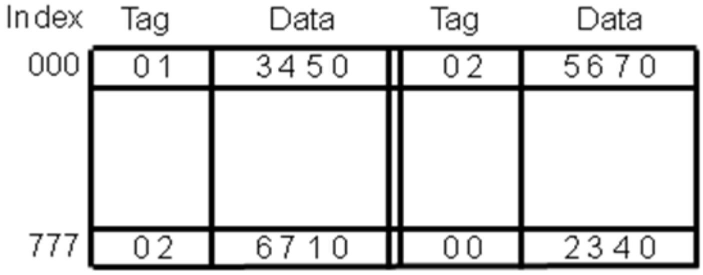


Figure5-5 Direct mapping cache with block size of 8 words

The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since 64 X 8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

5-1-2 Set-Associative Mapping

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Fig. 5-6. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 X 36. It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size *k* will accommodate *k* words of main memory in each word of cache.



Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

Figure 5-6 Two-way set-associative mapping cache

The octal numbers listed in Fig. 5-6 are with reference to the main memory contents illustrated in Fig. 5-4 (a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search: thus the name "set-associative." The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache. However, an increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first-out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

5-1-3 Writing into Cache

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can precede.

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is trough called the *write-through* method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

The second procedure is called the *writs-back* method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

5-1-4 Cache Initialization

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some non-valid data. It is customary to include with each word in cache a *valid bit* to indicate whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

5-2 Virtual Memory

In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU. *Virtual memory* is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

5-2-1 Address Space and Memory Space

An address used by a programmer will be called a *virtual address,* and the set *of* such addresses the *address space.* An address in main memory is called a *location* or *physical address.* The set of such locations is called the *memory space.* Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example *N* = 1024K and *M*= 32K.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember that for efficient transfers, auxiliary storage moves an entire record to the main memory.) A table is then needed, as shown in Fig. 5-7, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

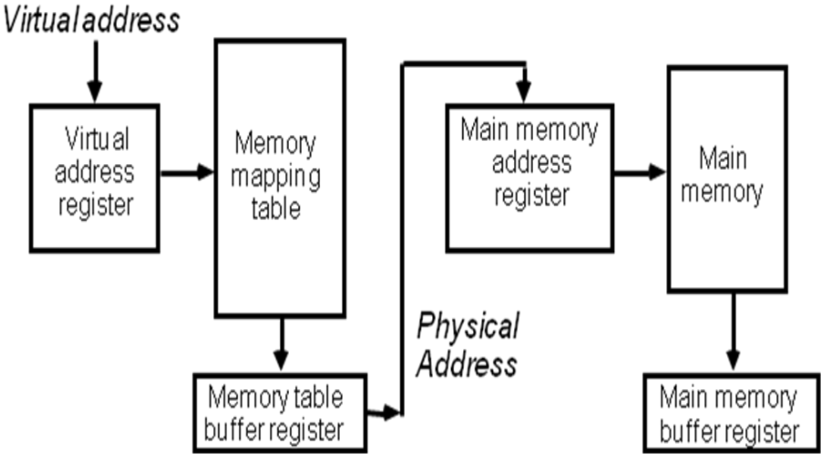


Figure 5-7 Memory table for mapping a virtual address

The mapping table may be stored in a separate memory as shown in Fig. 5-7 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

5-2-2 Address Mapping Using Pages

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called *blocks,* which may range from 64 to 4096 words each. The term *page* refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term "page frame" is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. 5-8. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

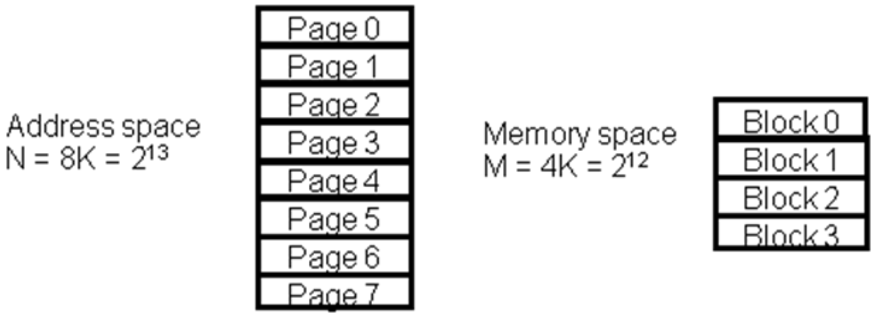


Figure 5-8 Address space and memory space split into groups of 1K words

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, *p* bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig. 5-8, a virtual address has 13 bits. Since each page consists of 2^{10} = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

The organization of the memory mapping table in a paged system is shown in Fig. 5-9. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available j in main memory. The CPU references a word in memory with a virtual | address of 13 bits. The three high-order bits of the virtual address specify a j page number and also an address for the memory-page table. The content of the word in the memory page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low-order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

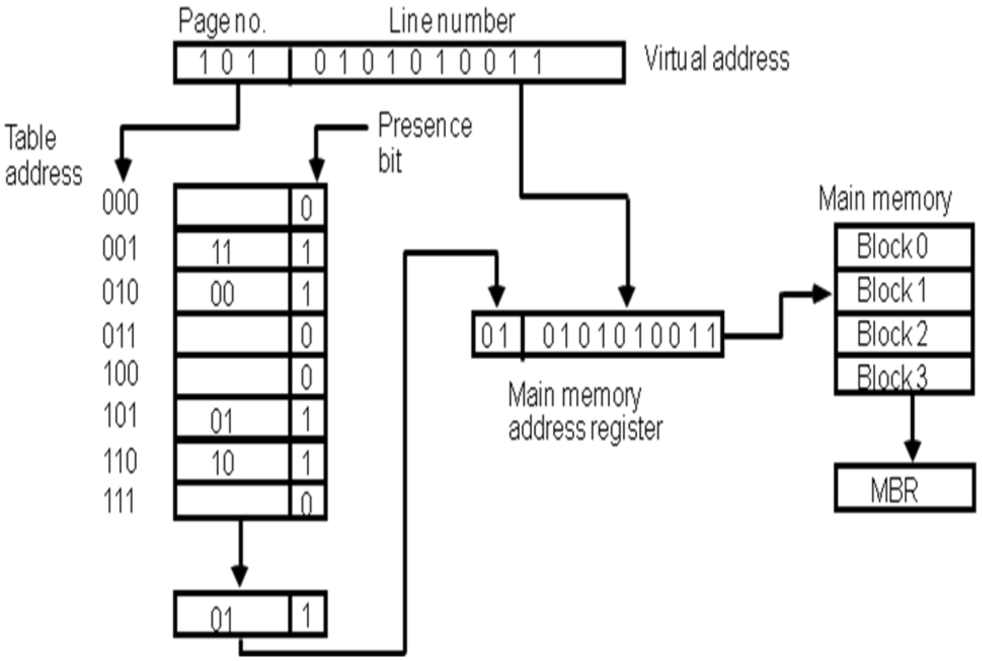


Figure 5-9 Memory table in a paged system

5-2-3 Associative Memory Page Table

A random-access memory page table is inefficient with respect to storage utilization. In the example of Fig. 5-9 we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks. In general, a system with *n* pages and *m* blocks would require a memory-page table of *n* locations of which up to *m* blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the example of Fig. 5-9. We replace the random access memory-page table with an associative memory of four words as shown in Fig. 5-10. Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument register are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

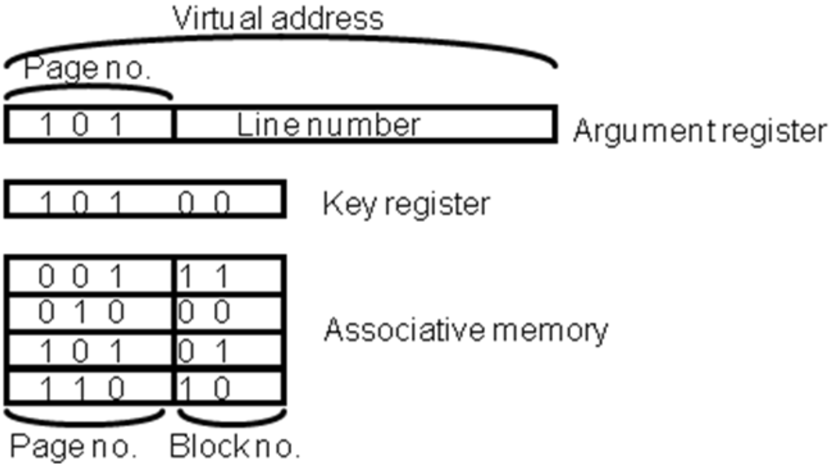


Figure 5-10 An associative memory page table

5-2-4 Page Replacement

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called *page fault.* When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the *first-in, first-out* (FIFO) and the *least recently used* (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantage that under certain circumstances pages are removed and loaded from memory too frequently.

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count The counters are often called *aging registers,* as their count indicates their age, that is, how long ago their associated pages have been referenced.

5-3 Memory Management Hardware

In a multiprogramming environment where many programs reside in memory it becomes necessary to move programs and data around the memory, to vary the amount of memory in use by a given program, and to prevent a program from changing other programs. The demands on computer memory brought about by multiprogramming have created the need for a memory management system. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in many computers. Here we are concerned with the hardware unit associated with the memory management system.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses

2. A provision for sharing common programs stored in memory by different users

3. Protection of information against unauthorized access between users and preventing users from changing operating system functions

The dynamic storage relocation hardware is a mapping process similar to the paging system described in Sec. 5-2. The fixed page size used in the virtual memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and data into logical parts called segments. A *segment is* a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are a subroutine, an array of data, a table of symbols, or a user's program.

The sharing of common programs is an integral part of a multiprogramming system. For example, several users wishing to compile their Fortran programs should be able to share a single copy of the compiler rather than each user having a separate copy in memory. Other system programs residing in memory are also shared by all users in a multiprogramming system without having to produce multiple copies.

The third issue in multiprogramming is protecting one program from unwanted interaction with another. An example of unwanted interaction is one user's unauthorized copying of another user's program. Another aspect of protection is concerned with preventing the occasional user from performing operating system functions and thereby interrupting the orderly sequence of operations in a computer installation. The secrecy of certain programs must be kept from unauthorized personnel to prevent abuses in the confidential activities of an organization.

The address generated by a segmented program is called a *logical address.* This is similar to a virtual address except that logical address space is associated with variable-length segments rather than fixed-length pages. The logical address may be larger than the physical memory address as in virtual memory, but it may also be equal, and sometimes even smaller than the length of the physical memory address. In addition to relocation information, each segment has protection information associated with it. Shared programs are placed in a unique segment in each user's logical address space so that a single physical copy can be shared. The

NOTES

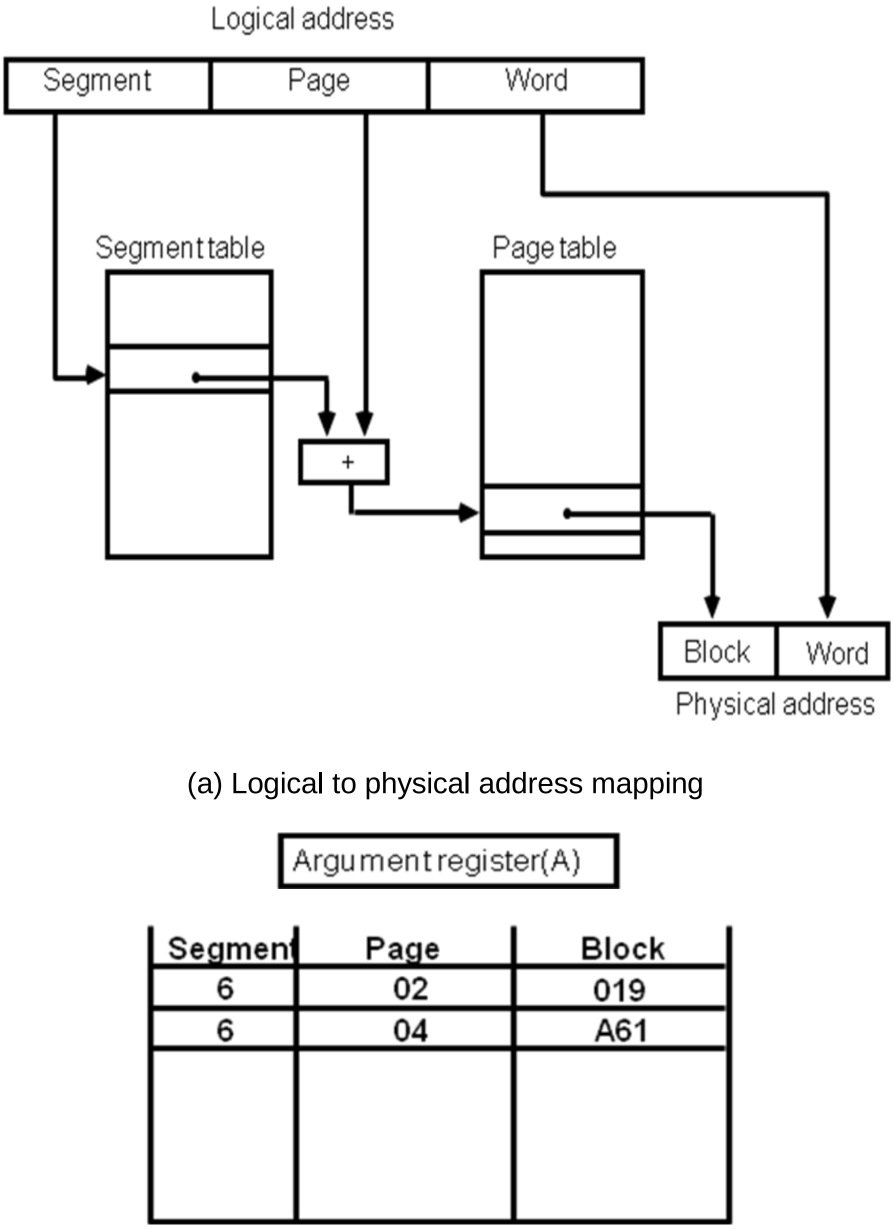
function of the memory management unit is to map logical addresses into physical addresses similar to the virtual memory mapping concept.

5-3 -1Segmented-Page Mapping

It was already mentioned that the property of logical space is that it uses variable-length segments. The length of each segment is allowed to grow and contract according to the needs of the program being executed. One way of specifying the length of a segment is by associating a number of equal-size pages. To see how this is done, consider the logical address shown in Fig.5-11. The logical address is partitioned into three fields. The segment field specifies a segment number. The page field specifies the page within the segment and the word field gives the specific word within the page. A page field of *k* bits can specify up to 2^k pages. A segment number may be associated with just one page or with as many as 2^k pages. Thus the length of a segment would accord the number of pages that are assigned to it.

The mapping of the logical address into a physical address is done by means of two tables, as shown in Fig. 5-11 (a). The segment number of the logical address specifies the address for the segment table. The entry in the segment table is a pointer address for a page table base. The page table base is added to the page number given in the logical address. The sum produces a pointer address to an entry in the page table. The value found in the page table provides the block number in physical memory. The concatenation of the block field with the word field produces the final physical mapped address.

The two mapping tables may be stored in two separate small memories or in main memory. In either case, a memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table, and the third from main memory. This would slow the system significantly when compared to a conventional system that requires only one reference to memory. To avoid this speed penalty, a fast associative memory is used to hold the most recently referenced table entries. (This type of memory is sometimes called a *translation look-aside buffer,* abbreviated TLB.) The first time a given block is referenced, its value together with the corresponding segment and page numbers are entered into the associative memory as shown in Fig. 5-11(b). Thus the mapping process is first attempted by associative search with the given segment and page numbers. If it succeeds, the mapping delay is only that of the associative memory. If no match occurs, the slower table mapping of Fig. 5-11(a) is used and the result transformed into the associative memory for future reference.



(a) Logical to physical address mapping

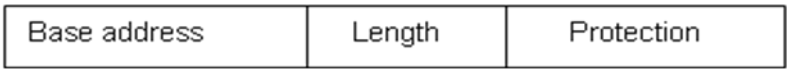
Segment	Page	Block
6	02	019
6	04	A61

Figure 5-11 Mapping in segmented-page memory management unit

5-3 -2 Memory Protection

Memory protection can be assigned to the physical address or the logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits that indicate the type of access allowed to its corresponding block. Every time a page is moved from one block to another it would be necessary to update the block protection bits. A much better place to apply protection is in the logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of the memory management hardware.

The content of each entry in the segment table or a segment register is called a descriptor. A typical descriptor would contain, in addition to a base address field, one or two additional fields for protection purposes. A typical format for a segment descriptor is shown in Fig. 5-12. The base address field gives the base of the page table address in a segmented-page organization or the block base address in a segment register organization. This is the address used in mapping from a logical to the physical address. The length field gives the segment size by specifying the maximum number of pages assigned to the segment. The length field is compared against the page number in the logical address. A size violation occurs if the page number falls outside the segment length boundary. Thus a given program and its data cannot access memory not assigned to it by the operating system.



Base address	Length	Protection

Figure 5-12 Format of a typical segment descriptor

The protection field in a segment descriptor specifies the access rights available to the particular segment. In a segmented-page organization, each entry in the page table may have its own protection field to describe the access rights of each page. The protection information *is set* into the descriptor by the master control program of the operating system. Some of the access rights of interest that are used for protecting die programs residing in memory are:

1. Full read and write privileges
2. Read only (write protection)
3. Execute only (program protection)
4. System only (operating system protection)

Full read and write privileges are given to a program when it is executing its own instructions. Write protection is useful for sharing system programs such as utility programs and other library routines. These system programs are stored in an area of memory where they can be shared by many users. They can be read by all programs, but no writing is allowed. This protects them from being changed by other programs.

The execute-only condition protects programs from being copied. It restricts the segment to be referenced only during the instruction fetch phase but not during the execute phase. Thus it allows the users to execute the segment program instructions but prevents them from reading the instructions as data for the purpose of copying their content.

Portions of the operating system will reside in memory at any given time. These system programs must be protected by making them inaccessible to unauthorized users. The operating system protection condition is placed in the descriptors of all operating system programs to prevent the occasional user from accessing operating system segments.

5-4 Summary

In this lesson, we have discussed about cache memory maintains. A paged virtual memory augments a main memory with disk storage. The physical memory serves as a window on the paged virtual memory, which is maintained in its entirety on a hard magnetic disk. Cache and paged virtual memories are commonly used on the same computer, but for different reasons. A cache memory improves the average access time to the main memory, whereas a paged virtual memory extends the size of the main memory. Our discussion on virtual memory started with the issues related to address translation. Three address translation techniques were discussed and compared. These are the direct, associative, and the set-associative techniques.

5-5 Keywords

Locality of reference: Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory.

Hit ratio: The performance of cache memory is frequently measured in terms of a quantity called *hit ratio.* When the CPU refers to memory and finds the word in cache, it is said to produce a *hit.*

Mapping process: The transformation of data from main memory to cache memory is referred to as a *mapping* process.

Write-through: The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address.

Write-back: The cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory.

5-6 PROBLEMS

1. The logical address space in a computer system consists of 128 segments. Each segment can have up to 32 pages of 4K words in each. Physical memory consists of 4K blocks of 4K words in each. Formulate the logical and physical address formats.

2. A virtual memory system has an address space of 8K. words, a memory space of 4K words, and page and block sizes of 1K words (see Fig. 5-8). The following page reference changes occur during a given time interval. (Only page changes are listed. If the same page is referenced again, it is not listed twice.) 420126140102357. Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm used is (a) FIFO; (b) LRU

3. An address space is specified by 24 bits and the corresponding memory space by 16 bits.

   a. How many words are there in the address space?

   b. How many words are there in the memory space?

   c. If a page consists of 2K words, how many pages and blocks are there in the system?

4. A digital computer has a memory unit of 64K X 16 and a cache memory of 1K words. The cache uses direct mapping with a block size of four words.

   a. How many bits are there in the tag, index, block, and word fields of the address format?

   b. How many bits are there in each word of cache, and how are they divided into functions? Include a valid bit.

   c. How many blocks can the cache accommodate?

5. A four-way set-associative cache memory has four words in each set. A replacement procedure based on the least recently used (LRU) algorithm is implemented by means of 2-bit counters associated with each word in the set. A value in the range 0 to 3 is thus recorded for each word. When a hit occurs, the counter associated with the referenced word is set to 0, those counters with values originally lower than the referenced one are incremented by 1, and all others remain unchanged. If a miss occurs, the word with counter value 3 is removed, the new word is put in its place, and its counter is set to 0. The other three counters are incremented by 1. Show that this procedure works for the following sequence of word reference: A, B, C, D, B, E, D, A, C, E, C, E. (Start with A, B, C, D as the initial four words, with word A being the least recently used.)

UNIT – IV

1. INPUT-OUTPUT INTERFACE

Structure

Objectives

At the end of this lesson you should be able to:

- Describe varies types of Character codes
- Discuss about Input-Output Interfaces

1-1 Introduction

The input-output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. A computer serves no useful purpose without the ability to receive information from an outside source and to transmit results in a meaningful form.

The most familiar means of entering information into a computer is through a typewriter-like keyboard that allows a person to enter alphanumeric information directly. Every time a key is depressed, the terminal sends a binary coded character to the computer. The fastest possible speed for entering information this way depends on the person's typing speed. On the other hand, the central processing unit is an extremely fast device capable of performing operations at very high speed.

When input information is transferred to the processor via a slow keyboard, the processor will be idle most of the time while waiting for the information to arrive. To use a computer efficiently, a large amount of programs and data must be prepared in advance and transmitted into a storage medium such as magnetic tapes or disks. The information in the disk is then transferred into computer memory at a rapid rate. Results of programs are also transferred into a high-speed storage, such as disks, from which they can be transferred later into a printer to provide a printed output of results.

Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the total computer system. Input or output devices attached to the computer are also called peripherals. There are three types of peripherals such as input, output, and input-output peripherals. These peripherals may be analog or digital and serial or parallel. Among the most common peripherals are keyboards, display units, and printers. Peripherals that provide auxiliary storage for the system are magnetic disks and tapes. Peripherals are electromechanical and electromagnetic devices of some complexity. Only a very brief discussion of their function will be given here without going into detail of their internal construction.

Video monitors are the most commonly used peripherals. They consist of a keyboard as the input device and a display unit as the output device. There are different types of video monitors, but the most popular use a cathode ray tube (CRT). The CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically. To produce a pattern on the screen, a grid inside the CRT receives a variable voltage that causes the beam to hit the screen and make it glow at selected spots. Horizontal and vertical signals deflect the beam and make it sweep across the tube, causing the visual pattern to appear on the screen. A characteristic feature of display devices is a cursor that marks the position in the screen where the next character will be inserted. The cursor can be moved to any position in the screen, to a single character, the beginning of a word, or to any line. Edit keys add or delete information based on the cursor position. The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.

Printers provide a permanent record on paper of computer output data or text. There are three basic types of character printers: daisywheel, dot matrix, and laser printers. The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon. The dot matrix printer contains a set of dots along the printing mechanism. For example, a 5 X 7 dot matrix printer that prints 80 characters per line has seven horizontal lines, each consisting of

5 X 80 = 400 dots. Each dot can be printed or not, depending on the specific characters that are printed on the line. The laser printer uses a rotating photographic drum that is used to imprint the character images. The pattern is then transferred onto paper in the same manner as a copying machine.

Magnetic tapes are used mostly for storing files of data: for example, a company's payroll record. Access is sequential and consists of records that can be accessed one after another as the tape moves along a stationary read-write mechanism. It is one of the cheapest and slowest methods for storage and has the advantage that tapes can be removed when not in use. Magnetic disks have high-speed rotational surfaces coated with magnetic material. Access is achieved by moving a read-write mechanism to a track in the magnetized surface. Disks are used mostly for bulk storage of programs and data.

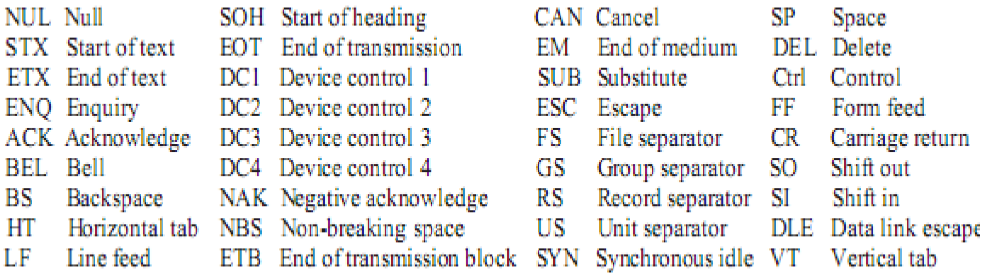
Other input and output devices encountered in computer systems are digital incremental plotters, optical and magnetic character readers, analog-to-digital converters, and various data acquisition equipment. Not all input comes from people, and not all output is intended for people. Computers are used to control various processes in real time, such as machine tooling, assembly line procedures, and chemical and industrial processes. For such applications, a method must be provided for sensing status conditions in the process and sending control signals to the process being controlled.

The input-output organization of a computer is a function of the size of the computer and the devices connected to it. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral. Certain techniques common to most peripherals are presented in this lesson.

1-1-1 ASCII Alphanumeric Characters

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer. The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in Table 1-1. The seven bits of the code are designated by b_1 through b_7, with b_7 being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z**,** the 26 lowercase letters, the 10 numerals 0 through 9, and 32 special printable characters such as %, * , and $.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL



NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

TABLE 1-1 American Standard Code for Information Interchange (ASCII)

The 34 control characters are designated in the ASCII table with abbreviated names. They are listed again below the table with their functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include the familiar typewriter controls, such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions like paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are useful during the transmission of text between remote terminals.

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a byte. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font.

When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded character.

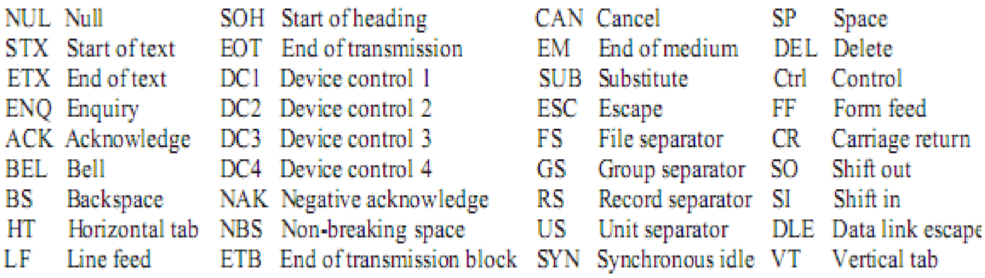
1-1-2 The EBCDIC Character set

A problem with the ASCII code is that only 128 characters can be represented, which is a limitation for many keyboards that have a lot of special characters in addition to upper and lower case letters. The Extended Binary Coded Decimal Interchange Code (**EBCDIC**) is an eight-bit code that is used extensively in IBM mainframe computers. Since seven-bit ASCII characters are frequently represented in an eight-bit modified form (one character per byte), in which a 0 or a 1 is appended to the left of the seven-bit pattern, the use of EBCDIC does not place a greater demand on the storage of characters in a computer. For serial transmission, however, (see Chapter 8), an eight-bit code takes more time to transmit than a seven-bit code, and for this case the wider code does make a difference.

The EBCDIC code is summarized in table 1-2. There are gaps in the table, which can be used for application specific characters. The fact that there are gaps in the upper and lower case sequences is not a major disadvantage because character manipulations can still be done as for ASCII, but using different offsets.

TABLE 1-2 Extended Binary Coded Decimal Interchange Code (**EBCDIC**)

00 NUL	20 DS	40 SP	60 –	80	A	C0 {	E0
01 SOH	21 SOS	41	61 /	81 a	0	C1 A	
02 STX	22 FS	42	62	82 b	A1 ~	C2 B	\ E1
03 ETX	23	43	63	83 c	A2 s	C3 C	E2 S
04 PF	24 BYP	44	64	84 d	A3 t	C4 D	E3 T
05 HT	25 LF	45	65	85 e	A4 u	C5 E	E4 U
06 LC	26 ETB	46	66	86 f	A5 v	C6 F	E5 V
07 DEL	27 ESC	47	67	87 g	A6 w	C7 G	E6 W
08	28	48	68	88 h	A7 x	C8 H	E7 X
09	29	49	69	89 i	A8 y	C9 I	E8 Y
0A SMM	2A SM	4A ¢	6A ‘	8A	A9 z	CA	E9 Z
0B VT	2B CU2	4B	6B ,	8B	AA	CB	EA
0C FF	2C	4C <	6C %	8C	AB	CC	E
0D CR	2D ENQ	4D (	6D _	8D	AC	C	B
0E SO	2E ACK	4E +	6E >	8E	AD	D	EC
0F SI	2F BEL	4F |	6F ?	8F	AE	CE	E
10 DLE	30	50 &	70	90	AF	CF	D
11 DC1	31	51	71	91 j	B0	D0 }	EE
12 DC2	32 SYN	52	72	92 k	B	D1 J	EF
13 TM	33	53	73	93 l	1	D2 K	F0 0
14 RES	34 PN	54	74	94 m	B	D3 L	F1 1
15 NL	35 RS	55	75	95 n	2	D4	F2 2
16 BS	36 UC	56	76	96 o	B	M D5	F3 3
17 IL	37 EOT	57	77	97 p	3	N	F4 4
18 CAN	38	58	78	98 q	B	D6 O	F5 5



NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

1-1-3 Unicode Character set

The ASCII and EBCDIC codes support the historically dominant (Latin) character sets used in computers. There are many more character sets in the world, and a simple ASCII-to-language-X mapping does not work for the general case, and so a new universal character standard was developed that supports a great breadth of the world's character sets, called Unicode.

Unicode is an evolving standard. It changes as new character sets are introduced into it, and as existing character sets evolve and their representations are refined. In version 2.0 of the Unicode standard, there are 38,885 distinct coded characters that cover the principal written languages of the Americas, Europe, the Middle East, Africa, India, Asia, and Pacifica.

The Unicode Standard uses a 16-bit code set in which there is a one-to-one correspondence between 16-bit codes and characters. Like ASCII, there are no complex modes or escape codes. While Unicode supports many more characters than ASCII or EBCDIC, it is not the end-all standard. In fact, the 16-bit Unicode standard is a subset of the 32-bit ISO 10646 Universal Character Set (UCS-4).

Glyphs for the first 256 Unicode characters are shown in Table 1-2, according to Unicode version 2.1. Note that the first 128 characters are the same as for ASCII.

1-2 Input-Output Interfaces

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.

3. Data codes and formats in peripherals differ from the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called interface units because they interface between the processor bus and the peripheral device. The word "Interface" is a general term for the point of contact between two parts of a system. In digital computer system the interface is referred as a complementary set of signal connection points between two parts of a system. Therefore, "to interface" means to attach two or more components or systems, via their respective interface points, for data exchanges between them. Two main types of interface are CPU interface that corresponds to the system bus and input-output interface that depends on the nature of input-output device. To attach an input-output device to CPU and input-output interface, circuit is placed between the device and the system bus. This circuit is meant for matching the signal formats and timing characteristics of the CPU interface to those of the input-output device interface. The main function of input-output interface circuit is data conversion, synchronization and device selection. Data conversion refers to conversion between digital and analog signals, and conversion between serial and parallel data formats. Synchronation refers to matching of operating speeds of CPU and other peripherals. Device selection refers to the selection of I/O device by CPU in a queue manner. In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

1-2-1 I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 1-1. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

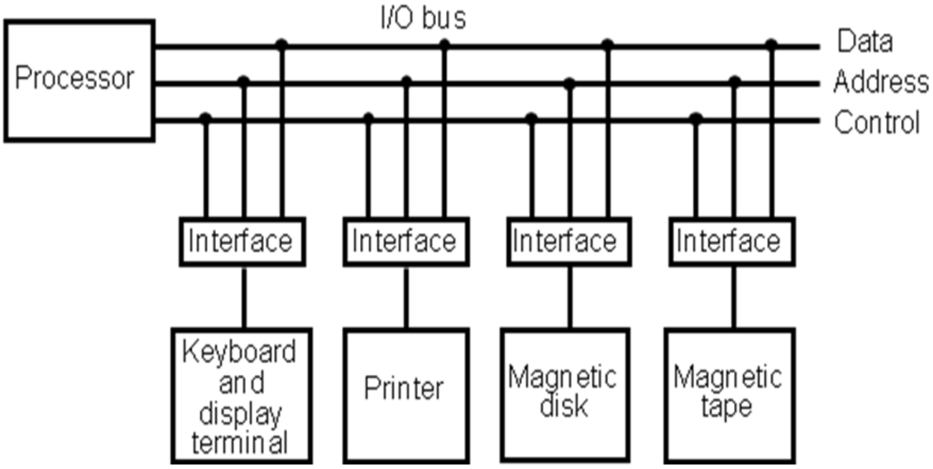


Figure 1-1 Connection of I/O bus to input-output devices

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface.

At the same time that the address is made available in the address lines, the processor provides a function code in the control lines. The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

A control command is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

A status command is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

A data output command causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

1-2-1 I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel.

1-2-2 Isolated versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O write control lines are enabled during an I/O transfer. The memory read and memory write control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the isolated I/O method for assigning addresses in a common bus.

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

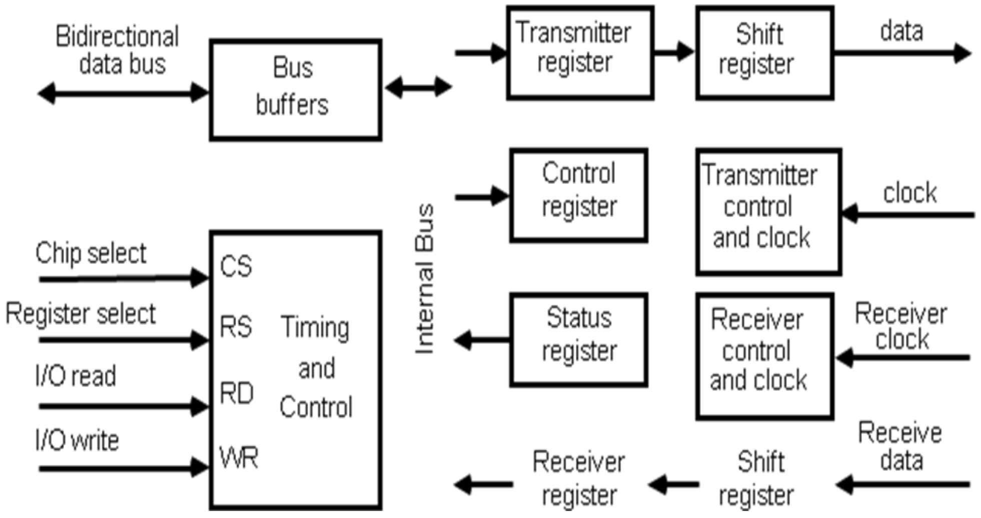
The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduce the memory address range available.

In a memory mapped I/O organization there are no specific inputs or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.

Computers with memory mapped I/O can use memory type instructions to access I/O data. It allows the computer to use the same instructions for either input-output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory reference instructions than I/O instructions. With memory mapped I/O all instructions that refer to memory are also available for I/O.

1-2-3 Example of I/O Interface

An example of an I/O interface unit is shown in block diagram form in Fig. 1.2. It consists of two data registers called parts, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.



CS	RS	Oper.	Register selected
0	x	x	None
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Figure 1-2 Example of I/O interface

The I/O data to and from the device can be transferred into either port A or port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular interface register with which the CPU communicates.

The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. For example, port A may be denned as an input port and port B as an output port. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port A has received a new data item from the I/O device. Another bit in the status register may indicate that a parity error has occurred during the transfer.

The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS] input when the interface is selected by the address bus. The two register select inputs RS1 and RSO are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

1-3 Summary

This completes our discussion on the Character codes, such as ASCII, EBCDIC, and Unicode, have finite sizes and can thus be completely represented in a finite number of bits. The number of bits used for representing numbers is also finite, and as a result only a subset of the real numbers can be represented. This leads to the notions of range, precision, and error. Input, output, and communication involve the transfer of information between transmitters and receivers. The transmitters, receivers, and methods of communication are often mismatched in terms of speed and in how information is represented, and so an important consideration is how to match input and output devices with a system using a particular method of communication. This lesson focused on the I/O system and the way the processor and the I/O devices exchange data in a computer system.

1-4 Keywords

CRT: Cathode Ray Tube

ASCII: American Standard Code for Information Interchange. It uses seven bits to code 128 characters

EBCDIC: Extended Binary Coded Decimal Interchange Code (**EBCDIC**) is an eight-bit code that is used extensively in IBM mainframe computers.

Memory mapped I/O: In this case there is no deference between memory and I/O address.

I/O mapped I/O: In this case there is deference between memory and I/O address (separate address locations).

1-5 Exercise

1. The addresses assigned to the four registers of the I/O interface of Fig. 1-2 are equal to the binary equivalent of 12, 13, 14, and 15. Show the external circuit that must be connected between an 8-bit I/O address from the CPU and the CS, RSl, and RSO inputs of the interface

2. Six interface units of the type shown in Fig. 1-2 are connected to a CPU that uses an I/O address of eight bits. Each one of the six chip select (CS) inputs is connected to a different address line. Thus the high-order address line is connected to the CS input of the first interface unit and the sixth address line is connected to the CS input of the sixth interface unit. The two low-order address lines are connected to the RSl and RSO of all six interface units. Determine the 8-bit address of each register in each interface.

3. List four peripheral devices that produce an acceptable output for a person to understand. Write your full name in ASCII using eight bits per character with the leftmost bit always 0. Include a space between names and a period after middle initial.

4. What is the difference between isolated I/O and memory-mapped I/O? What are the advantages and disadvantages of each?

1-6 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha — BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes.

2. ASYNCHRONOUS DATA TRANSFER

Structure

Objectives

At the end of this lesson you should be able to:

- Discuss about the asynchronous data transfer
- Describe the strobe control
- Describe the handshaking
- Discuss about the Asynchronous Serial Transfer
- Discuss about the Asynchronous Communication Interface
- Discuss First-In, First-Out Buffer

2-1 Introduction

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

2-2 Asynchronous Data Transfer

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.

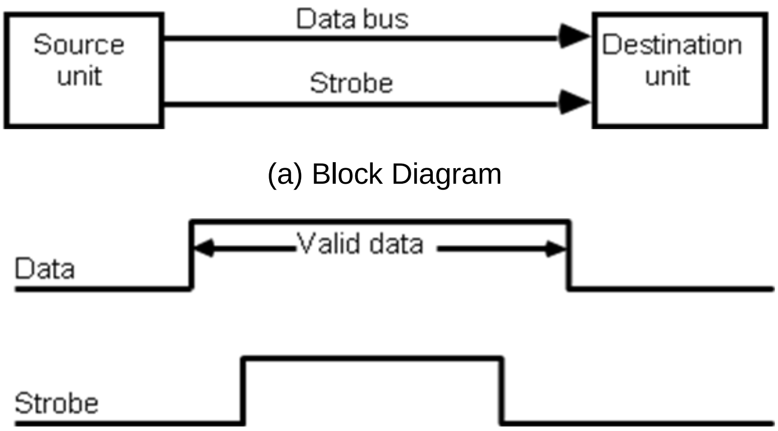
The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

2-3 Strobe Control

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure 2-1 (a) shows a source-initiated transfer. The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

As shown in the timing diagram of Fig. 2-1(b), the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valid data. New valid data will be available only after the strobe is enabled again.

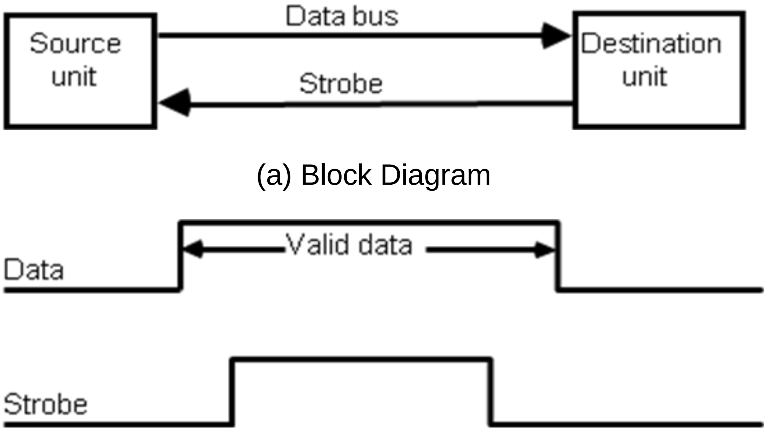
Figure 2-2 shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.



(a) Block Diagram

(b) Timing Diagram

Figure 2-1 Source-Initiated Strobe for Data Transfer



(a) Block Diagram

(b) Timing Diagram

Figure 2-2 Destination-Initiated Strobe for Data Transfer

In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data. For example, the strobe of Fig. 2-1 could be a memory-write control signal from the CPU to a memory unit. The source, being the CPU, places a word on the data bus and informs the memory unit, which is the destination, that this is a write operation. Similarly, the strobe of Fig. 2-2 could be a memory-read control signal from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.

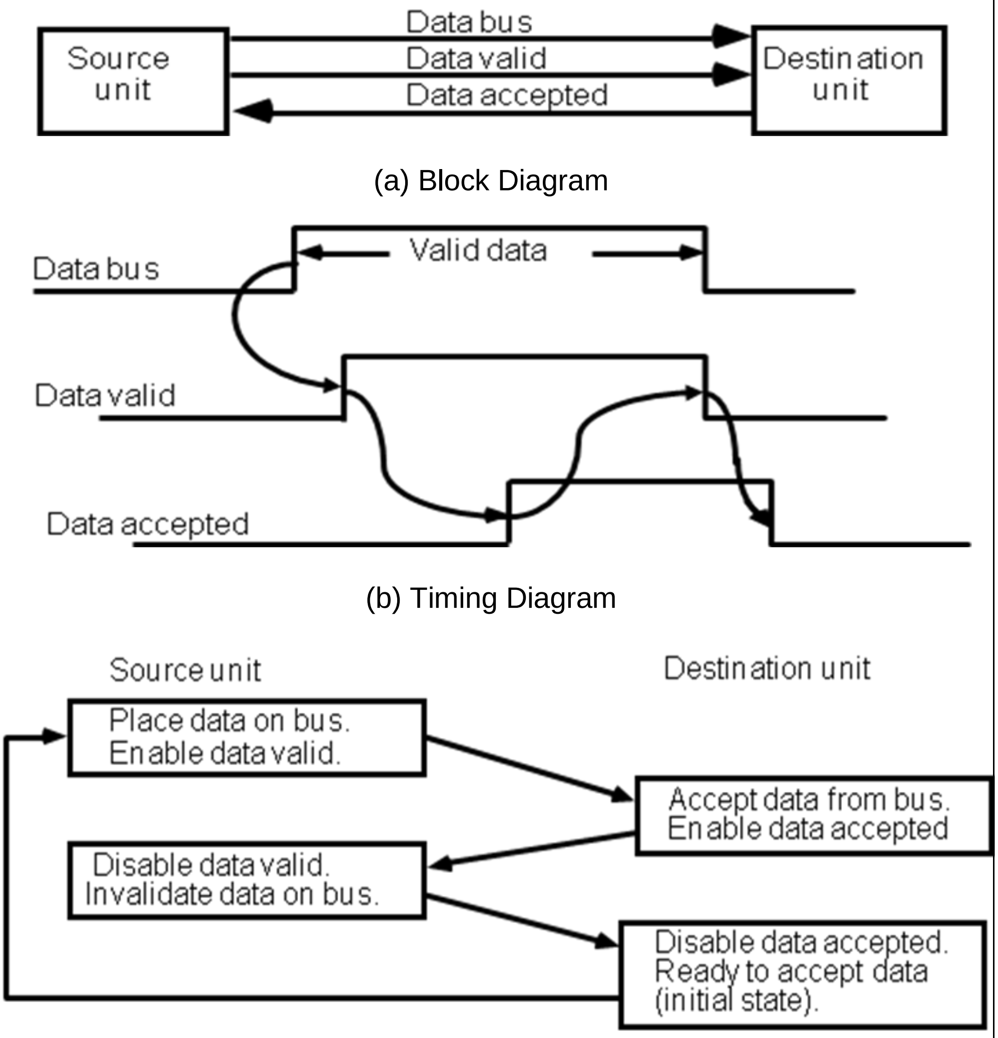
The transfer of data between the CPU and an interface unit is similar to the strobe transfer just described. Data transfer between an interface and an I/O device is commonly controlled by a set of handshaking lines.

2-4 Handshaking

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The basic principle of the two-wire handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

Figure 2-3 shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal. This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

The destination-initiated transfer using handshaking lines is shown in Fig. 2-4. Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case. Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted. In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.



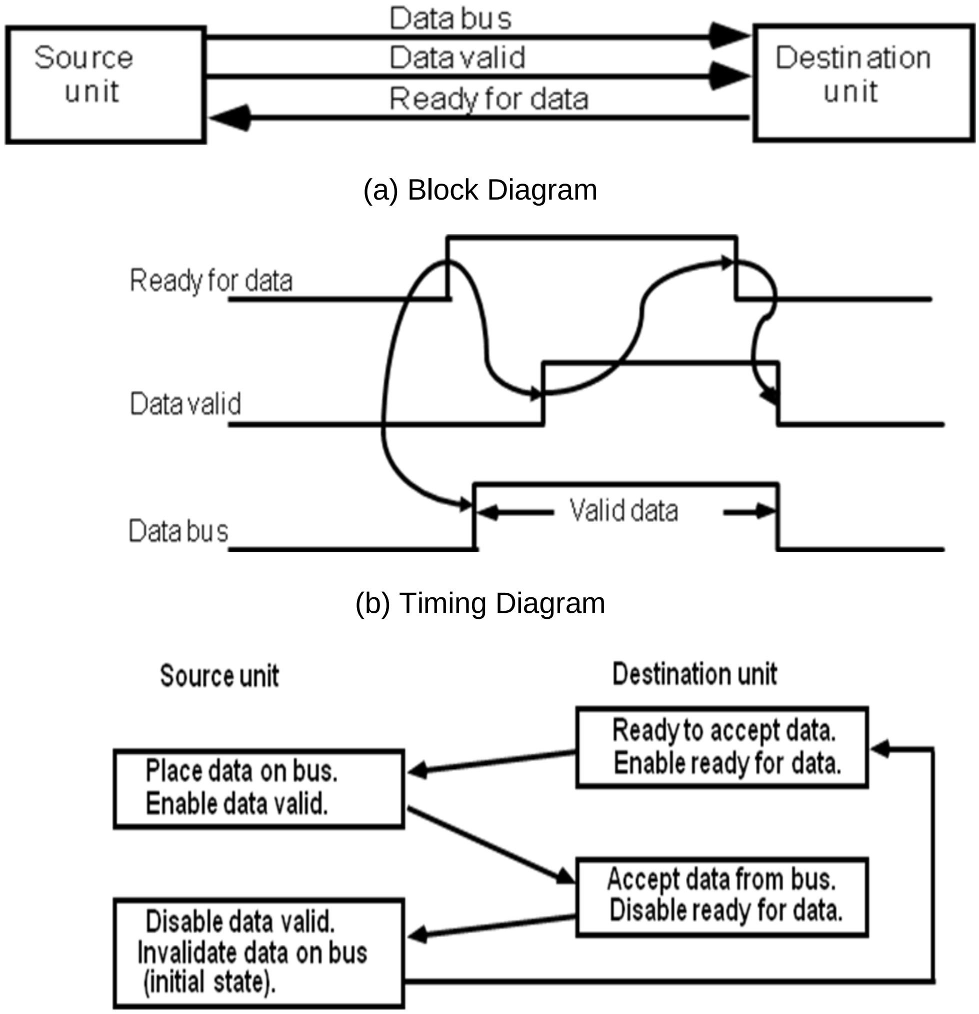
(a) Block Diagram

(b) Timing Diagram

(c) Sequence of Events

Figure 2-3 Source-Initiated transfer using handshaking

The handshaking scheme provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time. The timeout is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error has occurred. The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.


Data bus
Data valid
Ready for data
Source unit
Destination unit
Ready for data
Data valid
Data bus
Valid data
Source unit
Destination unit
Ready to accept data. Enable ready for data.
Place data on bus. Enable data valid.
Accept data from bus. Disable ready for data.
Disable data valid. Invalidate data on bus (initial state).

(a) Block Diagram

(b) Timing Diagram

(c) Sequence of Events

Figure 2-4 Destination-Initiated transfer using handshaking

2-5 Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n bit message must be transmitted through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence one at a time. This method requires the use of one pair of conductors or one conductor and common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive since it requires only one pair of conductors.

Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, where bits must be transmitted continuously to keep the clock frequency in both units synchronized with each other.

A serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in Fig. 2-5

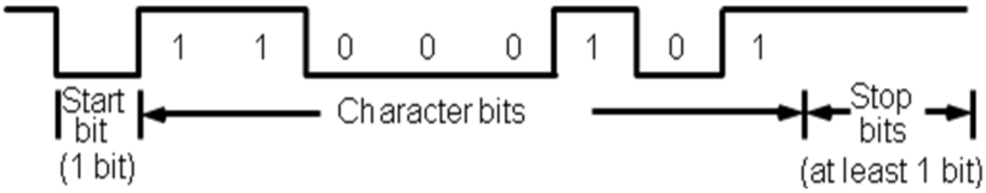


Figure 2-5 Asynchronous serial transmission

A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

Using these rules, the receiver can detect the start bit when the line goes from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of the character to signify the idle or wait state.

At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means each character takes 0.1sec for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09msec. The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format have a transfer rate of 110 baud.

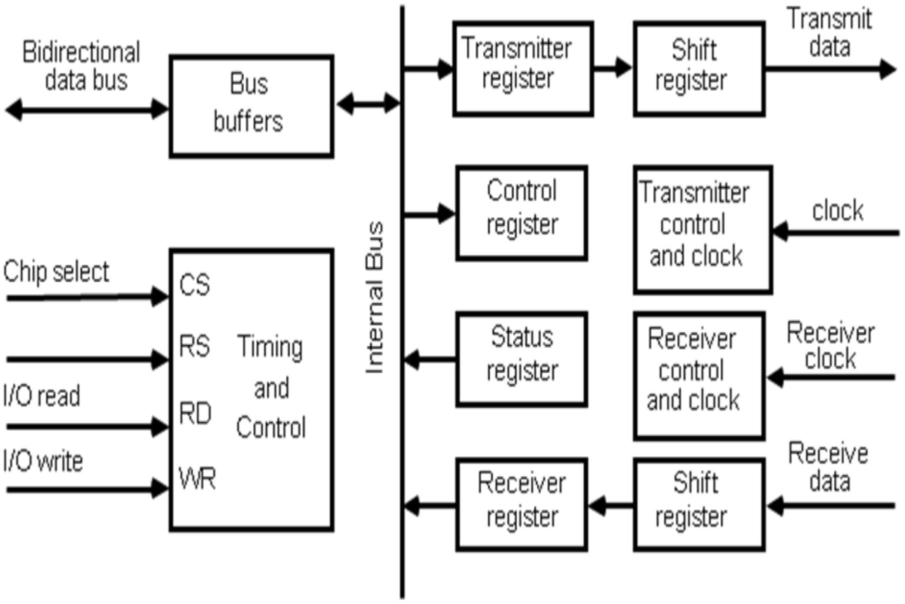
The terminal has a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially along a wire. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver. The transmitter accepts an 8-bit character from the computer and proceeds to send a serial 11-bit message into the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals. Such a circuit is called an asynchronous communication interface or a universal asynchronous receiver-transmitter (UART).

2-6 Asynchronous Communication Interface

The block diagram of an asynchronous communication interface is shown in Fig. 2-6. It functions as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into another shift register, and when a complete data byte is accumulated, it is transferred to the receiver register. The CPU can select the receiver register to read the byte through the data bus. The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred. The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD) and write (WR) controls. Two registers are write-only

NOTES

and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.



CS	RS	Oper.	Register selected
0	×	×	None
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Figure 2-6 Block Diagram of a typical asynchronous communication interface

The operation of the asynchronous communication interface is initialized by the CPU by sending a byte to the control register. The initialization procedure places the interface in a specific mode of operation as it defines certain parameters such as the baud rate to use, how many bits are in each character, whether to generate and check parity, and how many stop bits are appended to each character. Two bits in the status register are used as flags. One bit is used to indicate whether the transmitter register is empty and another bit is used to indicate whether the receiver register is full.

The operation of the transmitter portion of the interface is as follows. The CPU reads the status register and checks the flag to see if the transmitter register is empty. If it is empty the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit. The character is transferred in parallel from the transmitter register to the shift register and the appropriate number of stop bits is appended into the shift register. The transmitter register is then marked empty. The character can now be transmitted one bit at a time by shifting the data in the shift register at the specified baud rate. The CPU can transfer another character to the transmitter register after checking the flag in the status register. The interface is said to be double buffered because a new character can be loaded as soon as the previous one starts transmission.

The operation of the receiver portion of the interface is similar. The receive data input is in the 1-state when the line is idle. The receiver control monitors the receive-data line for a 0 signal to detect the occurrence of a start bit. Once a start bit has been detected, the incoming bits of the character are shifted into the shift register at the prescribed baud rate. After receiving the data bits, the interface checks for the parity and stop bits. The character without the start and stop bits is then transferred in parallel from the shift register to the receiver register. The flag in the status register is set to indicate that the receiver register is full. The CPU reads the status register and checks the flag, and if set, it reads the data from the receiver register.

The interface checks for any possible errors during transmission and sets appropriate bits in the status register. The CPU can read the status register at any time to check if any errors have occurred. Three possible errors that the interface checks during transmission are parity error, framing error, and overrun error. Parity error occurs if the number of 1's in the received data is not the correct parity. A framing error occurs if the right number of stop bits is not detected at the end of the received character. An overrun error occurs if the CPU does not read the character from the receiver register before the next one becomes available in the shift register. Overrun error results in a loss of characters in the received data stream.

2-7 First-In, First-Out Buffer

A first-in, first-out (FIFO) buffer is a memory unit that stores information in such a manner that the item first in is the item first out. A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input data and output data at two different rates and the output data are always in the same order in which the data entered the buffer. When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate. If the source unit is slower than the destination unit, the buffer can be filled with data at a slow rate and later emptied at the higher rate. If the source is faster than the destination, the FIFO is useful for those cases where the source data arrive in bursts that fill out the buffer but the time between bursts is long enough for the destination unit to empty some or all the information from the buffer. Thus a FIFO buffer can be useful in some applications when data are transferred asynchronously. It piles up data as they come in and gives them away in the same order when the data are needed.

The logic diagram of a typical 4X4 FIFO buffer is shown in Fig. 2-7. It consists of four 4-bit registers RI, I = 1,2, 3, 4, and a control register with flip-flops F_i, i = 1, 2, 3,4, one for each register. The FIFO can store four words of four bits each. The number of bits per word can be increased by increasing the number of bits in each register and the number of words can be increased by increasing the number of registers.

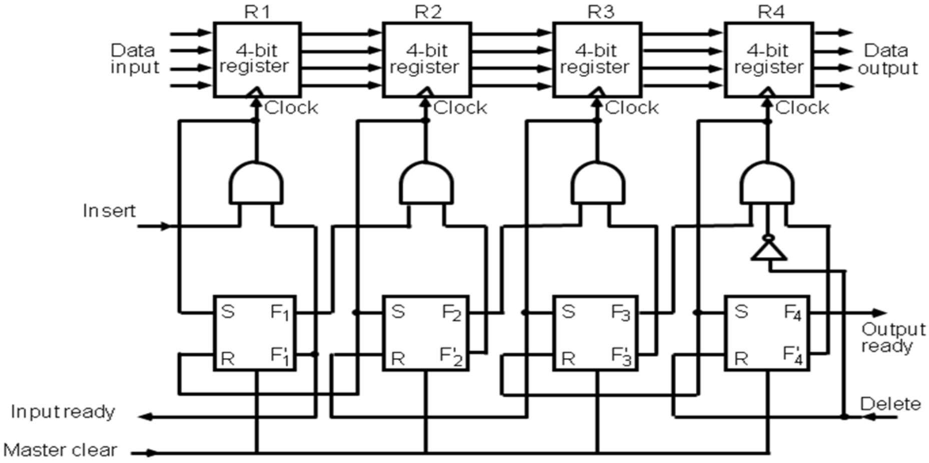


Figure 2-7 Circuit diagram of 4 X 4 FIFO buffer

A flip-flop F_i in the control register that is set to 1 indicates that a 4-bit data word is stored in the corresponding register RI. A 0 in F_i indicates that the corresponding register does not contain valid data. The control register directs the movement of data through the registers. Whenever the F, bit of the control register is set F_i = 1) and the F_{i+1} bit is reset (F'_{i+1} = 1), a clock is generated causing register R(I+ 1) to accept the data from register RI. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. These ripple-through operation stops when the data reach a register RI with the next flip-flop F_{i+1} being set to 1, or at the last register R4. An overall master clear is used to initialize all control register flip-flops to 0.

Data are inserted into the buffer provided that the input ready signal is enabled. This occurs when the first control flip-flop F_1 is reset, indicating that register Rl is empty. Data are loaded from the input lines by enabling the clock in Rl through the insert control line. The same clock sets F_1 which disables the input ready control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that R2 is empty. The data in Rl are transferred into R2 and F_1 is cleared. This enables the input ready line, indicating that the inputs are now available for another data word. If the FIFO is full, F_1 remains set and the input ready line stays in the 0 state. Note that the two control lines input ready and insert constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The output ready control line is enabled when the last control flip-flop F_4 is set, indicating that there are valid data in the output register R4:. The output data from R4: are accepted by a destination unit, which then enables the delete control signal. This resets F_4 causing output ready to disable, indicating that the data on the output are no longer valid. Only after the delete signal goes back to 0 can the data from R3 move into R4. If the FIFO is empty, there will be no data in R3 and F_4 will remain in the reset state. Note that the two control lines output ready and delete constitute a source-initiated pair of handshake lines.

2-8 Summary

In this lesson, we have introduced communication of peripheral devices to the system. We have mainly focus on asynchronous transfer in between of the devices using hand shaking signals. In an asynchronous transfer, there is no common clock and the master and slave can follow some sort of acknowledgement protocol during data transfer sequence.

2-9 Keywords

Strobe Control: One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.

Handshaking: The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.

Asynchronous Communication: The two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses.

Asynchronous Communication: binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.

Chip select: The chip select (CS) input is used to select the interface through the address bus.

2-10 PROBLEMS

1. A commercial interface unit uses different names for the handshake lines associated with the transfer of data from the I/O device into the interface unit. The interface input handshake line is labeled STB (strobe), and the interface output handshake line is labeled IBF (input buffer full). A low-level signal on STB loads data from the I/O bus into the interface data register. A high-level signal on IBF indicates that the data item has been accepted by the interface. IBF goes low after an I/O read signal from the CPU when it reads the contents of the data register.

a. Draw a block diagram showing the CPU, the interface, and the I/O device together with the pertinent interconnections among the three units.

b**.** Draw a timing diagram for the handshaking transfer. c. Obtain a sequence-of-events flowchart for the transfer from the device to the interface and from the interface to the CPU.

2. A CPU with a 20-MHz clock is connected to a memory unit whose access time is 40 ns. Formulate a read and write timing diagrams using a READ strobe and a WRITE strobe. Include the address in the timing diagram.

3. The asynchronous communication interface shown in Fig. 2-6 is connected between a CPU and a printer. Draw a flowchart that describes the sequence of operations in the transmitter portion of the interface when the CPU sends characters to be printed.

4. Give at least six status conditions for the setting of individual bits in the status register of an asynchronous communication interface.

5. How many bits are there in the transmitter shift register of Fig. 2-7 when the interface is attached to a terminal that needs one stop bit? List the bits in the shift register when the letter W is transmitted using ASCII with even parity.

6. How many characters per second can be transmitted over a 1200-baud line in each of the following modes? (Assume a character code of eight bits.)

   a. Synchronous serial transmission.

   b. Asynchronous serial transmission with two stop bits.

   c. Asynchronous serial transmission with one stop bit.

7. Information is inserted into a FIFO buffer at a rate of m bytes per second. The information is deleted at a rate of n byte per second. The maximum capacity of the buffer is k bytes.

   a. How long does it take for an empty buffer to fill up when m > n?

   b. How long does it take for a full buffer to empty when m < n?

   c. Is the FIFO buffer needed if m = n?

2-11 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes,.

3. MODES OF DATA TRANSFER

Structure

Objectives

At the end of this lesson you should be able to:

- Discuss about the Programmed I/O
- Discuss about the Interrupt-initiated I/O

3-1 Introduction

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O

2. Interrupt-initiated I/O

3. Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks. When the transfer is made, the DMA requests memory cycles the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/0-memory transfers are infrequent compared to processor access to memory. DMA transfer is discussed in more detail in lesson 4.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in lesson 5

3-2 Example of Programmed I/O

In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 3-1. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an F or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until this data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig. 2-3 in lesson 2.

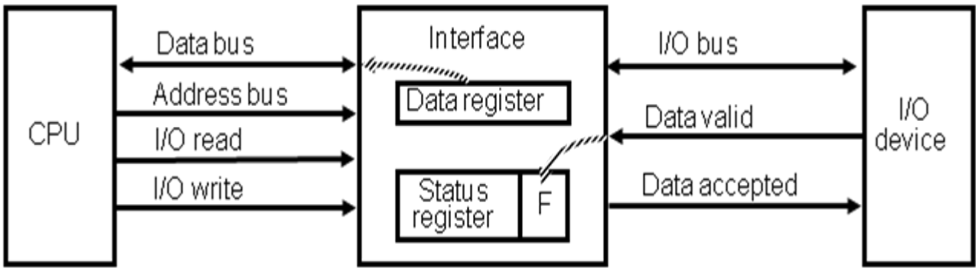


Figure 3-1 Data transfer from I/O device to CPU

A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and(checking the value of the flag bit. If the flag is equal to 1, the CPU reads the(data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. 3-2. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.

2. Check the status of the flag bit and branch to step 1 if not set or to step 2 if set.

3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer.

The programmed I/O method is particularly useful in small low-speed computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the two instructions that read the status register and check the flag in 1µs. Assume that the input device transfers its data at an average rate of 100 bytes per second. This is equivalent to on byte every 10,000 µs. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

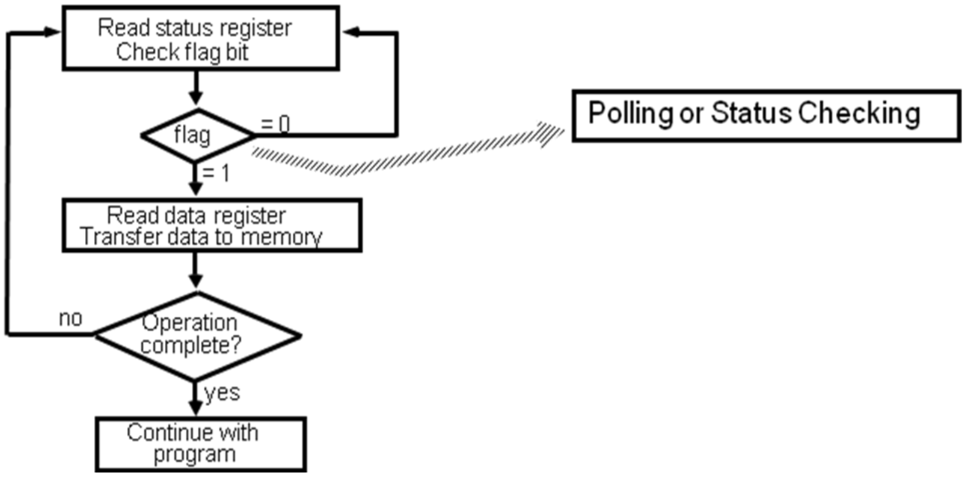


Figure 3-2 Flowchart for CPU program to input data

3-3 Interrupt-Initiated I/O

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to (a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this One is called vectored interrupt and the other, non-vectored interrupt. In a non-vectored interrupt, the branch address is assigned to a fixed location in memory, In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

3-4 Software Considerations

The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers. In interrupt controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.

Software control of input-output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

3-5 Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on. Thus the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest-priority device among all devices that interrupted the computer. The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy-chaining method.

3-5-1 Daisy-Chaining Priority

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in die chain. This method of connection between three devices and die CPU is shown in Fig. 3-3. The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in die CPU. When no interrupts arc pending, the interrupt line stays in die high-level state and no interrupts are recognized by die CPU. This is equivalent to a negative-logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its PI(priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the

PO output. It then proceeds to insert its own interrupt vector address (VAD) into die data bus for the CPU to use during the interrupt cycle.

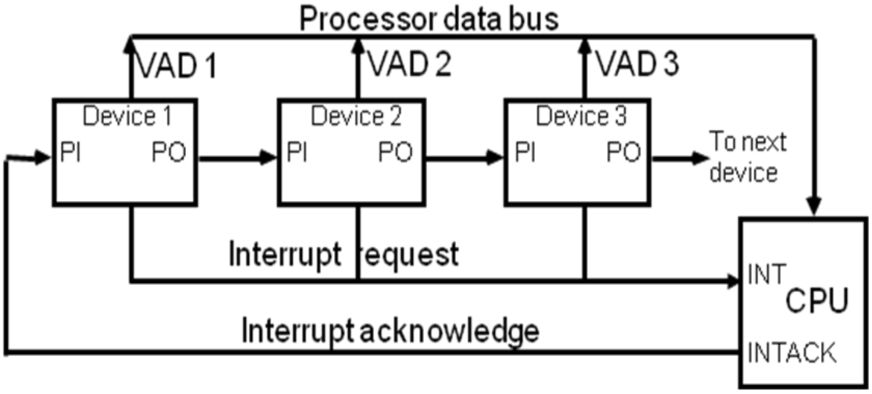


Figure 3-3 Daisy-chain priorities interrupt

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device mat the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position; the lower is its priority.

Figure 3-4 shows the internal logic that must be included within each device when connected in the daisy-chaining scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI= 1 and RF= 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI = 1 and RF= 1. This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

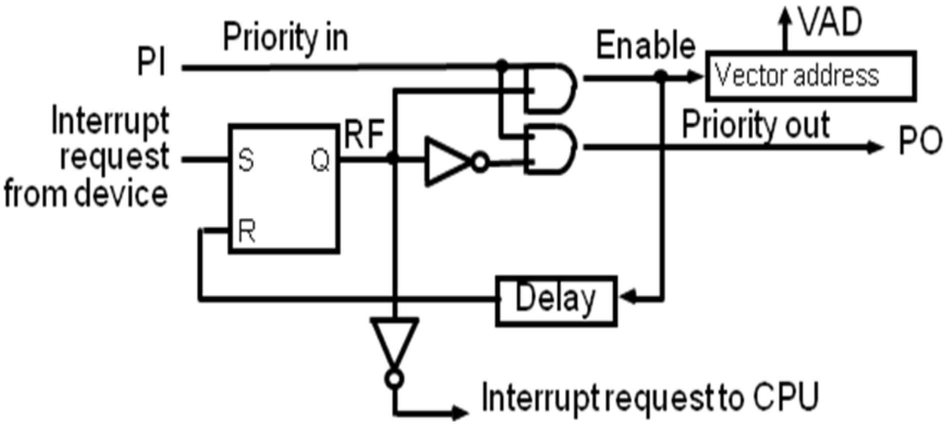


Figure 3-4 One stage of the daisy-chain priority arrangement

3-5-2 Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Fig. 3-5. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

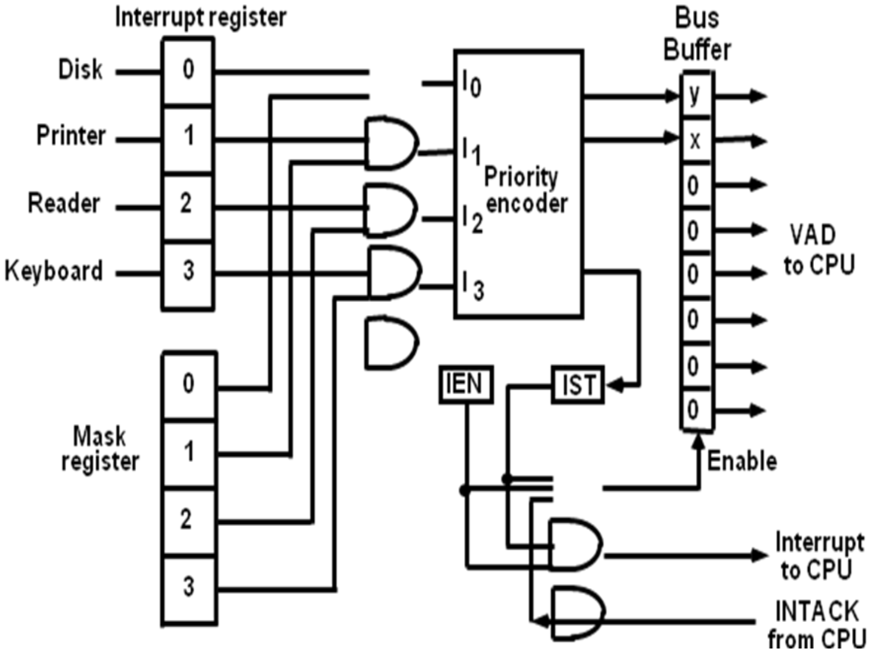


Figure 3-5 Priority interrupt hardware

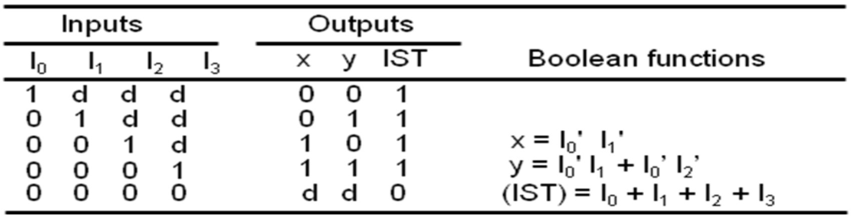
Another output from the encoder sets an interrupt status flip-flop 1ST when an interrupt that is not masked occurs. The interrupt enable flip-flop lEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

3-5-3 Priority Encoder

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 3-1. The X's in the table designate don't-care conditions. Input I_0 has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output *xy* = 00. I_1 has the next priority level. The output is 01 if I_1 = 1 provided that I_0 = 0, regardless of the values of the other two lower-priority inputs. The output for I_2 is generated only if higher-priority inputs are 0, and so on down the priority level. The interrupt status *IST is set* only when one or more inputs are equal to 1. If all inputs are 0, IST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when *IST*= 0. The Boolean functions listed in the table specify the internal logic of the encoder. Usually, a computer will have more than four interrupt sources. A priority encoder with eight inputs, for example, will generate an output of three bits.

The output of the priority encoder is used to form part of the vector address for each interrupt source. The other bits of the vector address can be assigned any value. For example, the vector address can be formed by appending six zeros to the *x* and *y* outputs of the encoder. With this choice the interrupt vectors for the four I/O devices are assigned binary numbers 0, 1, 2, and 3.

TABLE 3-1 Priority Encoder Truth Table

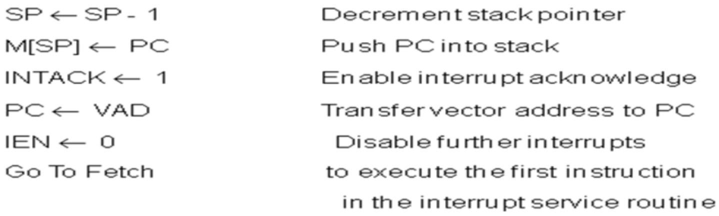


Inputs				Outputs			
I_0	I_1	I_2	I_3	x	y	IST	Boolean functions
1	d	d	d	0	0	1	
0	1	d	d	0	1	1	
0	0	1	d	1	0	1	$x = I_0' \ I_1'$
0	0	0	1	1	1	1	$y = I_0' I_1 + I_0' I_2'$
0	0	0	0	d	d	0	$(IST) = I_0 + I_1 + I_2 + I_3$

3-5-4 Interrupt Cycle

The interrupt enable flip-flop *IEN* shown in Fig. 3-5 can be set or cleared by program instructions. When *IEN is* cleared, the interrupt request coming from IST is neglected by the CPU. The program-controlled *IEN bit* allows the programmer to choose whether to use the interrupt facility. If an instruction to clear *IEN has* been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set *IEN* indicates that the interrupt facility will be used while the current program is running. Most computers include internal hardware that clears *IEN to 0* every time an interrupt is acknowledged by the processor.

At the end of each instruction cycle the CPU checks *IEN and* the interrupt signal from *IST.* If either is equal to 0, control continues with the next instruction. If both *IEN and IST are* equal to 1, the CPU goes to an interrupt cycle. During the interrupt cycle the CPU performs the following sequence of microoperations:



SP ← SP - 1	Decrement stack pointer
M[SP] ← PC	Push PC into stack
INTACK ← 1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
IEN ← 0	Disable further interrupts
Go To Fetch	to execute the first instruction in the interrupt service routine

The CPU pushes the return address from *PC* into the stack. It then acknowledges the interrupt by enabling the *INTACK* line. The priority interrupt unit responds by placing a unique interrupt vector into the CPU data bus. The CPU transfers the vector address into PC and clears *IEN* prior to going to the next fetch phase. The instruction read from memory during the next fetch phase will be the one located at the vector address.

3-5-5 Software Routines

A priority interrupt system is a combination of hardware and software techniques. So far we have discussed the hardware aspects of a priority interrupt system. The computer must also have software routines for servicing the interrupt requests and for controlling the interrupt hardware registers. Figure 3-6 shows the programs that must reside in memory for handling the interrupt system. Each device has its own service program that can be reached through a jump (JMP) instruction stored at the assigned vector address. The symbolic name of each routine represents the starting address of the service program the stack shown in the diagram is used for storing the return address after each interrupt.

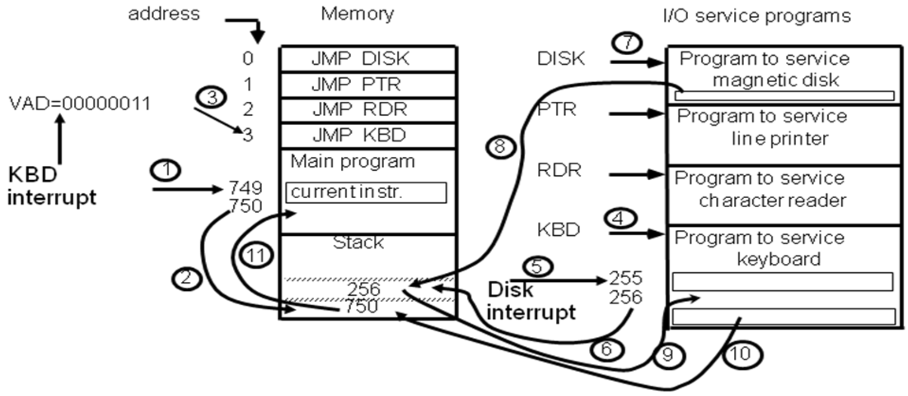


Figure 3-6 Programs in memory for servicing interrupts

To illustrate with a specific example assume that the keyboard sets it interrupt bit while the CPU is executing the instruction in location 749 of the main program. At the end of the instruction cycle, the computer goes to an interrupt cycle. It stores the return address 750 in the stack and then accepts the vector address 00000011 from the bus and transfers it to *PC.* The instruction in location 3 is executed next, resulting in transfer of control to the KBD routine. Now suppose that the disk sets its interrupt bit when the CPU is executing the instruction at address 255 in the KBD program. Address 256 pushed into the stack and control is transferred to the DISK service program. The last instruction in each routine is a return from interrupt instruction when the disk service program is completed, the return instruction pops through stack and places 256 into PC. This returns control to the KBD routine to continue servicing the keyboard. At the end of the KBD program, the last instruction pops the stack and returns control to the main program at address 750. Thus, a higher-priority device can interrupt a lower-priority device. It is assumed that the time spent in servicing the high-priority interrupt is short compared to the transfer rate of the low-priority device so that no loss of information takes place.

3-5-6 Initial and Final Operations

Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system.

Remember that the interrupt enable *IEN is* cleared at the end of an interrupt cycle. This flip-flop must be set again to enable higher-priority interrupt requests, but not before lower-priority interrupts are disabled. The initial sequence of each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear lower-level mask register bits.

2. Clear interrupt status bit *IST.*

3. Save contents of processor registers.

4. Set interrupt enable bit *IEN.*

5. Proceed with service routine.

The lower-level mask register bits (including the bit of the source that interrupted) are cleared to prevent these conditions from enabling the interrupt. Although lower-priority interrupt sources are assigned to higher-numbered bits in the mask register, priority can be changed if desired since the programmer can use any bit configuration for the mask register. The interrupt status bit must be cleared so it can be set again when a higher-priority interrupt occurs. The contents of processor registers are saved because they may be needed by the program that has been interrupted after control returns to it. The interrupt enable *IEN is* then set to allow other (higher-priority) interrupts and the computer proceeds to service the interrupt request.

The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear interrupt enable bit *IEN.*

2. Restore contents of processor registers.

3. Clear the bit in the interrupt register belonging to the source that has been serviced.

4. Set lower-level priority bits in the mask register.

5. Restore return address into *PC* and set *IEN.*

The bit in the interrupt register belonging to the source of the interrupt must be cleared so that it will be available again for the source to interrupt. The lower-priority bits in the mask register (including the bit of the source being interrupted) are set so they can enable the interrupt. The return to the interrupted program is accomplished by restoring the return address to *PC.* Note that the hardware must be designed so that no interrupts occur while executing steps 2 through 5; otherwise, the return address may be lost and the information in the mask and processor registers may be ambiguous if an interrupt is acknowledged while executing the operations in these steps. For this reason *IENis* initially cleared and then set after the return address is transferred into *PC.*

The initial and final operations listed above are referred to as *overhead* operations or *housekeeping* chores. They are not part of the service program proper but are essential for processing interrupts. All overhead operations can be implemented by software. This is done by inserting the proper instructions at the beginning and at the end of each service routine. Some of the overhead operations can be done automatically by the hardware. The contents of processor registers can be pushed into a stack by the hardware before branching to the service routine. Other initial and final operations can be assigned to the hardware. In this way, it is possible to reduce the time between receipt of an interrupt and the execution of the instructions that service the interrupt source.

3-6 Summary

One of the major features in a computer system is its ability to exchange data with other devices and to allow the user to interact with the system. This lesson focused on the I/O system and the way the processor and the I/O devices exchange data in a computer system. The lesson described three ways of organizing I/O: programmed I/O, interrupt-driven I/O, and DMA. In programmed I/O, the CPU handles the transfers, which take place between registers and the devices. In interrupt-driven I/O, CPU handles data transfers and an I/O module is running concurrently. We also studied two methods for synchronization: polling and interrupts. In polling, the processor polls the device while waiting for I/O to complete. Clearly processor cycles are wasted in this method. Using interrupts, processors are free to switch to other tasks during I/O. Devices assert interrupts when I/O is complete. Interrupts incurs some delay penalty.

3-7 Keywords

Programmed I/O: Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program.

Interrupt: An interrupt is an event inside a computer system requiring some urgent action by the CPU

DMA: A dedicated hardware unit such as data channel or direct memory access (DMA) controller takes care of controlling the transfer of data between memory and I/O controller

Priority interrupts: Interrupt requests from various sources are connected as input to the interrupt controller. As soon as the interrupt controller senses the presence of any one or more interrupt requests, immediately it issues an interrupt signals to the processor.

Polling: One-by-one transfer.

3-8 Exercise

1. Show a block diagram similar to Fig. 3-1 for the data transfer from a CPU to an interface and then to an I/O device. Determine a procedure for setting and clearing the flag bit.

2. Using the configuration established in Prob.1, obtain a flowchart (similar to Fig. 3-2) for the CPU program to output data.

3. What is the basic advantage of using interrupt-initiated data transfer over transfer under program control without an interrupt?

4. In most computers an interrupt is recognized only after the execution of the instruction. Consider the possibility of acknowledging the interrupt at any time during the execution of the instruction. Discuss the difficulty that may arise.

5. What happens in the daisy-chain priority interrupt shown in Fig. 3-3 when device 1 requests an interrupt after device 2 has sent an interrupt request to the CPU but before the CPU responds with the interrupt acknowledge?

6. Consider a computer without priority interrupt hardware. Any one of many sources can interrupt the computer and any interrupt request results in storing the return address and branching to a common interrupt routine. Explain how a priority can be established in the interrupt service program.

7. Using combinational circuit design techniques, derive the Boolean expressions listed in Table 3-1 for the priority encoder. Draw the logic diagram of the circuit.

8. What programming steps are required to check when a source interrupts the computer while it is still being serviced by a previous interrupt request from the same source?

3-7 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes.

4. DIRECT MEMORY ACCESS (DMA)

Structure

Objectives

At the end of this lesson you should be able to:

- Describe the DMA Controller
- Describe the DMA Transfer

4-1 Introduction

The transfer of data between fast storage devices such is magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 4-1 shows two control signals in the CPU that facilitate the DMA transfer. The *bus request (BR)* input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance (see Sec. 4-3). The CPU activates the *bus grant (BG)* output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

When the DMA takes control of the bus system, it communicates directly with the memory. The transfer can be made in several ways. In DMA *burst transfer,* a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called *cycle stealing* allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to "steal" one memory cycle.

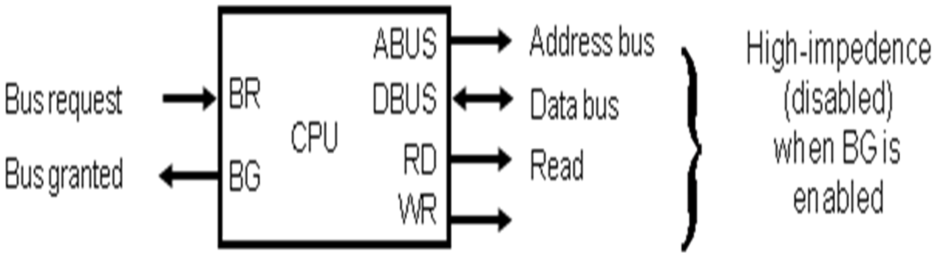


Figure 4-1 CPU bus signals for DMA transfer

4-2 DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The addresses register and address lines are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 4-2 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the *DS* (DMA select) and *RS* (register select) inputs. The *RD* (read) and *WR* (write) inputs are bidirectional. When the *BG* (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When *BG* = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the *RD* or *WR* control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

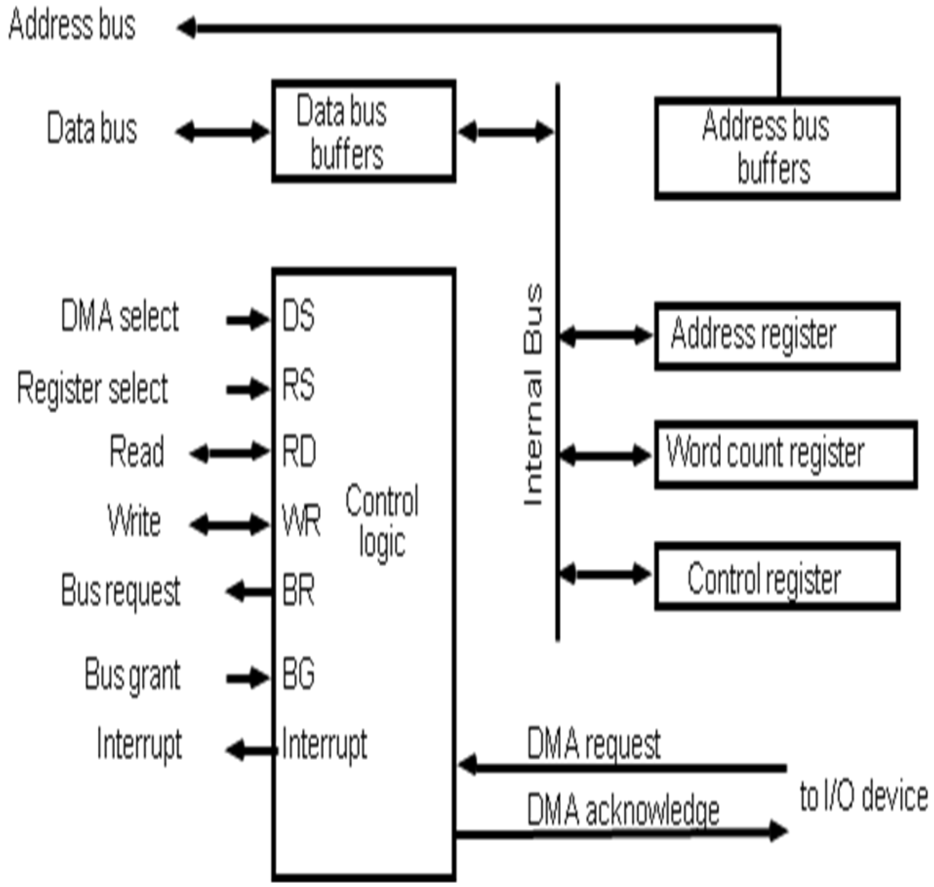


Figure 4-2 Block diagram of DMA controller

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)

NOTES

2. The word count, which is the number of words in the memory block

3. Control to specify the mode of transfer such as read or write

4. A control to start the DMA transfer

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

4-3 DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Fig. 4-3. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and *R S* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device. Note that the *RD* and *WR* lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the *BG* line. When *BG* = 0, the *RD* and *WR* are input lines allowing the CPU to communicate with the internal DMA registers. When *BG* = 1, the *RD* and *WR* are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

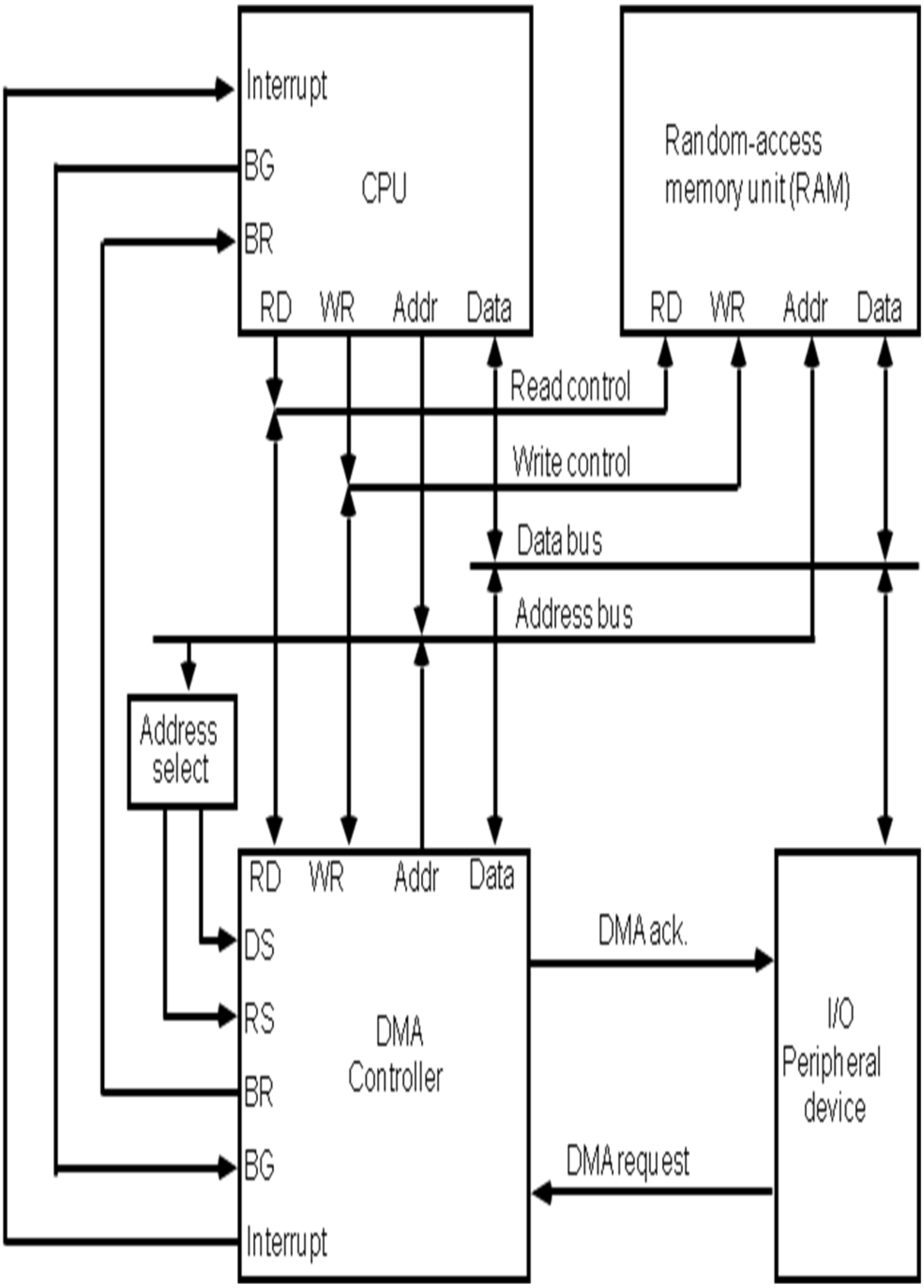


Figure 4-3 DMA transfer in a computer system

For each word that is transferred, the DMA increments its address registers and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disables the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were transferred successfully. The CPU can read this register at any time to check the number of words already transferred.

A DMA controller may have more than one channel. In this case, each channel has a request and acknowledge pair of control signals which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

4-4 Summary

This lesson was devoted mainly towards I/O of computer system we have discussed about the DMA module. In a DMA based I/O, a special unit called Direct Memory Access controller is used as an intermediary between the I/O interface unit and the main memory. The CPU after issuing the I/O command proceeds with its activity. When I/O is ready, DMA controller requests CPU to yield the data and address buses to it. The DMA controller now directly sends/receives data to/from main memory. CPU has no role to play in data transfer other than yielding the buses.

4-5 Keywords

Bus request: The bus request (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses.

Bus grant: The CPU activates the bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state

.**Burst transfer:** A block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses.

Cycle stealing: the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU.

4-6 PROBLEMS

1. Why are the read and write control lines in a DMA controller bidirectional? Under what condition and for what purpose are they used as inputs? Under what condition and for what purpose are they used as outputs?

2. It is necessary to transfer 256 words from a magnetic disk to a memory section starting from address 1230. The transfer is by means of DMA as shown in Fig. 4-3.

   a. Give the initial values that the CPU must transfer to the DMA controller.

   b. Give the step-by-step account of the actions taken during the input of the first two words.

3. A DMA controller transfers 16-bit words to memory using cycle stealing. The words are assembled from a device that transmits characters at a rate of 2400 characters per second. The CPU is fetching and executing instructions at an average rate of 1 million instructions per second. By how much will the CPU be slowed down because of the DMA transfer?

4. Why does DMA have priority over the CPU when both request a memory transfer?

4-7 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes

5. INPUT-OUTPUT PROCESSOR (IOP)

&

SERIAL COMMUNICATION

Structure

Objectives

At the end of this lesson you should be able to:

- Define an input-output processor; and
- Identify the Serial Communication.

5-1 Introduction

Instead of having each interface communicate with the CPU, a computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An input-output processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit, and a number of processors comprised of the CPU and one or more IOPs. Each IOP takes care of input and output tasks, relieving the CPU from the housekeeping chores involved in I/O transfers. A processor that communicates with remote terminals over telephone and other communication media in a serial fashion is called a data communication processor (DCF).

I/O processing

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in Fig. 5-1. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit. The CPU is usually assigned the task of initiating the I/O program. From then on the IOP operates independent of the CPU and continues to transfer data from external devices and memory.

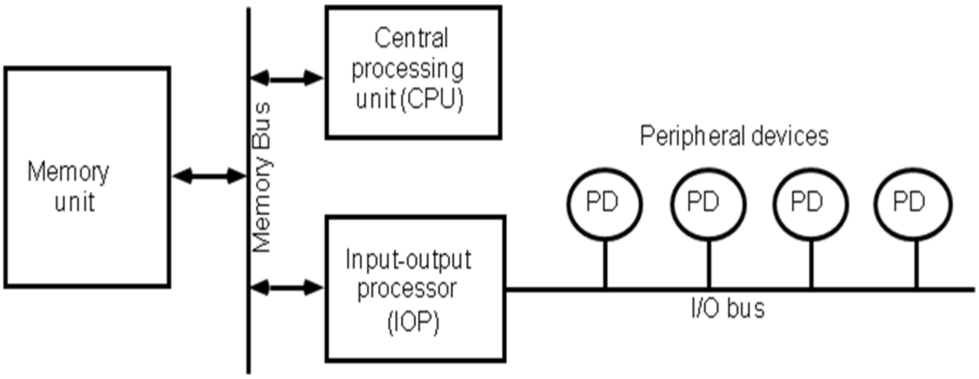


Figure 5-1 Block diagram of a computer with I/o processor

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in me IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. Communication with the memory is similar to the direct memory access method. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In very-large-scale computers, each processor is independent of all others and any one processor can initiate an operation. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations, but I/O instructions are executed in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

Instructions that are read from memory by an IOP are sometimes called *commands,* to distinguish them from instructions that are read by the CPU. Otherwise, an instruction and a command have similar functions. Commands are prepared by experienced programmers and are stored in memory. The command words constitute the program for the IOP. The CPU informs the IOP where to find the commands in memory when it is time to execute the I/O program.

5-2 CPU-IOP Communication

The communication between CPU and IOP may take different forms, depending on the particular computer considered. In most cases the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate by a specific example the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Fig. 5-2. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the 10P and I/O device, such as 10P overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the 10P where to find it? Program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the 10P terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the 10P. The IOP responds by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or if any errors occurred during the transfer. From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.

The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory. It is not possible to saturate the memory by I/O devices in most systems, as the speed of most devices is much slower than the CPU. However, some very fast units, such as magnetic disks, can use an appreciable number of the available memory cycles. In that case, the speed of the CPU may deteriorate because it will often have to wait for the 10P to conduct memory transfers.

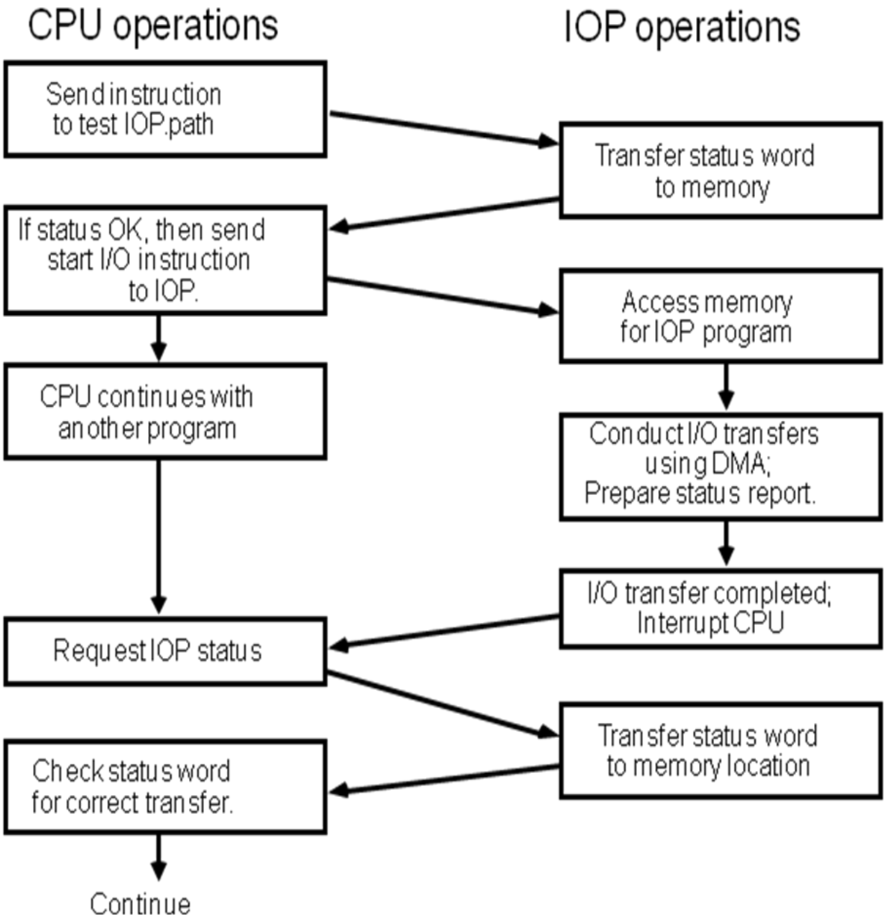

CPU operations
IOP operations
Send instruction to test IOP.path
Transfer status word to memory
If status OK, then send start I/O instruction to IOP.
Access memory for IOP program
CPU continues with another program
Conduct I/O transfers using DMA; Prepare status report.
I/O transfer completed; Interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer.
Continue

Figure 5-2 CPU-IOP communication

5-3 Serial Communication

A data communication processor is an I/O processor that distributes and collects data from many remote terminals connected through telephone and other communication lines. It is a specialized I/O processor designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

The most striking difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are transferred in a serial fashion with the result that the transfer rate is much slower. The task of the data communication processor is to transmit and collect digital information to and from each terminal, determine if the information is data or control and respond to all requests according to predetermined established procedures. The processor, obviously, must also communicate with the CPU and memory in the same manner as any I/O processor.

The way that remote terminals are connected to a data communication processor is via telephone lines or other public or private communication facilities. Since telephone lines were originally designed for voice communication and computers communicate in terms of digital signals, some form of conversion must be used. The converters are called *data sets, acoustic couplers,* or *modems* (from "modulator-demodulator"). A modem converts digital signals into audio tones to be transmitted over telephone lines and also converts audio tones from the line to digital signals for machine use. Various modulation schemes as well as different grades of communication media and transmission speeds are used. A communication line may be connected to a synchronous or asynchronous interface, depending on the transmission method of the remote terminal. An asynchronous interface receives serial data with start and stop bits in each character as shown in Fig. 2-5. This type of interface is similar to the asynchronous communication interface unit presented in Fig. 2-6.

Synchronous transmission does not use start-stop bits to frame characters and therefore makes more efficient use of the communication link. High-speed devices use synchronous transmission to realize tills efficiency. The modems used in synchronous transmission have internal clocks that are set to the frequency that bits are being transmitted in the communication line. For proper operation, it is required that the clocks in the transmitter and receiver modems remain synchronized at all times. The communication line, however, contains only the data bits from which the clock information must be extracted. Frequency synchronization is achieved by the receiving modem from the signal transitions that occur in the received data. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock at the frequency of the incoming bit stream. The modem transfers the received data together with the clock to the interface unit. The interface or terminal on the transmitter side also uses the clock information from its modem. In this way, the same bit rate is maintained in both transmitter and receiver.

Contrary to asynchronous transmission, where each character can be sent separately with its own start and stop bits, synchronous transmission must send a continuous message in order to maintain synchronism. The message consists of a group of bits transmitted sequentially as a block of data. The entire block is transmitted with special control characters at the beginning and end of the block. The control characters at the beginning of the block supply the information needed to separate the incoming bits into individual characters.

One of the functions of the data communication processor is to check for transmission errors. An error can be detected by checking the parity in each character received. Another procedure used in asynchronous terminals involving a human operator is to *echo* the character. The character transmitted from the keyboard to the computer is recognized by the processor and retransmitted to the terminal printer. The operator would realize that an error occurred during transmission if the character printed is not the same as the character whose key he has struck.

In synchronous transmission, where an entire block of characters is transmitted, each character has a parity bit for the receiver to check. After the entire block is sent, the transmitter sends one more character that constitutes a parity over the length of the message. This character is called a longitudinal redundancy check (LRC) and is the accumulation of the exclusive-OR of all transmitted characters. The receiving station calculates the LRC as it receives characters and compares it with the transmitted LRC. The calculated and received LRC should be equal for error-free messages. If the receiver finds an error in the transmitted block, it informs the sender to retransmit the same block once again. Another method used for checking errors in transmission is the cyclic redundancy check (CRC). This is a polynomial code obtained from the message bits by passing them through a feedback shift register containing a number of exclusive-OR gates. This type of code is suitable for detecting burst errors occurring in the communication channel.

5-3-1 Data can be transmitted between two points in three different modes:

Simplex, half-duplex, or full-duplex. A *simplex* line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are radio and television broadcasting.

A *half-duplex* transmission system is one that is capable of transmitting in both directions but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode. A common situation is for one modem to act as the transmitter and the other as the receiver. 'When transmission in one direction is completed, the role of the modems is reversed to enable transmission in the reverse direction. The time required to switch a half-duplex line from one direction to the other is called the turnaround time.

A *full-duplex* transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission. Alternatively, a two-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two non-overlapping frequency bands to create separate receive and transmit channels in the same physical pair of wires.

The communication lines, modems, and other equipment used in the transmission of information between two or more stations is called a *data link.* The orderly transfer of information in a data link is accomplished by means of a *protocol.* A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of information. The purpose of a data link protocol is to establish and terminate a connection between two stations, to identify the sender and receiver, to ensure that all messages are passed correctly without errors, and to handle all control functions involved in a sequence of data transfers. Protocols are divided into two major categories according to the message-framing technique used. These are character-oriented protocol and bit-oriented protocol.

5-3-2 Character-Oriented Protocol

The character-oriented protocol is based on the binary code of a character set. The code most commonly used is ASCII (American Standard Code for Information Interchange). It is a 7-bit code with an eighth bit used for parity. The code has 128 characters, of which 95 are graphic characters and 33 are control characters. The graphic characters include the upper- and lowercase letters, the ten numerals, and a variety of special symbols. A list of the ASCII characters can be found in Table 1-1. The control characters are used for the purpose of routing data, arranging the test in a desired format, and for the layout of the printed page. The characters that control the transmission are called *communication control* characters. These characters are listed in Table 1-1. Each character has a 7-bit code and is referred to by a three-letter symbol. The role of each character in the control of data transmission is stated briefly in the function column of the table.

The SYN character serves as synchronizing agent between the transmitter and receiver. When the 7-bit ASCII code is used with an odd-parity bit in the most significant position, the assigned SYN character has the 8-bit code 00010110 which has the property that, upon circular shifting, it repeats itself only after a full 8-bit cycle. When the transmitter starts sending 8-bit characters, it sends a few characters first and then sends the actual message. The initial continuous string of bits accepted by the receiver is checked for a SYN character. In other words, with each clock pulse, the receiver checks the last eight bits received. If they do not match the bits of the SYN character, the receiver accepts the next bit, rejects the previous high-order bit, and again checks the last eight bits received for a SYN character. This is repeated after each clock puise and bit received until a SYN character is recognized. Once a SYN character is detected, the receiver has framed a character. From here on the receiver counts every eight bits and accepts them as a single character. Usually, the receiver checks two consecutive SYN characters to remove any doubt that the first did not occur as a result of a noise signal on the line. Moreover, when the transmitter is idle and does not have any message characters to send, it sends a continuous string of SYN characters. The receiver recognizes these characters as a condition for synchronizing the line and goes into a synchronous idle state. In this state, the two units maintain bit and character synchronism even though no meaningful information is communicated.

5-3-3 Data Transparency

The character-oriented protocol was originally developed to communicate with keyboard, printer, and display devices that use alphanumeric characters exclusively. As the data communication field expanded, it became necessary to transmit binary information which is not ASCII text. This happens, for example, when two remote computers send programs and data to each other over a communication channel. An arbitrary bit pattern in the text message becomes a problem in the character-oriented protocol. This is because any 8-bit pattern belonging to a communication control character will be interpreted erroneously by the receiver. For example, if the binary data in the text portion of the message has the 8-bit pattern 10000011, the receiver will interpret this as an ETX character and assume that it reached the end of the text field. When the text portion of the message is variable in length and contains bits that are to be treated without reference to any particular code, it is said to contain transparent data. This feature requires that the character recognition logic of the receiver be turned off so that data patterns in the text field are not accidentally interpreted as communication control information.

Data transparency is achieved in character-oriented protocols by inserting a DLE (data link escape) character before each communication control character. Thus, the start of heading is detected from the double character DLE SOH, and the text field is terminated with the double character DLE ETX. If the DLE bit pattern 00010000 occurs in the text portion of the message, the transmitter inserts another DLE bit pattern following it. The receiver removes all DLE characters and then checks the next 8-bit pattern. If it is another DLE bit pattern, the receiver considers it as part of the text and continues to receive text. Otherwise, the receiver takes the following 8-bit pattern to be a communication control character.

The achievement of data transparency by means of the DLE character is inefficient and somewhat complicated to implement. Therefore, other protocols have been developed to make the transmission of transparent data more efficient. One protocol used by Digital Equipment Corporation employs a byte count field that gives the number of bytes in the message that follows. The receiver must then count the number of bytes received to reach the end of the text field. The protocol that has been mostly used to solve the transparency problem (and other problems associated with the character-oriented protocol) is the bit-oriented protocol.

5-3-4 Bit-Oriented Protocol

The bit-oriented protocol does not use characters in its control field and is independent of any particular code. It allows the transmission of serial bit stream of any length without the implication of character boundaries. Messages are organized in a specific format called a frame. In addition to the information field, a frame contains address, control, and error-checking fields. The frame boundaries are determined from a special 8-bit number called a flag. Examples of bit-oriented protocols are SDLC (synchronous data link control) used by IBM, HDLC (high-level data link control) adopted by the International Standards Organization, and ADCCP (advanced data communication control procedure) adopted by the American National Standards Institute.

Any data communication link involves at least two participating stations. The station that has responsibility for the data link and issues the commands to control the link is called the primary station. The other station is a secondary station. Bit-oriented protocols assume the presence of one primary station and one or more secondary stations. All communication on the data link is from the primary station to one or more secondary stations or from a secondary station to the primary station.

5-4 Summary

In this lesson, we have discussed about I/O processor and communication in between of the devices. The I/O Processor with direct memory access capability that communicates with I/O devices. Channel accesses memory by cycle stealing. Channel can execute a Channel Program Stored in the main memory consists of Channel Command Word (CCW). Each CCW specifies the parameters needed by the channel to control the I/O devices and perform data transfer operations. CPU initiates the channel by executing an channel I/O class instruction and once initiated, channel operates independently of the CPU.

5-5 Keywords

I/O processing: Input and out put devices help us to interact with the computer in order to give/take the program, data, and results to/from the computer.

Commands: The main function of the devices controller is to execute a command transmitted by the CPU/software.

Modem: MOdular-DEModular is communication device, is used for long distance communication through telephone lin.

Protocol: A protocol is a set of rules that govern all aspects of information process.

5-6 Exercise

1. The address of a terminal connected to a data communication processor consists of two letters of the alphabet or a letter followed by one of the 10 numerals. How many different addresses can be formulated?

2. List a possible line procedure and the character sequence for the communication between a data communication processor and a remote terminal. The processor inquires if the terminal is operative. The terminal responds with yes or no. If the response is yes, the processor sends a block of text.

3. Draw a flowchart similar to the one in Fig. 5-2 that describes the in the IBM CPU-I/O channel communication.

5-7 References:

1. Computer System Architecture By: M.Morris Mano

2. Computer Fundamentals By: Pradeep .K.Sinha and Priti Sinha —BPB Publications

3. Computer Organisation and Architecture By: William Stallings — Prentice Publications

4. Computer Architecture and Organisation By: J.P.Hayes.