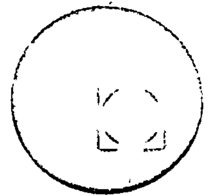
centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



A LOS ASISTENTES A LOS CURSOS DEL CENTRO DE EDUCACION CONTINUA

La Facultad de Ingeniería, por conducto del Centro de Educación Continua, otorga constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso. Las personas que deseen que aparezca su título profesional precediendo a su nombre en el diploma, deberán entregar copia del mismo o de su cédula profesional a más tardar el Segundo Día de Clases, en las oficinas del Centro, con la Señorita Barraza, de lo contrario no será posible. El control de asistencia se efectuará a través de la persona encargada de entregar notas, en la mesa de entrega de material, mediante listas especiales. Las ausencias serán computadas por las autoridades del Centro.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece el Centro están planeados para que los profesores expongan una tésis, pero sobre todo para que coordinen las opiniones de todos los interesados constituyendo verdaderos seminarios.

Al finalizar el curso se hará una evaluación del mismo a través de un cuestionario diseñado para emitir juicios anónimos por parte de los asistentes. Las personas comisionadas por alguna institución deberán pasar a inscribirse en las oficinas del Centro en la misma forma que los demás asistentes.

Con objeto de mejorar los servicios que el Centro de Educación Continua ofrece, es importante que todos los asistentes llenen y entreguen su hoja de inscripción con los datos que se les solicitan al iniciarse el curso.

ATENTAMENTE

ING. SALVADOR MEDINA RIVERO
COORDINADOR DE CURSOS.

Tacuba 5, primer piso. México 1, D. F.
Teléfonos: 521-30-95 y 513-27-95

'eds.

APLICACION DE MINICOMPUTADORAS

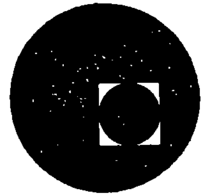
Fecha	Duración	Tema	Profesor
Nov. 28	17 a 19 h	INTRODUCCION 1.1 Sistemas numéricos 1.2 Electrónica digital	Ing. Marcial Portilla Roberton
Nov. 28 Nov. 29	19 a 21 h 9 a 10 am	ARQUITECTURA DE MINICOMPUTADORAS 2.1 C P U 2.2 Memoria 2.2.1 Dirección 2.2.2 Dirección relativa 2.2.3 Dirección directa 2.2.4 Indice 2.3 Fallas eléctricas 2.4 Autocargador 2.5 Entrada salida (E/S)	Ing. Marcial Portilla Robertson
Nov. 29	10 a 13 h	E / S DE MINICOMPUTADORAS 3.1 Teletipos 3.2 Lectoras Opticas 3.3 Cintas magnéticas 3.4 Discos 3.5 tarjetas perforadas 3.6 cassetts 3.7 C R T 3.8 Convertidores A / D 3.9 Graficadores X - y 3.10 Instrumentos científicos	Ing. Marcial Portilla Roberton
Nov. 29	14 a 16 h	INTERRUPCIONES 4.1 Interrupciones simples 4.2 Interrupciones múltiples 4.3 Vector de prioridades	Ing. José Ruiz Ascencio

APLICACION DE MINICOMPUTADORAS

FECHA	DURACION	TEMA	PROFESOR
Nov. 29	16 a 18 h	ACCESO DIRECTO A MEMORIA DMA 5.1 Robo de ciclos 5.2 interfases D M A 5.3 El Uni bus 5.4 Bus de datos	ING. JOSE RUIZ ASCENCIO
Dic. 5	17 a 20 h	PAGINACION 6.1 Memoria virtual 6.2 Paginación 6.3 Organización y Administración de la memoria	DR. ADOLFO GUZMAN ARENAS
Dic. 5 Dic. 6	20 a 21 h 9 a 11 am	SOFTWARE (PROGRAMACION DE SISTEMAS) 7.1 Ensamblador 7.2 cargador 7.3 interpretador 7.4 compilador 7.5 macros 7.6 ensamblador condicional	DR. ADOLFO GUZMAN ARENAS
Dic. 6 y de	11 a 13 h 14 a 15 h	SISTEMAS OPERATIVOS EN TIEMPO REAL 8. 1 monitor 8. 2 prioridades 8. 3 cambio de prioridades	SR. RAYMUNDO SEGOVIA
Dic. 6	15 a 18 h	INTERFACES 9.1 señales analógicas 9.2 transductores 9.3 ruido	ING. FEDERICO KUHLMANN

Dic. 12	17 a 20 h	9.3.1 Precisión 9.3.2 Ganancia 9.4 Amplificadores 9.5 Convertidores A / D 9.6 Convertidores D / A 9.7 C A M A C	ING. JOSE LUIS VAZQUEZ
Dic. 12	20 a 21 h	APLICACIONES	DR. VICTOR GEREZ GREISER
Dic. 13	9 a 13 h		[illegible]. JORGE GIL [illegible].GERTRUDIS KURZ
Dic. 13	14 a 18 h		OTROS

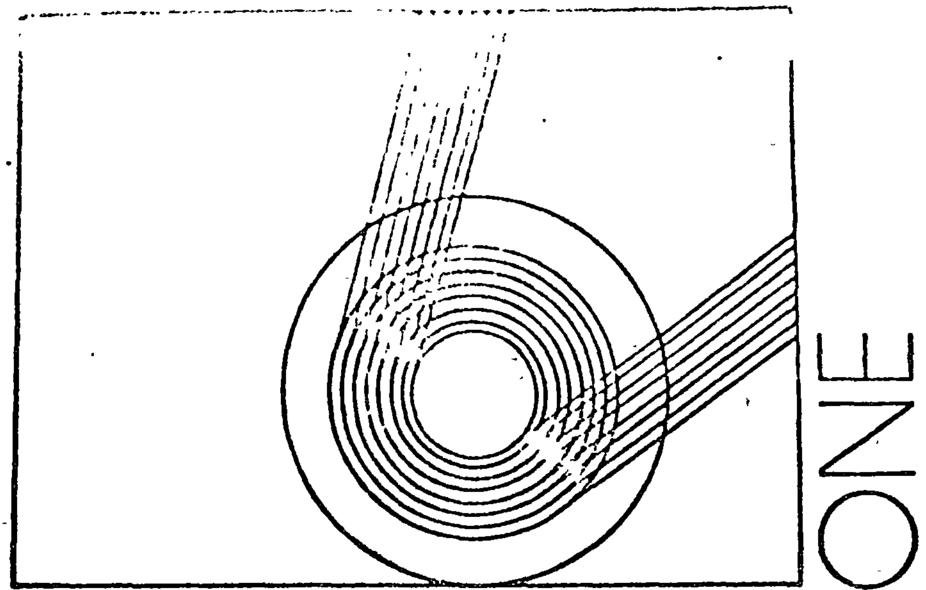
centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



APLICACION DE MINICOMPUTADORAS

M. EN I. MARCIAL PORTILLA.

Tacuba 5, primer piso. México 1, D. F.
Teléfonos: 521-30-95 521-73-35



INTRODUCTION

The number of areas in our modern society where digital computers are now employed is truly remarkable. Computers are used in our banks; airlines and train companies use them to make reservations, in schools the bookkeeping and sometimes even class scheduling are done by computer. They are used in the control systems for large factories and to navigate and guide everything from space vehicles to submarines. Computers help design and then control the operation of nuclear reactors, help manage hospitals and monitor patients' recovery, record our births and deaths on government records.

The widespread use of computers appears even more remarkable when the life-span of the computer industry is considered. In less than 30 years the digital computer industry has grown from a few experimental machines to a 50-to-100-billion-dollar-a-year international industry, with a growth rate estimated at over 15 percent a year. Further, as the price of individual computers comes down, the number of application areas where the use of a computer is economically attractive seems to rise correspondingly, so that this phenomenal growth rate appears due to continue

1.1 THE STORED PROGRAM

The history of mechanical and electronic aids to computation is long, dating back to the abacus and progressing through the early mechanical calculators and early punched-card data processing equipment The modern digital computer has one characteristic, however, which really distinguishes it from earlier devices. This characteristic, from which the digital computer's advantages are largely derived, lies in the computer's ability to perform long sequences of calculations without human intervention Whereas, for instance, a conventional desk calculator requires that numbers be individually inserted and that operations on the numbers be sequenced by an operator, the digital computer has a memory in which resides a sequence of instructions detailing exactly what operations are to be performed, including alternate paths through the instructions if decisions are to be made

The computer executes this sequence of instructions, called a *program*, at its own speed, performing instructions at a rate of a few hundred thousand to millions per second, depending on the computer.

The existence of a stored program consisting of instructions to the computer implies, of course, that humans have prepared these instructions, and it is the way such programs are prepared that occupies most of this book.

Before a computer can be used to "solve a problem" in science or to "process data" in business, someone must prepare a program detailing exactly how the computer is to proceed. The preparation of this program is called *programming*,[†] and the amount of effort and expense currently going into programming exceeds the amount currently expended on hardware. (The computer industry calls the actual physical electronic machine and its associated peripheral devices, such as printers or card readers, *hardware* Computer programs are then called *software*. In computer jargon, software costs now exceed hardware costs)

Other digital computer attributes are less obvious, and perhaps more matters of degree, but contribute considerably to the computer's worth. These lie in the computer's ability to store large volumes of data, including names, addresses, sentences, and other nonnumeric data, as well as numeric data and programs The computer is able to find specific data in short periods of time, ranging from less than a millionth of a second to perhaps seconds, depending on the amount of data and the types of computer memory used

Still other important computer attributes include the abilities to read data and write results quickly and to communicate complicated results to users in various forms, including in some cases graphical devices which display curves or "pictures" of various kinds.

[†]To use a computer, it is necessary to learn how to prepare computer programs, to use a computer *well*, it is necessary to learn something more This ill-defined "something more" lies in the area called "computer science"

1.2 THE STRUCTURE OF GENERAL-PURPOSE COMPUTERS

Before studying actual programming, it is useful to examine the overall structure of computers. Particular attention will be paid to the input and output devices, for it is by means of these that we communicate with the computer. Later chapters will examine computer memories and architecture.[†]

A general-purpose digital computer can be divided into five major parts: *input devices, output devices, memory, arithmetic element*, and *control section*. Figure 1.1 shows a block diagram of a general-purpose computer with system block diagram symbols for some of the typical devices used.

Input Devices

The most used input device is doubtless the punched-card reader, an electromechanical device that accepts a deck of punched cards, separates single cards from the deck in order, and senses the presence or absence of a hole in each of the positions on the card where a hole can be punched

A similar device is the perforated-tape reader, which senses the presence or absence of holes in a strip of tape (commonly made of a reasonably sturdy paper or plastic) Reading devices use (1) metal "pins" which either enter or do not enter punching positions, thereby indicating the

[†]Computer architecture concerns overall organization with emphasis on the arithmetic element, control of and interaction between sections, and the instruction repertoire

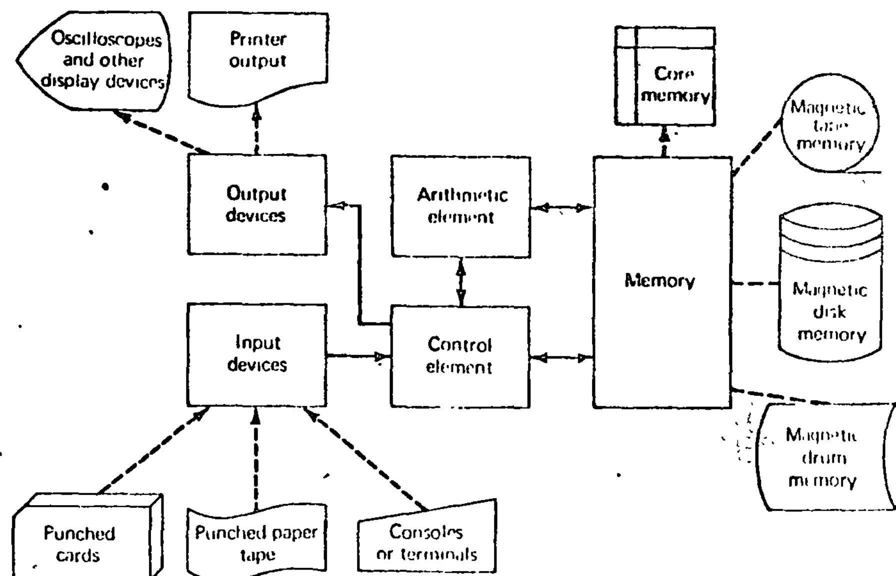


Figure 1.1 Block diagram of a digital computer

presence or absence of a hole in a given position or (2) photoelectric cells and a light source arranged so that the light passes or does not pass through the holes activating or not activating the cells Readers using photoelectric cells tend to be much faster

Other inputs to digital computers include magnetic tape, photographic film, microphones and keyboards such as the Teletype keyboard in Fig 1 2 which is often used to enter typed characters directly into a computer by means of electrical connections Most Teletypes can also be used to prepare punched paper tapes and to read tapes into a computer Input devices also include the push buttons on the computer console, used to start and stop the machine and sometimes to enter data.

Special types of input devices are used in *real-time control* systems which process data in a dynamic system and use information supplied by sensors to make decisions and to control the systems' operations The inputs to real-time systems include accelerometers or gyroscopes in missiles, radar sets in air-traffic control or air-defense systems, and thermometers or blood flow measuring devices in hospital equipment These devices are likely to be analog in nature; and in order for information to be entered into a digital computer system, the analog data must be converted into digital form. Devices that convert analog signals to digital form are called *analog-to-digital converters*.

Output Devices

For human interpretation, there is no substitute for the printed word. Digital computers therefore come equipped with printing devices ranging from typewriters to large high-speed (line-at-a-time and 1,000 lines per minute) printers; these make up the majority of output devices. The Teletype in Fig. 1.2 can, for instance, be used to type computer-generated data via the typing mechanism Printing devices also include special printers for payroll checks and labeling machines. The emphasis is almost always on speed (with cost versus speed always a factor) because the computer can usually supply output faster than printers can reproduce it This has led to widespread use of "offline" printing systems, where the computer writes its results on magnetic tape and the tape is later read and reproduced by one or more relatively lethargic printers The speed attainable by the computer in writing on magnetic tape is much greater than that of the printer, and the tape mechanism is more reliable than are the printing devices Other output devices include oscilloscope displays, lights, and loudspeakers

Memory

A computer has the ability to store its program, the input data, and the intermediate and final results of its calculations. The computer's ability to operate in an autonomous manner, after the program and input data have

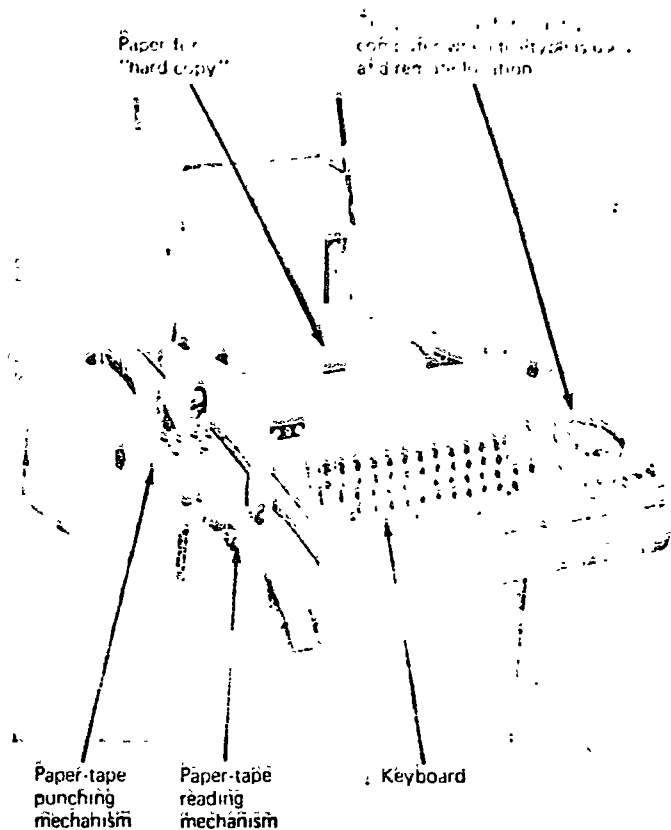


Figure 1 2 Teletype

been read, is made possible by a memory that contains these data during the processing

Large general-purpose computers come equipped with several kinds of memory. The most important is the high-speed memory, which is integral to the control and arithmetic sections and with which these sections can communicate directly High-speed memories are generally magnetic-core or integrated-circuit devices† They can be read from or written into at relatively high speeds, but are fairly expensive in terms of initial cost and also because of space requirements, complexity, and power consumption

Slower backup memories are usually magnetic tapes or disk storage devices, which use magnetic disks resembling phonograph records (The disk reading and writing device looks somewhat like a jukebox) Photographic film and magnetic cards are also used The characteristics of these

†Details are given in later chapters

and other mass or backup memory devices are low cost per digit stored, simplicity of the mechanisms required to read from and write on the storage media, and, generally, the ability to store information for long periods of time without regeneration. There are no hard-and-fast rules that determine what is a backup memory device and what is to be used as part of the computer's high-speed memory. Speed and cost are relative: magnetic drums once served as high-speed memory devices for low-priced computers and are now used primarily for buffering or bulk memory in the most expensive, larger computers.

Arithmetic Element

This section of the digital computer performs the arithmetic and other operations on the operands (data) stored in the memory. These operands are delivered to the arithmetic element from the memory at the direction of the control element, which sequences the operations performed.

Most arithmetic units can perform the operations of addition, subtraction, multiplication, and division, and some can do such operations as finding square roots. It is important to note that the arithmetic unit can perform not only arithmetic but also logical operations and can help the control unit make decisions.

Control Element

The control element obtains the sequence of instructions which direct the computer in order from the memory and then controls the operations of the other parts of the computer. The control section interprets the instructions and converts them into actions, directing the operation of the arithmetic unit, input-output devices, and memory. An important attribute of the computer is its ability to change the sequence of calculations it performs as a result of its calculations. The control unit provides this facility.

1.3 INPUT DEVICES

This section describes several of the more popular input devices in more detail. Some consideration is also given to the preparation of data for entry into the computer.

Perforated Tape

When the first computers were designed, telegraph systems had been using perforated paper tapes for some time, and as a result, devices for punching and reading paper tapes had already been fairly well developed, leading to wide use of tape. Today, minicomputers make considerable use of punched tape. It is an economical input medium.

The tape used is of many types and sizes. A medium-thickness paper tape has been used a great deal, and oiled tapes and plastic tapes are also used. The widths of the tapes used have varied from $\frac{1}{2}$ to 3 inches

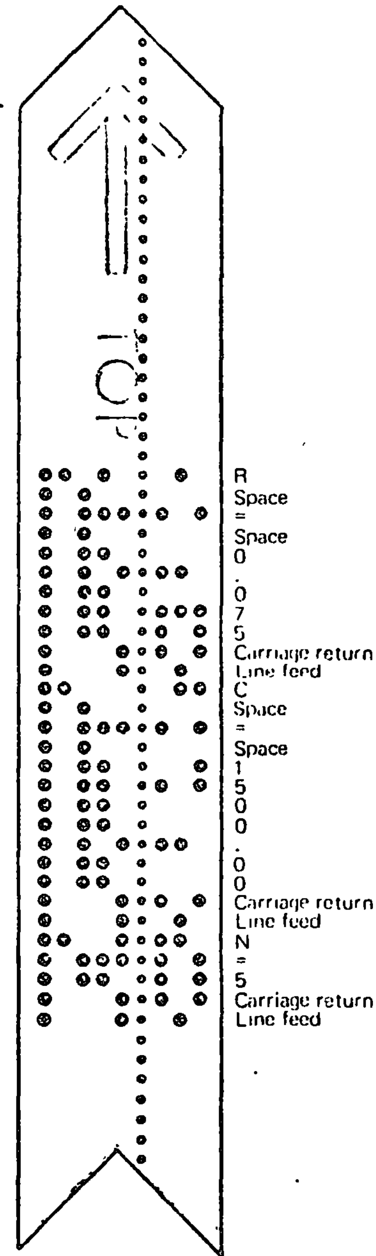


Figure 1.3 A section of punched-paper tape

Information is punched into the tape a line at a time. Figure 1.3 illustrates a section of a perforated tape. Multiple channels are used (a channel runs lengthwise along the tape), and the code for a single character is punched as a pattern of bits in each lateral line.

The preparation of input tapes is generally referred to as *keyboarding*. A typical paper-tape punching device was shown in Fig. 1.2. The tape-punching device used may be one of a number of types, however, the more popular devices resemble a typewriter, and the keyboards of these tape punches contain conventional symbols, similar to those on an ordinary typewriter. The keyboards of many tape-punch machines are identical with the keyboards of manual typewriters used in businesses, and sometimes electric typewriters are converted to tape-punching machines by attaching a punching device which is actuated by the typewriter mechanism.

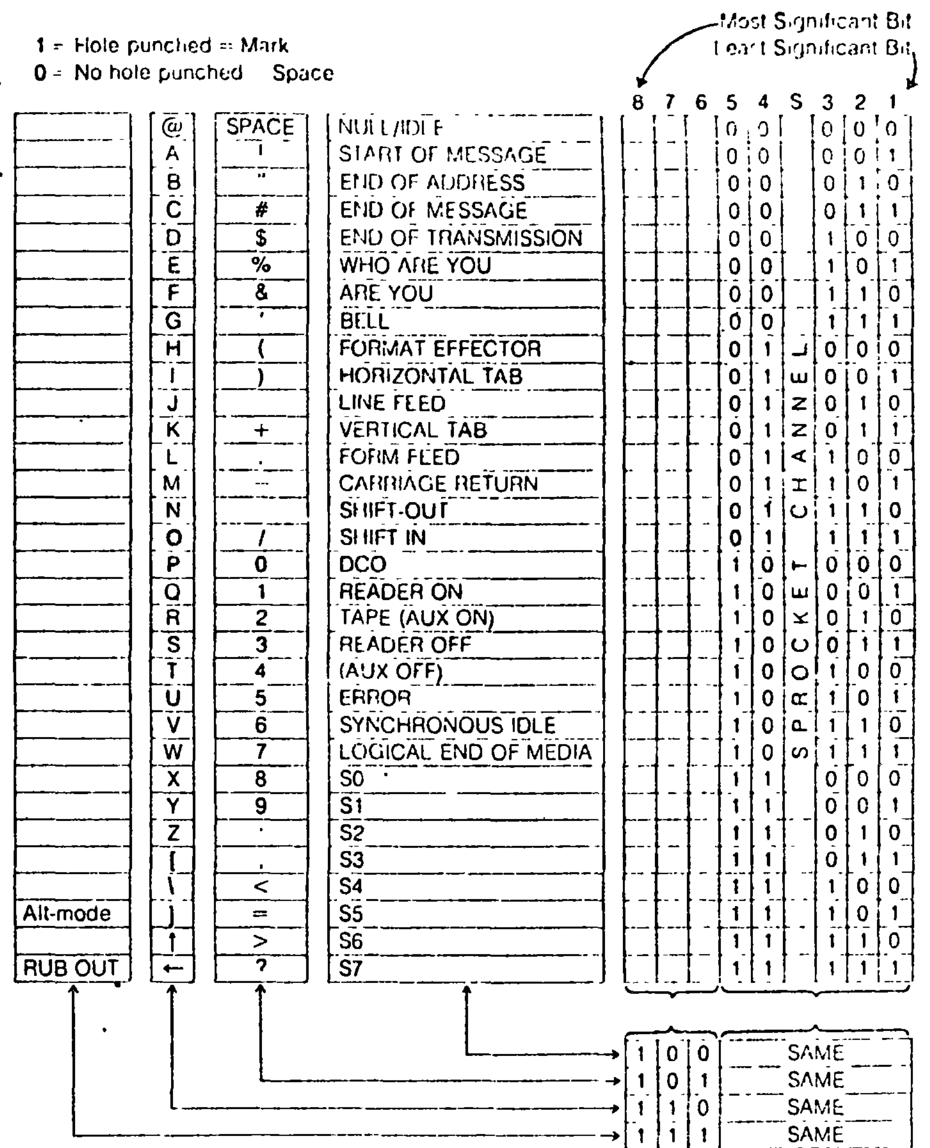
The coded symbol for a character is punched into the tape each time a key is depressed, and the tape then advances to the next line. In most cases, the tape-punching device also prints on a separate piece of paper, in the same manner as a typewriter, the character which was punched, thus making a typewritten copy of the program, which may be checked for errors, in addition to the paper tape punched with the coded symbols. This printed copy of the program is referred to as the "hard copy." Many of the tape-punch machines are able to read a perforated tape and to type printed copy from it. A punched section of tape may be placed in the tape reader attached to the tape-punch machine, and a typed copy of the information which was punched in the tape may be made.

Several codes are used in punched tape systems. One of the most popular is shown in Fig. 1.4. This figure shows an 8-channel code where there are eight channels running lengthwise along the tape.

Most of the tape readers used in Teletype machines and office equipment are electromechanical devices. Often mechanical "sensing pins" are used to determine the symbol punched into each line of the tape. In a system of this type, there will be a sensing pin for each information channel, plus a means of moving the tape and positioning it for reading. The tape is not moved continuously but only a single line at a time, stopped while the coding is sensed, and then moved to the next line. The motion of the sensing pin operates a switch the contacts of which are opened or closed, depending on whether there is a hole in the tape.

The motion of the tape through the reader is generally controlled by the computer. Each time the tape is to be advanced and a new character read, the computer supplies the reader with a signal which causes it to advance the tape to the next character. To read characters as fast as possible, a line is generally read at the same time as the advancing pulse is transmitted. Since there is a delay due to inertia before the tape is actually moved, the reading of the state of the sensing relays occurs during this delay period. In this case, when a STOP character is sensed, the reader proceeds to the next character before actually stopping.

High-speed tape readers use photoelectric cells or photodiodes to read the characters punched into the tape. In these readers, a light-sensitive cell is placed under each channel of the tape, including the "tape feed hole," or "sprocket channel." A light source is placed above the tape



				8	7	6	5	4	S	3	2	1
	@	SPACE	NULL/IDLE				0	0		0	0	0
	A	!	START OF MESSAGE				0	0		0	0	1
	B	"	END OF ADDRESS				0	0		0	1	0
	C	#	END OF MESSAGE				0	0		0	1	1
	D	$	END OF TRANSMISSION				0	0		1	0	0
	E	%	WHO ARE YOU				0	0		1	0	1
	F	&	ARE YOU				0	0		1	1	0
	G	'	BELL				0	0		1	1	1
	H	(	FORMAT EFFECTOR				0	1		0	0	0
	I	)	HORIZONTAL TAB				0	1		0	0	1
	J		LINE FEED				0	1		0	1	0
	K	+	VERTICAL TAB				0	1		0	1	1
	L	,	FORM FEED				0	1		1	0	0
	M	-	CARRIAGE RETURN				0	1		1	0	1
	N		SHIFT-OUT				0	1		1	1	0
	O	/	SHIFT IN				0	1		1	1	1
	P	0	DC0				1	0		0	0	0
	Q	1	READER ON				1	0		0	0	1
	R	2	TAPE (AUX ON)				1	0		0	1	0
	S	3	READER OFF				1	0		0	1	1
	T	4	(AUX OFF)				1	0		1	0	0
	U	5	ERROR				1	0		1	0	1
	V	6	SYNCHRONOUS IDLE				1	0		1	1	0
	W	7	LOGICAL END OF MEDIA				1	0		1	1	1
	X	8	S0				1	1		0	0	0
	Y	9	S1				1	1		0	0	1
	Z	:	S2				1	1		0	1	0
	[	;	S3				1	1		0	1	1
	\	<	S4				1	1		1	0	0
Alt-mode	]	=	S5				1	1		1	0	1
	↑	>	S6				1	1		1	1	0
RUB OUT	←	?	S7				1	1		1	1	1

8	7	6	5–1
1	0	0	SAME
1	0	1	SAME
1	1	0	SAME
1	1	1	SAME

Figure 1.4 American Standard Code for Information Interchange (ASCII) 8-channel paper tape code

so that the light-sensitive element beneath a hole in the tape will be energized and will produce a signal indicating the presence of a hole. The signals from the light-sensitive elements are then amplified and supplied to the computer as input information.

The tape feed hole is used to determine when the outputs of the light-sensitive elements are to be sensed. The tape in a reader of this type is

generally friction-driven and moved continuously until a STOP character is sensed Extremely fast starting and braking of the tape are very desirable features, and most readers are capable of stopping the tape on any given character

The speeds attainable with various tape readers are generally expressed as the number of characters per second which can be read Mechanical sensing readers have been designed to operate at speeds as high as 200 characters per second, although speeds of from 10 to 60 characters per second are more common Present-day photoelectric readers operate at speeds of up to 1,000 characters per second.

Punched Cards

The most popular punched card at this time is an 80-column card $3\frac{1}{4}$ inches wide and $7\frac{3}{4}$ inches long

Just as with tape, there are numerous ways in which punched cards may be coded Frequently used is the Hollerith code, an *alphanumeric code*† in which a single character is punched in each column of the card The basic code is illustrated in Fig 1 5 As an example, the symbol *A* is coded by means of a punch in the top and 1 row of the card, and the symbol 8 by a punch in the 8 row of the card.

There are other types of cards with different hole positions on the cards just as there are many ways of preparing the cards to be read into the computer. The most common technique is very similar to that for preparing punched tape, in that a *card punch* with a keyboard like that of a typewriter is used. Generally, the card punch prints the characters punched into a

† An alphanumeric code is one which includes alphabetic and numeric characters Generally, special characters such as periods, commas etc, are included

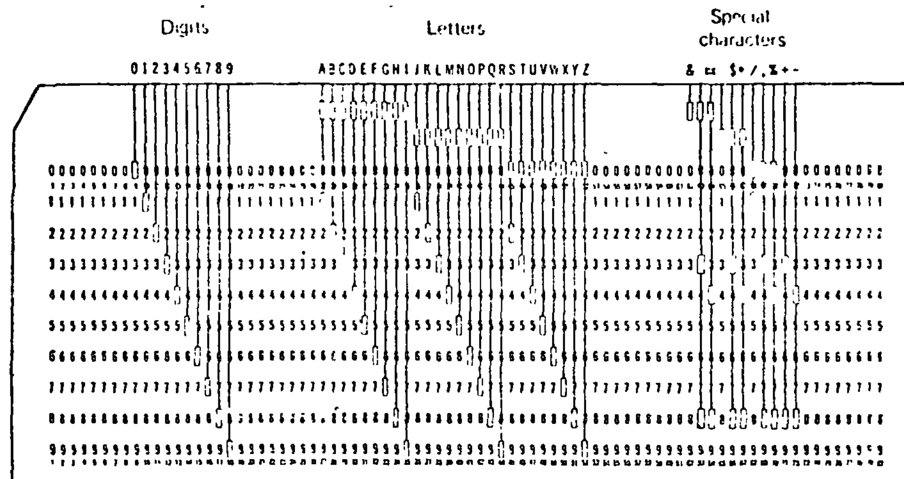


Figure 1 5 Punched card with Hollerith code

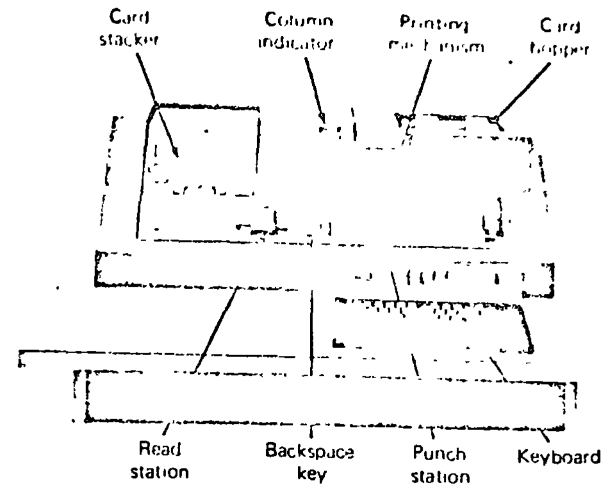


Figure 1 6 Card punch

card at the top of the card itself In this way, a card may be identified without examination of the punches Each character is usually printed at the top of the card directly above the column in which the character is punched.

The card punch contains a hopper in which the blank cards are stacked (refer to Fig 1 6) The operator of the card punch causes a card to enter the punching area, and the section of the program or data to be processed are punched into the card laterally, a column at a time, starting at the left. If a key of the card punch is depressed, the code for the character is punched into a column of the card, and the card is then moved so that the next column on the right is under the punch.

When a program is punched, often only one instruction or statement to the computer is punched into each card. Then if an error is made in programming, the erroneous instruction may be changed by throwing the incorrect card away and replacing it with a correct card.

The preparation of cards containing data for business systems is highly developed Large businesses such as insurance companies, credit agencies, and banks must gather and process almost incredible amounts of data Most of the processing of these data is now done by electronic computers, but even prior to the advent of the digital computer, punched cards were extensively used, and a considerable technology for handling punched cards was developed

Whenever punched cards are used, the data—be it checking account records, income tax rates, or whatever—must first be entered into a punched card by a keypunch operator. To give some idea of the magnitude of this operation; over 500,000 keypunching consoles are in existence at this time.

Because of the enormous amount of keypunching which is done in large

Figure 1 7 IBM 129 card punches in operation

businesses, many special features have been introduced into card punches. Figure 1 7 shows a card punch with a small memory capable of holding all the data which can be punched into two 80-column records and six program cards The keypunch operator keyboards the data into the small memory and can backspace and change characters until the data keyboarded are correct After all the data are in the small memory, an ENTER DATA key is depressed, and the card is then punched from the characters in the memory. Facilities for controlling the format of the card are included by means of integrated-circuit logic

Most *punched-card readers* are electromechanical devices which read the information punched into a card, converting the presence or absence of a hole at each punching position into an electric signal which is read by the computer. The punched cards are placed into a hopper, and when the

command to read is given, a lever pushes a card from the bottom of the stack Generally, the card is then moved lengthwise over a row of 80 "read brushes" These brushes read the information punched along the bottom row of the card If a hole is punched in a particular row, a brush makes electrical contact through the hole in the card, providing a signal which may be used by the computer. The next row up is then read, and this process continues until all rows have been read, after which the next card is moved into position on the brushes.

Faster card readers have been constructed that use photoelectric cells under the 12 punch positions along a column and an illuminating source above the card. As each column on the card is passed over the 12 photoelectric cells, whether a given position is punched is determined by the presence or absence of light on the corresponding cell. Card readers operate at speeds of from 12 to 1,000 cards per minute.

1.4 PREPARING DATA FOR COMPUTER USE

As has been mentioned, the preparation of data for use in businesses can be expensive and complicated Insurance companies, for example, employ hundreds of operators to keyboard data concerning policies and claims for computer use Banks must process hundreds of thousands of transactions each day, and each must be prepared for entry to the computer.

In many cases it is desirable to buffer large amounts of data by first recording the data on some medium that can be quickly read and which is more easily handled than large stacks of punched cards One form of buffering consists in first recording the data to be read in on magnetic tape and then reading from the tape into the computer. A number of devices are available which will read punched tape or cards and transfer the information punched into them onto magnetic tape. This process takes place outside the central computer. Since the magnetic tape may be read much faster than punched cards or paper tape, the time required to read in information is reduced

Converting from punched cards to magnetic tapes has been bypassed by a number of input devices Several companies now offer a line of keyboard-to-magnetic-tape devices Figure 1 8 shows a console of this type In its simplest form, when the operator depresses a key, the character selected appears on a display and is also entered on the tape When the magnetic tape has been filled or all records have been transcribed, the tape can be read by the computer

More advanced systems permit the keyboard operator to type a number of characters, these characters are displayed on a console where they may be read and checked and also are stored in a small memory The characters can be edited (changed), and when the operator is satisfied that the data are correct, a RECORD button is depressed, the data are entered on magnetic tape, and new data may be keyboarded

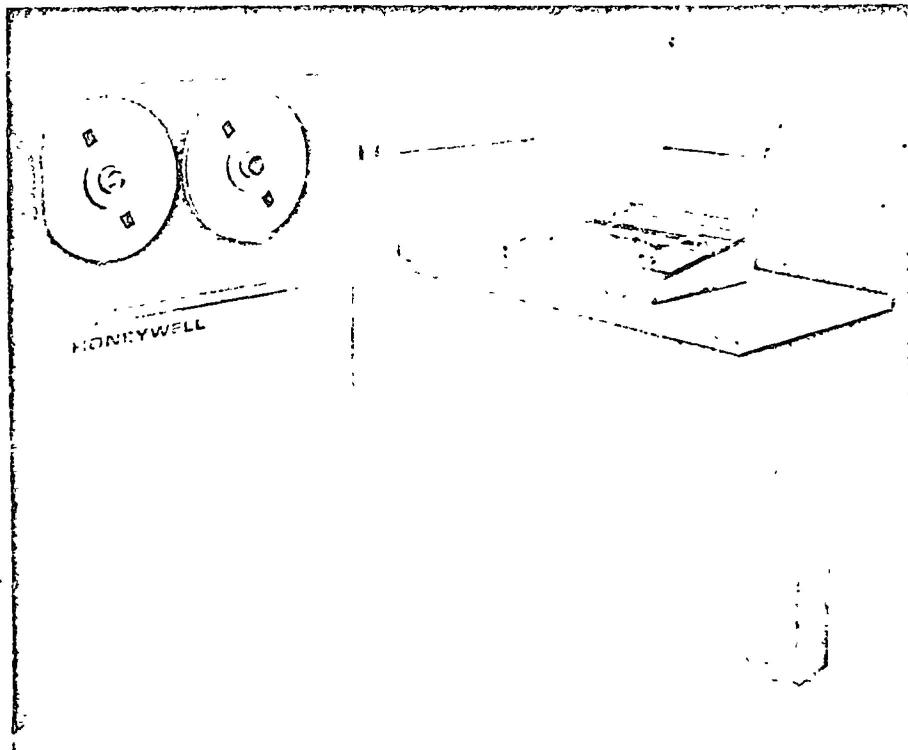


Figure 1.8 Keyboard-to-magnetic-tape device

Larger systems use a number of keyboards, all connected to a small memory and processor equipped with a magnetic-tape recorder. This permits extensive editing; checking by the small processor-computer to compare inputs, error detection, particularly format checking of many types, and many sorting and collating operations. The magnetic tape prepared by a system of this sort has had many "preprocessing" operations already performed, and the data have been subjected to considerable checking and preprocessing. For companies with extensive data processing requirements, such systems are gaining popularity.

The use of magnetic tapes, including cassettes and cartridges, has gained in popularity because tape is cheaper than cards for bulk storage and more convenient for system operation. However, the absence of cards containing printing which can be stored, examined, and manually handled still meets with some resistance.

Some idea of the importance of input data preparation can be seen from the fact that 30 to 50 percent of installation cost is often for data preparation. This is especially true, of course, for the larger business systems, but in even stric[illegible] scientific installations of modest size as much as 20 percent of the system [illegible]osts can go into preparation of data for computer input. As a

further example, in 1968 the Internal Revenue Service alone used about 400 million punched cards.

Numerous special devices and media are used to reduce the cost to business for data entry. For example, firms such as American Express use a combination of punched cards with printed characters for their billing. The customer returns the punched card with his check, and the card is then given directly to the computer.

Many companies use identification cards with special coding for their employees. In these systems, the employee inserts his or her card in a special reader which reads the identification and also enters the time from a clock. These data are recorded on tape or a disk, and later the computer reads all the check-in and check-out data recorded during the day. Still other systems use a time clock to simply punch check-in and check-out time in a card which has the employee's identification already punched into it. These cards are then collected at the end of the week. Each of these schemes reduces the expense of employing operators to punch the data. Gas pumps which read customers' credit cards and record the amounts of sales, cash registers which carry sales data to a central tape file in a large department store, and many similar devices are being used to alleviate the problems and expense of preparing data for computer entry.

1.5 CHARACTER RECOGNITION

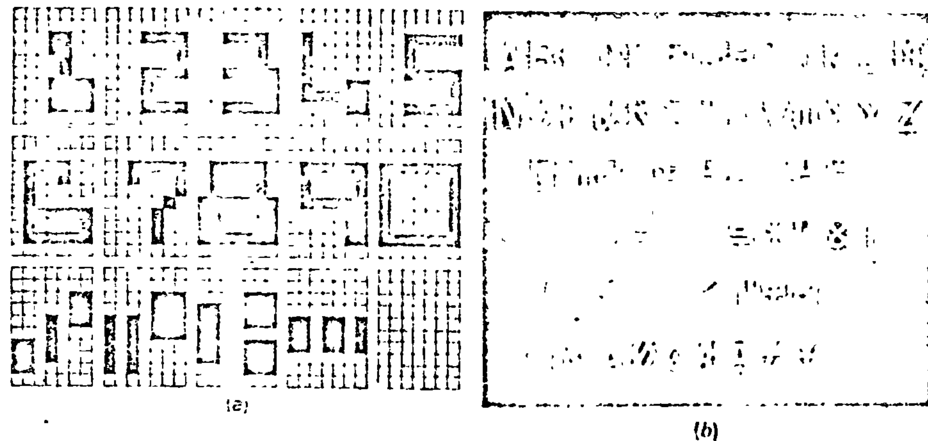
Techniques for data entry extend in many directions. The reading of handwritten or typewritten characters from conventional paper appears to offer an ideal input system for many applications. The systems currently in use work primarily as follows.

Magnetic-Ink Character Reading (MICR)

The recording of characters by means of an ink with special magnetic properties and with special forms for the characters was originally used in quantity by banks. The American Banking Association settled on a type font, and several of their characters are shown in Fig. 1.9*a*. A magnetic character reader "reads" these characters by examining their shapes using a 7 by 10 matrix and determines, from the response of the segments of the matrix to the magnetic ink, which of the characters has passed under the reader's head. This information is transmitted to the system. The determination of the character which is read is greatly facilitated by the careful design of the characters and the use of the magnetic ink.

Optical Character Reading (OCR)

The most used optical character readers require that a special type font (or fonts) be used to print on conventional paper with conventional ink. The printed characters are passed under a strong light and examined by a lens system that differentiates light (no ink) from inke[illegible] [illegible]eas and by a



(a) (b)

Figure 1.9 (a) A set of characters to be used in a magnetic-ink character reading (MICR) system. (b) Type font for optical character reading (OCR)

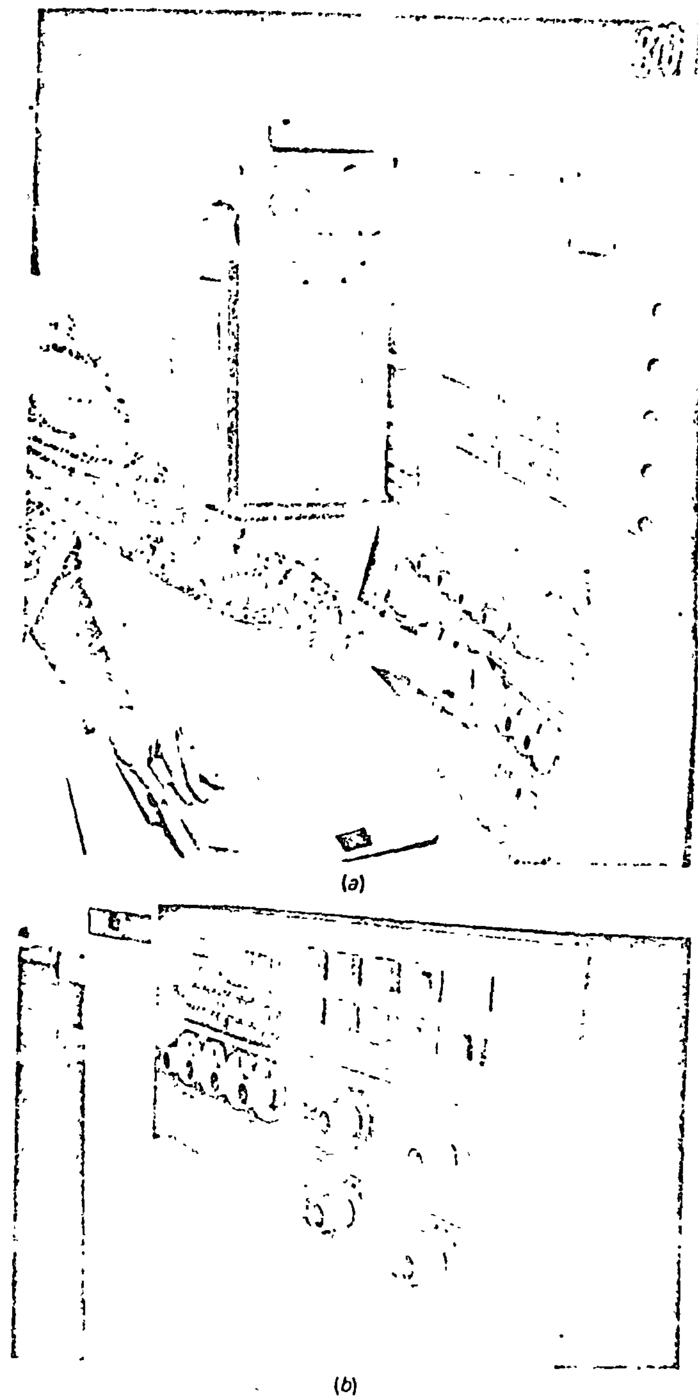
logical system that attempts to determine which of the possible characters in the system is being examined The systems in actual use depend heavily on the fact that only a limited number of characters in a particular font are used, but such systems are still quite useful The standard type font agreed on by the USASI optical character committee is shown in Fig 1.9*b*

The ideal system would, of course, be able to adapt to many different type fonts Some systems, particularly one developed by the Post Office, read handwritten characters The success of these systems has been limited, because of the many shapes that a given character can have; consider the ways you can write an *a* and the similarity between a handwritten *a* and an *o* or a *b* and an *f*. These problems are increased by the optical reader's difficulty with the porosity of paper, ink smearing at the edges of lines, etc. Much work continues in this area, and much more is needed, but the advantages of such systems continue to spur research.

1.6 OUTPUT DEVICES

Many types of output equipment are now in use, but the most popular form of output remains the printed word. Other types of display devices in regular use include neon lights and oscilloscopes, and some computers are even equipped with loudspeakers (Attempts have been made to compose music by means of a computer. Many banks have computers which "talk" with tellers)

Figure 1 10*a* shows the lights and switches on the console of the IBM System/370 Model 155, a large machine, and Fig 1 10*b* shows the console of the smaller IBM Model 7. These lights can be used as output devices for very simple programs where the answers may be read visually, however, ... lights are generally used as maintenance aids often to troubleshoot



(a)

(b)

Figure 1 10 (a) Console of a large computer, the IBM 370-155 (b) Console of a small computer, the IBM System 7

the operation of the machine Toggle switches and push buttons can be used to enter data into computer registers and to sequence operations of the computer (as well as to start and stop it) The lights are sometimes used to troubleshoot programs for small machines. If, for instance, a program works itself into an endless loop, the machine can be stopped and the location of the faulty instruction sequence in the program can be read from the lights on the console

Details of the operations of the two most used output devices; printers and oscilloscopes, follow.

Printers

For the sake of convenience, a printer should have the ability to print characters, decimal digits and common punctuation marks

A printer is said to be operated *online* when it is directly connected to a computer and prints characters directly given it by the computer. A printer is said to be operated *offline* when it prints data which have previously been recorded on punched cards, magnetic tape, or some other media by a computer The information delivered to a printer operated online will be in the form of electrical signals directly from the machine If the printer is operated offline, the reading and decoding of data stored on punched tape, punched cards, or magnetic tape may be a part of the printing operation. Since the electronic circuitry of a computer is able to operate at speeds much higher than those of mechanical printing devices, it is desirable that a printer operated online be capable of printing at a very high speed Even if the printer is operated offline, speed is highly desirable, since the volume of material to be printed may be quite large

Most of the original printers were converted electric typewriters or Teletype machines The speed of such printers is relatively low, perhaps from 10 to 30 characters per second

There are many different ways to construct printers Perhaps the two most basic types are *impact printers*, which use hammer or type bars or balls to print, and *nonimpact* printers, which squirt ink, use some photographic process, or even burn marks on the paper. Impact printers have so far been the most used

Impact printers vary greatly in construction, cost, and capability. One type has a number of "pallets," each with a raised character on the surface nearest the roller or contact drum, mounted in a movable type basket This basket is positioned electromechanically, and the back of the pallet is then struck by a hammer which forces the pallet against the page An inked ribbon is positioned between the pallet and the page (like a typewriter ribbon), the character on the pallet is thereby printed on the page, the roller is moved to the left, and another character is printed Sometimes the pallets are contained in a rectangular "type box" which is also positioned by the action of the decoding mechanism, and the correct pallet is then struck by a hammer [illegible] the typewriter mechanisms and type bar and type box mechanisms are capable of printing up to perhaps 40 characters per second

If a number of type bars are located in a row along the drum and positioning mechanisms and hammers are provided for each bar, a *line-at-a-time* printer may be constructed This type of printer can print up to four lines per second, a speed much higher than that of the character-at-a-time printer

Even faster printers are constructed in which the raised characters are distributed around a "print wheel" that revolves constantly (refer to Fig 1.11) In this case, the print wheel does not contain moving parts like the pallets just described, but consists of a motor-driven drum with a number of bands equal to the number of characters printed per line A set of all the characters which are used is distributed around each band The print wheel is revolved continuously When the selected character is in position, the print hammer strikes the ribbon against the paper and thus against the raised character on the print wheel located behind the paper Printers of this type can print up to 1,250 lines per minute with 160 characters per line

Figure 1 12 shows a printer for the ledger cards used in accounting systems This printer is typical of those used in many business-oriented systems The ledger cards are dropped in the feed slot in the center just above the keyboard This printer is often used in conjunction with a disk file, automatically posting data from the files in the ledger cards.

Oscilloscope Display Devices

Cathode-ray tubes were first used to display curves, the coordinates of which were calculated by programs More recently, the cathode-ray tube has been used a great deal to display characters as well The cathode-ray tube is a very fast output device but does not deliver permanent copy Therefore such tubes are sometimes used in conjunction with a camera, so that the display on the tube face can be photographed and recorded permanently

The cathode-ray tubes used in computer displays are the same type as those used in oscilloscopes and television sets, and entire television sets are sometimes used For these display systems, the displayed points are made by positioning and turning on an electron beam in the tube, as is customary in television sets or oscilloscopes The displays are called *oscilloscope*, or "scope," *displays* Figure 1 13 shows a cathode-ray tube display with a keyboard

An oscilloscope display or printer together with a keyboard is generally called a *terminal*. Terminals of this type are often used in systems where the terminal is directly connected to the computer and the user types directly into, and receives responses directly from, the computer Such systems are frequently called *interactive systems* because they promote "interaction" between the computer and the user

In some cases the terminal is connected to the computer using telephone lines When a key is depressed on the keyboard [illegible] sequence of

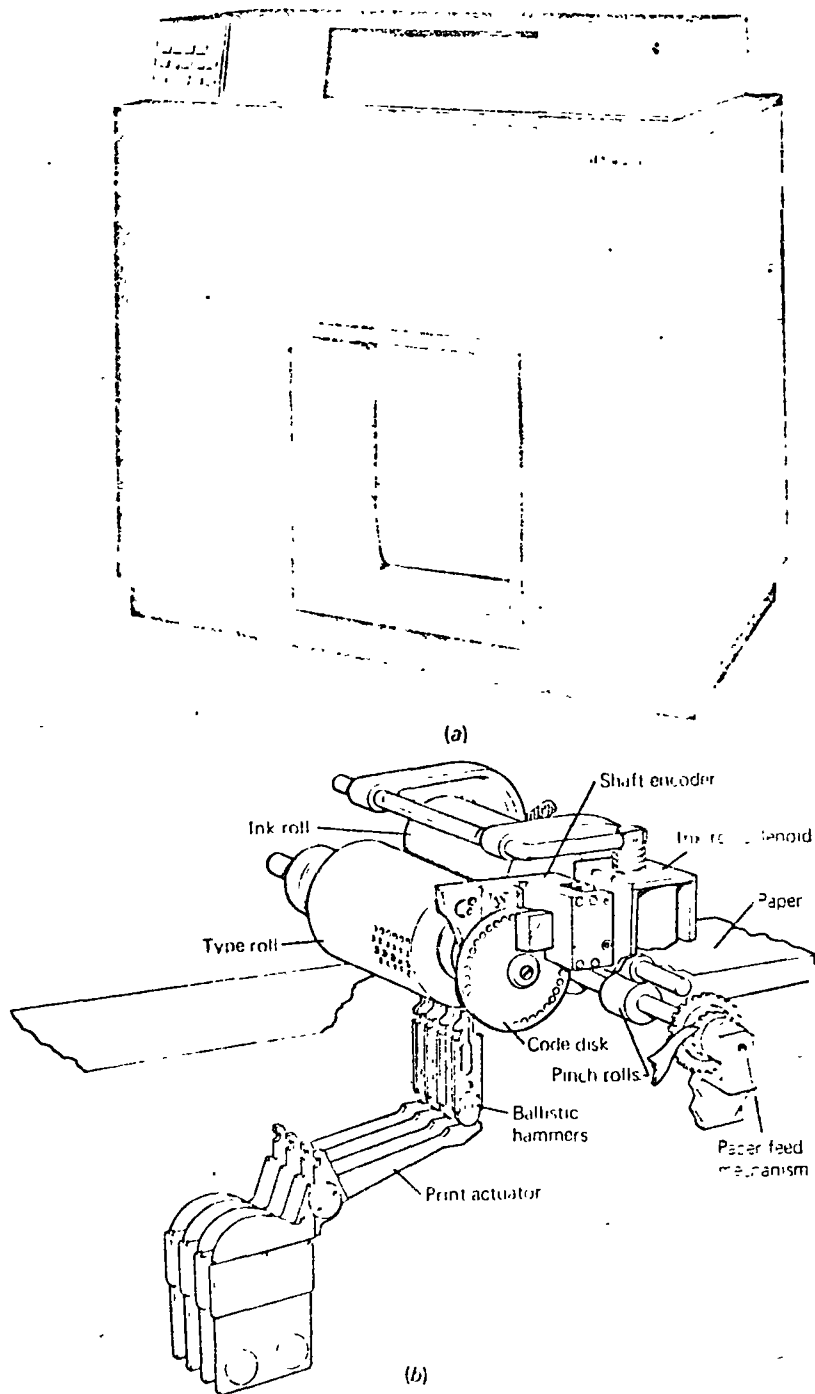


Figure 1.11 High-speed impact printers (a) High-speed line printer (b) Typical mechanism for printer with a print wheel

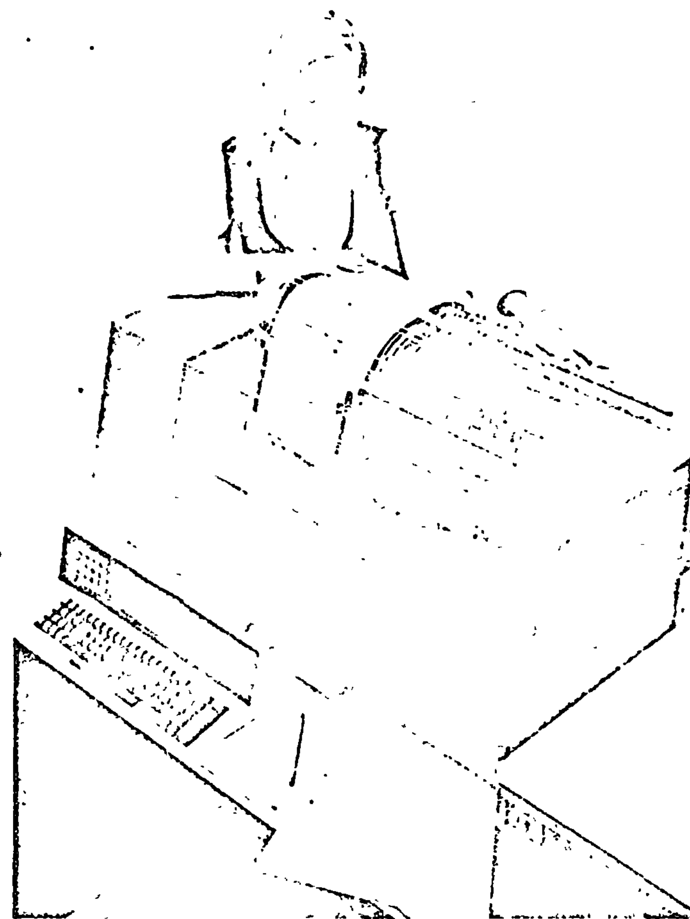
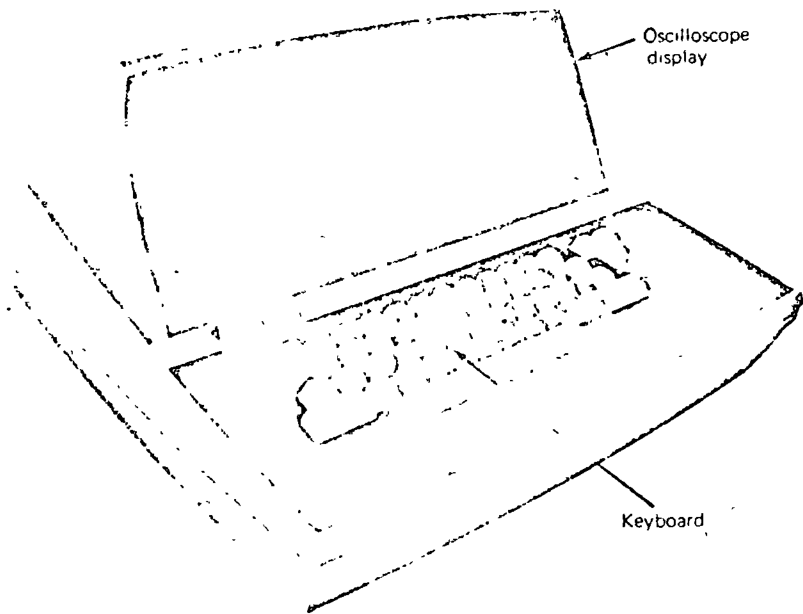


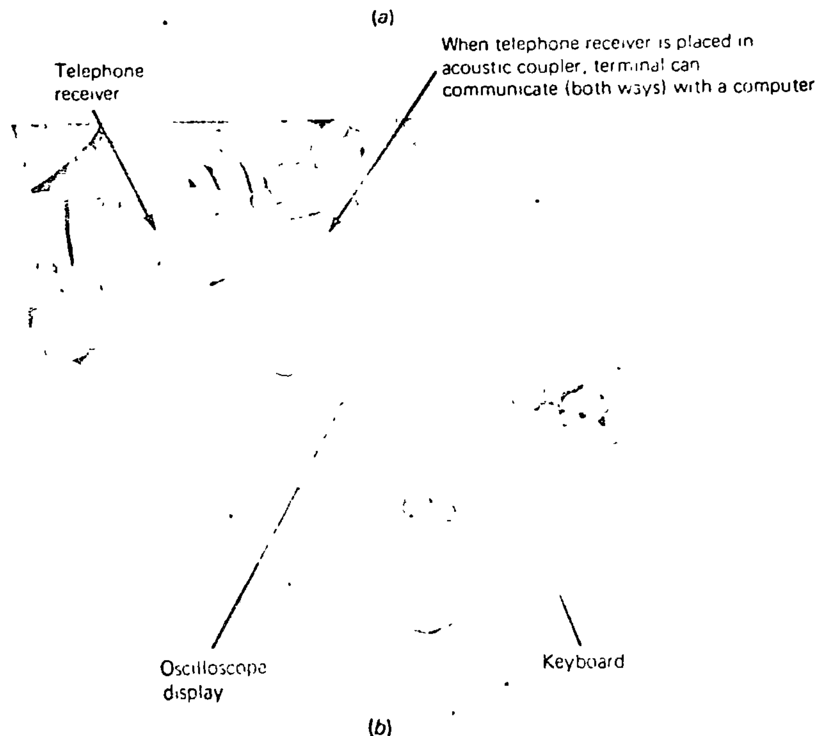
Figure 1.12 Printer for ledger cards

electrical signals representing the character on the depressed key is generated These signals are converted to a form suitable for transmission on a telephone line by a device called a *data coupler*. Figure 1 13 shows an oscilloscope display, keyboard, and a data coupler for connecting the terminal to a conventional telephone receiver At the computer, a receiver connected to the end of the telephone line converts the received signals back to a form suitable for introduction into the computer.

Similarly, signals generated by the computer are transmitted to the oscilloscope display, using a data coupler on the same telephone line at the computer.



(a)



(b)

Figure 1.13 (a) [illegible]illoscope display with keyboard (b) Terminal for connection via telepho[illegible] system

1.7 PROGRAMMING LANGUAGES

When computers were first introduced, computer users found great difficulty in preparing programs Despite the computer's speed and memory capacity, each individual instruction causes it to perform a relatively simple operation—perhaps adding or multiplying two numbers, or perhaps moving some numbers from one part of memory to another As a result, in order to perform a calculation of any complexity, quite a few instructions are required, and they must be carefully written In addition, the instructions that the computer actually performs are in what is called *binary form* and must be written in machine language, which generally seems most unnatural to humans. (This subject will be explained in detail in later chapters)

As a result, computer users wrote special programs which assisted in the writing and running of programs There were then programs of two types. (1) *Application* or *user* programs, which were the programs written by the user of the computer to solve his problems, and (2) *systems programs*,† which were programs written by professional *systems programmers* to aid the user in preparing and running his programs and in removing errors from them

The first systems programs were *translator programs* which translated the user programs from a language closely resembling machine language into actual machine language The language in which the programmer wrote his instructions before translation was called *assembly language* The systems program which converted a program written in assembly language to machine language so that it could be run on the computer was called an *assembler program* or simply an *assembler*

Having noted the advantages of assembly language over machine language, computer scientists decided it would be possible, using system programs, to perform even more complicated translations on the computer As a result, languages were invented which were called *high-level languages*‡

In order to use a high-level language, a systems program is required which translates a program written in the high-level language into a form in which it can be run on a computer A systems program which translates a *high-level language* program is referred to as a *compiler*

The best known of all high-level computer languages is Fortran, an acronym for FORmula TRANslation System This language was first specified in 1954, its developers included John Backus along with several other illustrious men Fortran is the most widely written high-level language

†Systems programs now include programs which control computer operation (called *operating systems*) programs to aid in troubleshooting when the computer develops problems and many other kinds of useful programs These will be dealt with in later sections

‡The designer of a high-level language tries to make the language translatable by any computer not just some particular computer Assembler languages are very closely welded to [illegible]icular computers, reflecting their structure

in the world, and it is remarkable that perhaps the first high-level language could be so useful and durable

The ability of Fortran to stand up through the years is a real tribute to its inventors It should be noted that the Fortran we now write has been modernized, improved, and enlarged (just as present-day English differs from elizabethan English), but most of the features in the original Fortran language remain

There are many other high-level languages (a recent survey uncovered over 500), but only a handful have been used very much. Fortran is primarily intended for scientific and engineering calculations and for general problem solving involving numeric calculations The next best known language, and the most widely used in business circles, is Cobol (for COmmon Business Oriented Language) Cobol is particularly suited to the demands of the business community and for military logistics (the language was initiated at a conference called by the Department of Defense in the Pentagon) It is designed to facilitate data processing of large files, including inventory maintenance and banking operations Whereas Fortran uses mathematical notation [A=(E+C) −D, for instance], Cobol is more like English (MULTIPLY X BY Y AND ADD Z). Further comparisons will be given after Fortran has been explained

Other high-level languages which are widely used include Basic, a computer language for use in "time-shared" or "interactive" systems, PL1, which is IBM's new language combining many of the features of Fortran and Cobol, Algol, a language invented by an international committee with some special features, and APL, a language for scientific use with strong matrix- and vector-handling capabilities Each of these is discussed in later sections The intent here is to note that there are a number of languages and that each has its advantages, disadvantages, advocates, and detractors In general, each has a particular area of application (i.e., science, business) in which it is particularly strong, and this has led to such languages being called *problem-oriented languages* since a given language is oriented toward a certain class of problems

1.8 RUNNING A PROGRAM

After a program has been written in a programming language, it must somehow be introduced into the computer There are several methods for preparing programs for a computer, and the choice of the exact method depends upon the computer input devices available and the manner in which the computer is operated This section examines several alternatives

The most used procedure for preparing programs for computer operation involves punched cards A Fortran program is normally written on a special coding form as shown in Fig 1 14. This form has 80 columns, and each line on the form thus contains up to 80 characters, each having a unique position Each line on the coding form is then punched into an


```
C    THIS PROGRAM CALCULATES THE INTEREST ON INVESTED CAPITAL
C    THE VARIABLE R GIVES THE INTEREST RATE
C    THE VARIABLE C GIVES THE AMOUNT OF CAPITAL INVESTED
C    THE VARIABLE N GIVES THE MAXIMUM NUMBER OF YEARS CONSIDERED
C    THE PROGRAM PRINTS THE INTEREST FOR EACH YEAR
      R = 0.075
      C = 1500.00
      N = 5
      DO 10 J = 1,N
      Y = (C * (1.00 + R) ** J) - C
      PRINT 15, C, R
   10 PRINT 16, J, Y
   15 FORMAT(//,18H THE INTEREST ON $,F8.2,13H AT A RATE OF,F6.3)
   16 FORMAT(14H INVESTED FOR ,I2,11H YEARS IS $,F7.2)
      STOP
      END
```

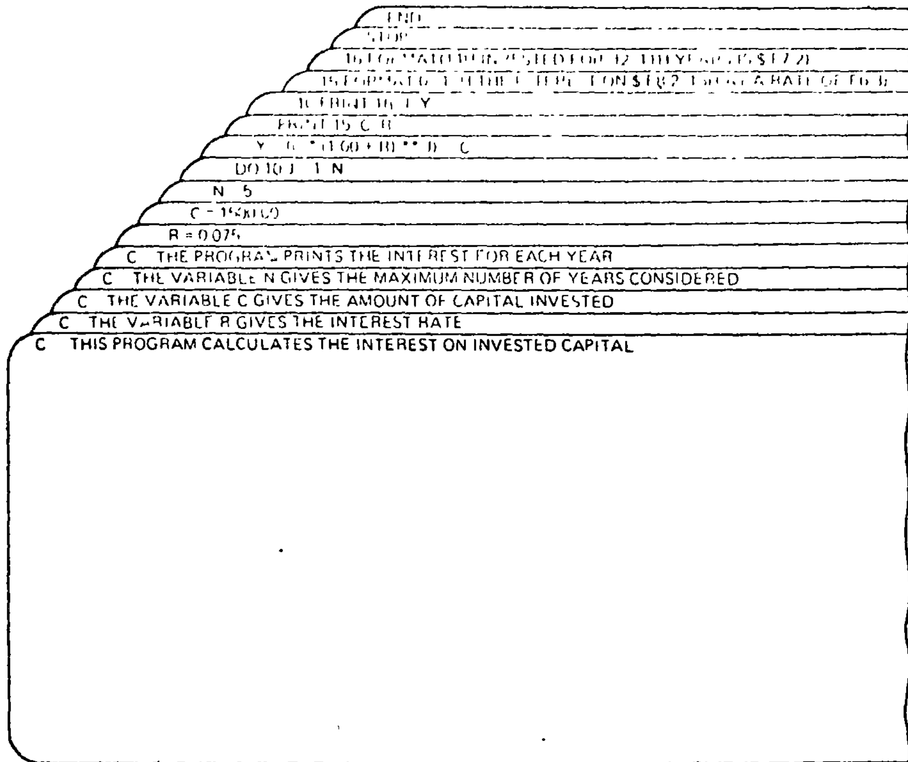
Figure 1.14 Fortran coding form

individual card The program on the form in Fig 1 14 would thus be punched into 16 cards.

Cards are prepared with a card punch as shown in Fig. 1 6. There are a number of different kinds of card punches, but each has the property that depressing a key on the punch causes the code for a character to be punched into a column of a card By "typing" a line from the coding form on the key punch, we thereby enter the line into the card, and the same line will be printed along the top of the card

Cards for the program in Fig 1.14 are shown in Fig. 1 15 These are ready to be read by a computer using a card reader There is more to operating a program than this, however. The Fortran compiler program must also be in the computer, for it must perform the necessary translation. At this point, a number of options are open. If the program is to be translated in a small system, it is possible that the Fortran compiler will first be read into the computer and the program will then be read in and compiled, and perhaps operated, under manual control. In large systems, where many programs are operated and computer time is at a premium the reading in and translation of the Fortran program is controlled by a set of systems programs which reside in the computer memory and are called the *operating system* It is the function of the operating system to sequence the operation of input programs and to call and use such translator programs as the Fortran compiler.

If we assume that an operating system is used, then we must notify the operating system programs that we wish to use the Fortran compiler We must also give the operating system certain other data concerning the program. These will include. Do we wish the translated program punched into a new set of cards so that we can operate it, without further translation,



(a)

```
THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  1 YEARS IS $ 112.50

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  2 YEARS IS $ 233.44

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  3 YEARS IS $ 363.45

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  4 YEARS IS $ 503.20

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  5 YEARS IS $ 653.44
```

(b)

Figure 1.15 (... ched cards for program (b) Printout when program in (a) is run

in the future? Do we wish to use the program at once, and have we therefore supplied data so the program can be run? If data have been supplied, where are they? Are we running more than one program, and where does our "job" end?

Details such as these are punched into what are called *system* or *job control cards*, and these cards must be read in with the program. A typical set of job control cards is shown in Fig 1.16 These cards indicate that the program is to be translated by a Fortran compiler and the translated program is to be loaded in the computer memory and immediately operated This mode of operation is called *load-and-go*.

The data which are to be used when the program is run are also punched into cards, and a control card announcing the end of the program is inserted between the program and the data A final control card announcing the "end of job" is placed at the end of the entire deck.

A complete set of control cards plus the program(s) to be run is called a *job*. The operating system controls the performance of jobs in an efficient manner. Running jobs in this way is called *batch processing*, and a group of programs (or records) to be considered as a single unit by the computer is often called a *batch*.

Control cards for the same program to be operated in load-and-go mode

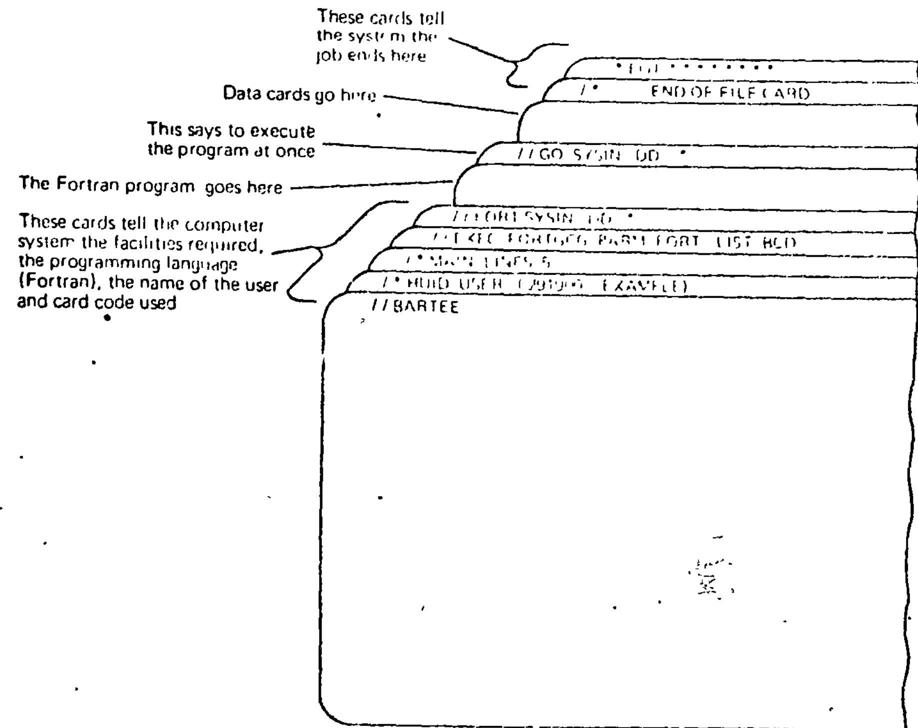


Figure 1.16 Job control cards for IBM 360/370 in OS

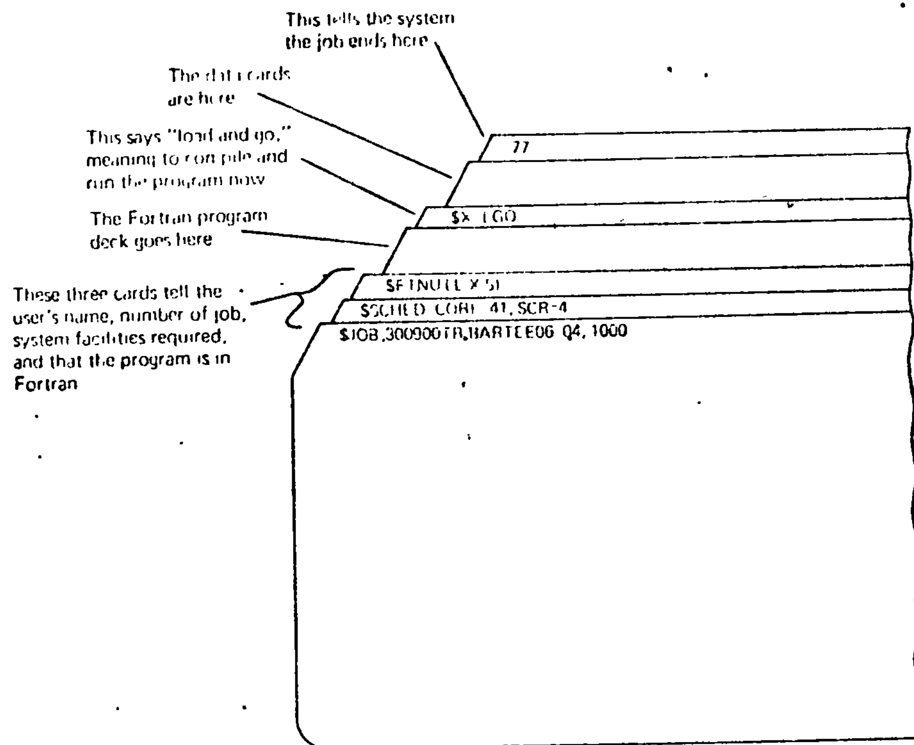


Figure 1.17 Job control cards for CDS 3300

are shown for the System/360 and /370 series OS operating system in Fig 1 16 and for the Control Data Corporation 3300 in Fig 1 17 The similarities and differences can be noted. Chapter 13 deals with operating systems in some detail

For a particular computer installation it is generally necessary to obtain information concerning the control cards to be used when operating a program from the systems programmers at that installation Thus, while a Fortran program can be run at almost any installation, it will be necessary to use different job control cards when different computer systems are used. In fact, for the System/360 and /370 there are several different operating systems (such as OS, DOS, and TOS), and for each operating system different control cards are used While this may seem to be a large problem, in actual practice it turns out to be a small inconvenience, because at most installations computer users are supplied standard sets of control cards, and use of the cards then is generally straightforward

Punched cards are by no means the only input medium which can be used to enter Fortran programs, often punched paper tape is used In this case the operator of the tape punching device "types" the program on the

keyboard, producing a punched paper tape with each character punched into a separate row.

When the tape containing a Fortran program is to be read and translated, the compiler has generally been read into the computer The actual procedure for compiling a program and running it varies widely for paper-tape systems, depending on whether the memory can contain both compiler and program, whether a disk pack or tape mechanism is available to back up the memory, whether an operating system is used, etc. Punched paper-tape systems are generally controlled by a keyboard on a console and by the computer switches, however, rather than by means of job control lines punched into the tape. The procedures to be used must be acquired at the installation.

1.9 INTERACTIVE SYSTEMS

This is still another widely used technique for introducing and operating programs. In these systems the program is keyboarded directly into the computer using a device such as the Teletype equipment in Fig 1.2. In this case the Teletype equipment is connected directly to the computer (sometimes using telephone lines), and when a key is depressed, coded signals are immediately transmitted to the computer which records them in its memory. A record of what was keyboarded is made on paper by a typing mechanism similar to a typewriter Such systems also allow the computer to type responses, however, and the interplay between computer and user has led these systems to be called *interactive systems*.

Figure 1.18 shows a record of the running of the program in Fig 1 14 on an interactive system, the General Electric Company Mark III System This particular system has Teletype keyboards in many laboratories and businesses around the country, and the computer connection is made by conventional telephone lines. To connect a given Teletype to the computer, the computer telephone number is simply dialed in conventional fashion. When the connection is made, the computer asks the user for a special code number by typing U#= on the Teletype equipment, the user then types his code number (which is CHW25125, 1KMS/STD in the figure) The computer then asks for an identification number by typing ID #, and the user responds with his ID number, in this case 2272.

The computer then lists the different compilers which are available In this case Fortran was to be used, and so after the computer typed READY, indicating that it was ready to accept a response, the user typed SYSTEM FORTRAN. After the computer read the line SYSTEM FORTRAN, it indicated it was again able to accept information (after having called the Fortran System) by typing READY. The user then asked to create a new file by typing NEW, to which the computer responded ENTER FILE NAME—and the user typed TCB1 to give the file a name.

```
*C
  U#=CH/25125,1KMS/STD
ID:2272

TYPE: SYSTEM BASIC , SYSTEM FORTRAN , SYSTEM FIV
AFTER THE COMPUTER RESPONDS: READY, TYPE: OLD OR NEW

PROGRAM STOP AT 999

USED  .14 UNITS
SYSTEM FORTRAN

READY
NEW
ENTER FILE NAME — TCB1

READY

1C THIS PROGRAM CALCULATES THE INTEREST ON INVESTED CAPITAL
2C THE VARIABLE R GIVES THE INTEREST RATE
3C THE VARIABLE C GIVES THE AMOUNT OF CAPITAL INVESTED
4C THE VARIABLE N GIVES THE MAXIMUM NUMBER OF YEARS TO BE CONSIDERED
5C THE PROGRAM PRINTS THE INTEREST FOR EACH YEAR
6          R = 0.075
7          C = 1500.00
8          N = 5
9          DO 10 J = 1, N
10         Y = (C * (1.00 + R) ** J) - C
11         PRINT 15, C, R
12   10    PRINT 16, J, Y
13   15    FORMAT(//,18H THE INTEREST ON $,F8.2,13H AT A RATE OF,F6.3)
14   16    FORMAT(14H INVESTED FOR ,I2,11H YEARS IS $,F7.2)
15         STOP;END
RUN

TCB1          13:16EDT    09/30/73

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  1 YEARS IS $ 112.50

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  2 YEARS IS $ 233.44

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  3 YEARS IS $ 363.45

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  4 YEARS IS $ 503.20

THE INTEREST ON $ 1500.00 AT A RATE OF 0.075
INVESTED FOR  5 YEARS IS $ 653.44

PROGRAM STOP AT 15

USED  .13 UNITS

BYE
0001.69 CRU  0001.16  TCH    0006.34 KC


OFF AT 13:18EDT 09/30/73
```

- Computer asks for code words and location with "U#=." User responds
- Computer types ID User types his number, in this case 2272
- Computer asks which compiler system will be used and gives highest line number available (999)
- User types SYSTEM FORTRAN
- Computer says it is ready
- User asks to start a new file
- Computer types ENTER FILE NAME—. User then types name of new file TCBI Computer then types READY
- This is the Fortran program entered by the user
- User types RUN, telling computer to run program
- Computer types file name, time, and date
- This is the output from the program
- Computer tells where program stopped
- Computer tells how much resource was used
- User signs off
- Computer lists total resources used and sign-off time and date

Figure 118 Rec[illegible] use of interactive system to run a program

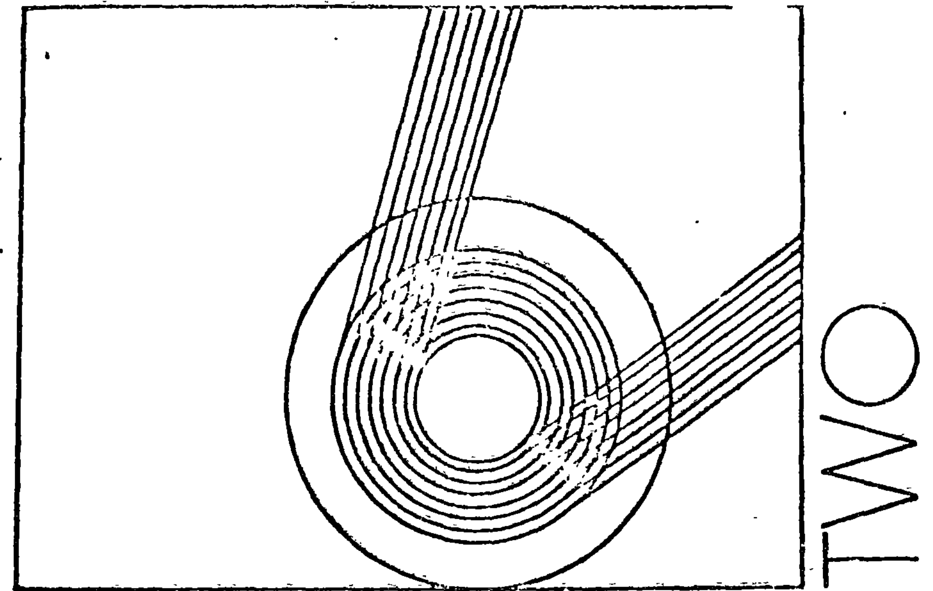
The Fortran program was then typed. After the program had been entered, the computer was directed to operate the program by the user typing the command word RUN The computer system responded by typing the name of the file, TCB1, followed by the time and date. The results of the running of the program were then printed Finally, the user "signed off" by simply typing BYE.

Interactive online systems vary from system to system, and the particular procedures to be used at each facility must be learned. The systems are really not too dissimilar from each other, however, and having learned one system, the user generally finds it easy to learn another.† The Fortran language remains basically the same from system to system—only the procedures for using the system vary.

EXERCISES

1 Discuss the terms *hardware* and *software*
2 Discuss the possible uses of a digital computer in a real-time control system of your choice What analog-to-digital conversion devices would be necessary?
3 Discuss some of the problems which might arise if a computer attempted to read handwriting in an optical character reading system Why are the MICR systems much simpler to design?
4 "A common complaint on oscilloscope displays concerns the lack of hard copy " Explain this sentence.
5 Explain the difference between systems programs and application programs
6 Explain the difference between an assembly language and an application-oriented high-level language High-level languages are often said to be "transportable " Can you explain why?
7 Are operating systems more likely to be used in small or large computer systems? Why?
8 List several advantages in using an operating system.
9 Why are job control cards used when programs are run with an operating system?
10 What are interactive systems?
11 List the results of the program run in Fig. 1.18
12 On what day and at what time was the program in Fig 1 18 run?
13 Give some advantages and disadvantages of batch processing systems as opposed to interactive systems

†Notice that each line in the Fortran program in Fig 1 18 is preceded by a line number This is a particular characteristic of the GE system (and several others) The line number is not a part of the program but simply indicates the ordering of the lines in the program



INFORMATION REPRESENTATION IN A DIGITAL COMPUTER

In order for a digital computer to execute a sequence of instructions at high speed without operator intervention, it must be able to store not only the program but also the input data and the intermediate results In many instances, the computer's memory will contain much more, often including the systems programs and sometimes other user's programs. Business systems often have very large memories with large files of data on customers, suppliers, and corporate finances As a result, the memory of a computer is a major cost item, and the widespread use of digital computers is made possible through the existence of low-cost high-speed reliable memory devices.

There is one outstanding characteristic of those computer memory devices that are reliable, fast, and relatively inexpensive. individual elements are all operated in a bistable manner.† The most familiar bistable device is probably the common switch used to turn lights (or other electrical devices) off or on (refer to Fig 2.1) Similarly, electric circuits using transistors can be reliably operated in a bistable manner in which a given circuit

†The word *bistable* means "having two stable states "

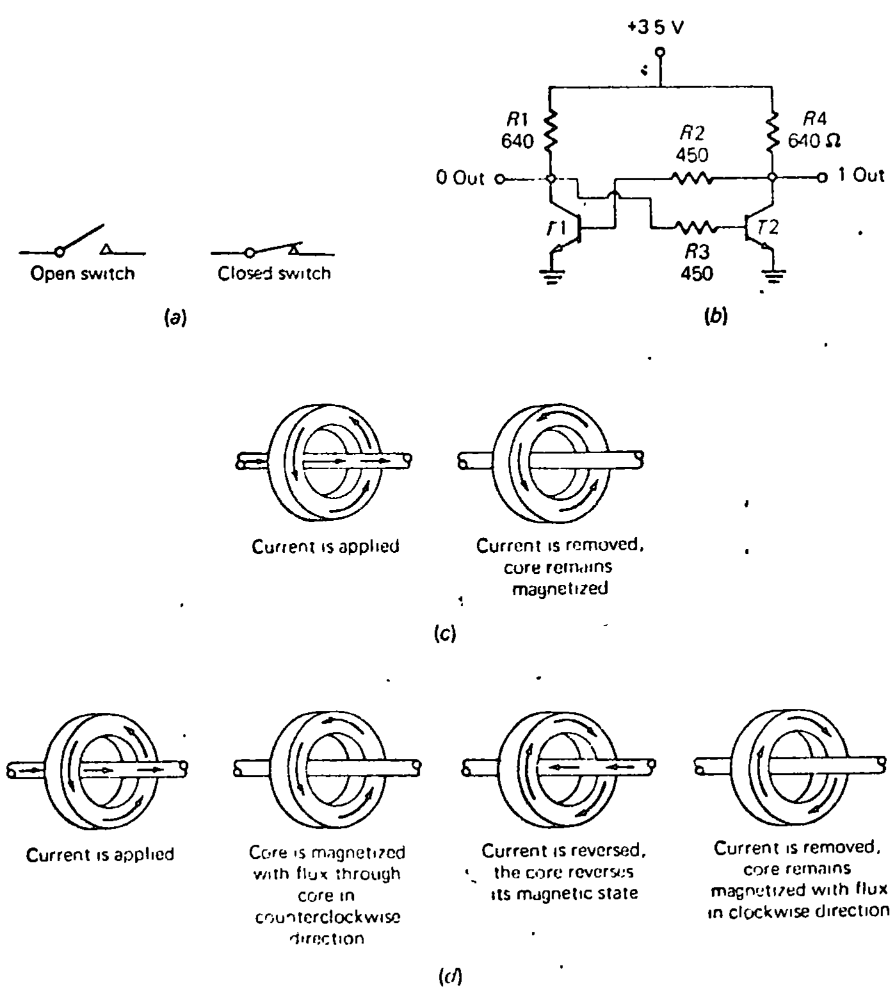


Figure 2.1 Bistable devices (a) Electric switch. (b) Two-transistor flip flop (c) Magnetizing a core. (d) Directions of magnetization.

is stable in either of two states that can be referred to as "off" or "on." The same principle applies to the magnetic cores used in high-speed memories, the "spots" on magnetic tapes which contain information, and the holes that are "punched" or "not punched" in paper tape and cards, etc. Thus, the information stored in computers is stored in devices which are operated in a bistable manner.

When information such as decimal numbers, names, and addresses must be stored in bistable devices, it is necessary to invent codes and encoding techniques so that the information can be represented by the devices. T' 's chapter describes the codes which are used, along with the organizat. of memories in computers. First, the binary number system,

which is the most natural means for representing numbers when bistabl devices are used, is described Then, more advanced number represent tion systems and techniques for representing alphabetic character punctuation marks, etc , are described

2.1 NUMBER REPRESENTATION SYSTEMS

By examination of the representation of integers in the decimal numb system an important principle can be observed, the principle of *position notation*. Consider the number 824. The use of positional notation may b plainly seen if we read this number aloud as "eight hundred and twenty four." The position of each digit in a number determines its value. the lea significant digit is the rightmost digit and the most significant the leftmos An examination indicates that the written number 824 represents, or is shorthand for, $(8 \times 100) + (2 \times 10) + 4$ or $(8 \times 10^2) + (2 \times 10^1) + (4 \times 10^0$ Similarly, 98,642 is a compact way to represent $(9 \times 10^4) + (8 \times 10^3)$ $(6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0)$

The idea of using the position of a digit to determine its value in a writte number is quite ingenious Earlier number systems did not use this princ ple: I is "one" in Roman numerals, V is "five," X is "ten," and XVI is "sixteen MCXVI is "one thousand one hundred and sixteen " Only minor rules suc as IV for "four" and VI for "six" use position to affect value Our numbe representation system is *completely* based on the use of position to dete mine value The procedures we use for adding, subtracting, multiplying and dividing decimal numbers are all based on the use of this principl and the compactness and clarity which result from positional notation plu the great efficiency of these procedures have so thoroughly establishe positional notation that we hardly notice its existence

The representation system for decimal integers using positional nota tion is based on the rule that the written number $a_m a_{m-1} \ldots a_2 a_1 a_0$ repre sents the sum $(a_m \times 10^m) + (a_{m-1} \times 10^{m-1}) + \cdots + (a_1 \times 10^1) + (a_0 \times 10^0$ Therefore, for 824 we have $8 = a_2$, $2 = a_1$, and $4 = a_0$, this represents $(8 \times 10$ $+ (2 \times 10^1) + (4 \times 10^0) = (8 \times 100) + (2 \times 10) + 4$. Similarly, 32,963 ha $a_4 = 3$, $a_3 = 2$, $a_2 = 9$, $a_1 = 6$; and $a_0 = 3$; 32,963 is the positional notatio representation for $(3 \times 10^4) + (2 \times 10^3) + (9 \times 10^2) + (6 \times 10^1) + (3 \times 10^0$ This "expanded sum of powers" is called the *literal expansion* of the num be Thus, the literal expansion of 54 is $(5 \times 10^1) + (4 \times 10^0)$

Each digit in a written decimal number is chosen from a set of 10 sym bols, which we write as 0, 1, 2, . . . , 9, and the existence of 10 different digit or symbols is the basis of the term *decimal number system*. A smaller c larger number of symbols could be used in each position, however, and fc each choice we would have a different number representation system.

The simplest of these systems is the *binary number system* which ha only two possible digits in each position, we designate them 0 and 1. Th rule for forming a binary integer is that we write $a_m a_{m-1} \ldots a_1 a_0$ as a shor

hand for $(a_m \times 2^m) + (a_{m-1} \times 2^{m-1}) + \cdots + (a_1 \times 2^1) + (a_0 \times 2^0)$, where the a's are each a 0 or a 1. For instance, the binary number 101 represents the sum $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$ which is represented by the digit 5 in the decimal system. Again, the binary number 11101 represents the sum $(1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = (1 \times 16) + (1 \times 8) + (1 \times 4) + (1 \times 1)$, which is represented by 29 in the decimal system.

Since the symbols 0 and 1 are used in both binary and decimal systems, numbers like 10 can be either binary or decimal. A notation commonly used to distinguish between binary and decimal numbers writes the base of the number as a subscript after it; hence 10_2 is "binary 2" and 10_{10} is "decimal 10."

Here are the decimal and binary representations for the first 20 integers:

$0_2 = 0_{10}$	$111_2 = 7_{10}$	$1110_2 = 14_{10}$
$1_2 = 1_{10}$	$1000_2 = 8_{10}$	$1111_2 = 15_{10}$
$10_2 = 2_{10}$	$1001_2 = 9_{10}$	$10000_2 = 16_{10}$
$11_2 = 3_{10}$	$1010_2 = 10_{10}$	$10001_2 = 17_{10}$
$100_2 = 4_{10}$	$1011_2 = 11_{10}$	$10010_2 = 18_{10}$
$101_2 = 5_{10}$	$1100_2 = 12_{10}$	$10011_2 = 19_{10}$
$110_2 = 6_{10}$	$1101_2 = 13_{10}$	$10100_2 = 20_{10}$

The positional notation system used for decimal integers extends easily to decimal fractions and in a similar manner to fractions in other bases.

The decimal fraction 0.264 has for its literal expansion the sum $(2 \times 10^{-1}) + (6 \times 10^{-2}) + (4 \times 10^{-3})$. The decimal fraction 0.23 has as its literal expansion the sum $(2 \times 10^{-1}) + (3 \times 10^{-2})$. In general, the decimal fraction $0.a_{-1}a_{-2} \ldots a_{-m}$ has as its literal expansion the sum $(a_{-1} \times 10^{-1}) + (a_{-2} \times 10^{-2}) + \cdots + (a_{-m} \times 10^{-m})$. Thus, for 0.12637 we have $a_{-1} = 1$, $a_{-2} = 2$, $a_{-3} = 6$, $a_{-4} = 3$, and $a_{-5} = 7$, the literal expansion is $(1 \times 10^{-1}) + (2 \times 10^{-2}) + (6 \times 10^{-3}) + (3 \times 10^{-4}) + (7 \times 10^{-5})$.

The same principle can be used for binary fractions. Here, negative powers of 2 instead of 10 appear in the literal expansion.

For example, 0.1011 has as its literal expansion the sum $(1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$. [This is $(1 \times \frac{1}{2}) + (0 \times \frac{1}{4}) + (1 \times \frac{1}{8}) + (1 \times \frac{1}{16})$ in decimal.] The binary fraction 0.11101 has as its literal expansion the sum $(1 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) + (0 \times 2^{-4}) + (1 \times 2^{-5})$.

In general, the binary fraction $0.a_{-1}a_{-2} \ldots a_{-m}$ has for its literal expansion the sum $(a_{-1} \times 2^{-1}) + (a_{-2} \times 2^{-2}) + \cdots + (a_{-m} \times 2^{-m})$. Thus, for the binary fraction 0.1011 we have $a_{-1} = 1$, $a_{-2} = 0$, $a_{-3} = 1$, and $a_{-4} = 1$, and the literal expansion $(1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$. (This has the value $\frac{1}{2} + \frac{1}{8} + \frac{1}{16} = \frac{11}{16}$ in decimal.)

We note the following correspondences:

$0.1_2 = 0.5_{10}$	$0.11_2 = 0.75_{10}$
$0.01_2 = 0.25_{10}$	$0.001_2 = 0.125_{10}$

2.2 ARITHMETIC OPERATIONS IN THE BINARY NUMBER SYSTEM

In this section we turn our attention to arithmetic operations in the binary number system, for it is in this system that most computers perform their operations.

The operations of addition, subtraction, multiplication, and division are straightforward in the binary number system. To add, the sums of only four combinations of 0s and 1s must be learned; they are:

$0 + 0 = 0$	$0 + 1 = 1$
$1 + 1 = 0$ plus a carry of 1	$1 + 0 = 1$

Addition of two binary numbers can be performed column by column, as in the decimal system, with carries being handled in the same manner.

$$\begin{array}{r} 101_2 \\ +1011_2 \\ \hline 10000_2 \end{array} \qquad \begin{array}{r} 1100_2 \\ +1110_2 \\ \hline 11010_2 \end{array} \qquad \begin{array}{r} 110101_2 \\ +1110_2 \\ \hline 1000011_2 \end{array}$$

To check the first addition above, note that $101_2 = 5_{10}$ and $1011_2 = 11_{10}$ and that the sum of these two integers is $10000_2 = 16_{10}$. The other two sums may be checked in a similar manner.

Again, there are four entries for subtraction.

$0 - 0 = 0$

$0 - 1 = 1$ with a borrow of 1 from the next most significant digit in the minuend

$1 - 0 = 1$

$1 - 1 = 0$

Here are four examples of subtractions. The "borrow" is handled as with decimal numbers. Again, the differences may be checked by converting to decimal numbers.

$$\begin{array}{r} 101.1_2 \\ -100.1_2 \\ \hline 1.0 \end{array} \qquad \begin{array}{r} 110.1_2 \\ -101.0_2 \\ \hline 1.1 \end{array} \qquad \begin{array}{r} 111.01_2 \\ -11.00_2 \\ \hline 100.01 \end{array} \qquad \begin{array}{r} 1010.10_2 \\ -1010.11_2 \\ \hline -0.01 \end{array}$$

The following four rules will suffice for the multiplication of binary digits. Note the reduction of the decimal multiplication table of 100 entries to four entries.

$0 \times 0 = 0$	$1 \times 0 = 0$
$0 \times 1 = 0$	$1 \times 1 = 1$

Multiplication can be performed using the same procedure as that of the decimal system. For binary, however, one simply copies the multiplicand if a multiplier digit is a 1, as an examination of the rules above will indicate.

```
  101_2        110_2         110 101_2
  110_2        111_2           11 01_2
 ------       ------        ----------
  000          110            110101
 101          110            000000
101          110            110101
------      -------        110101
11110_2     101010_2      ------------
                          10101.10001_2
```

Division may be performed using the familiar trial-and-error technique. In this case, however, each quotient digit can be only a 0 or 1, greatly simplifying the process

```
    110.1              101 01
10)1101        11.1)10010 011
   10                111
   --               ----
    10               1000
    10                111
    --               ----
    010                111
     10                111
     --                ---
```

2.3 BINARY-CODED-DECIMAL NUMBER REPRESENTATION

The electronic circuit devices used to construct digital computers are inherently binary in operation, as are the memory devices used to store information, and so the binary number system is the most natural one for a computer. On the other hand, the decimal system is what people use, and there is a natural reaction to the thought of performing calculations in a binary number system. Also, since checks, bills, tax rates, prices, etc., are all figured in the decimal system, the values of most input data to a computer must be converted from decimal to binary before computations can begin. Similarly, binary data must be converted to decimal before they can be printed or outputed in most cases.

For these and other reasons most of the early machines operated in *binary-coded-decimal* number systems. In such systems a coded group of binary bits† is used to represent each of the 10 decimal digits. For instance, an obvious and natural code is the simple "weighted binary code" shown in Table 2.1.

This is known as a binary-coded-decimal 8,4,2,1 code or simply BCD, and is the most used code at this time. Notice that 4 binary bits are required for each decimal digit, and each bit is assigned a weight, for instance, the rightmost bit has a weight of 1 and the leftmost bit in each code group has a weight of 8. By adding the weights of the positions in which 1s appear, the decimal digit represented by a code group may be derived. This is some-

†A binar[illegible] it is generally called a *bit* in computer terminology

Table 2.1 BCD Code (8,4,2,1)

Binary code	Decimal digit
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

what uneconomical since $2^4 = 16$ and therefore the 4 bits could actually represent 16 different values, but the next lesser choice, 3 bits, gives only 2^3, or 8, values, which are insufficient.

If the decimal number 214 is to be represented in this type of code, 12 binary bits are required as follows: 0010 0001 0100. For the decimal number 1,246 to be represented, 16 bits are required: 0001 0010 0100 0110.

2.4 OCTAL AND HEXADECIMAL NUMBER SYSTEMS

Two other number systems are widely used in computer science: the octal and the hexadecimal number systems.

The octal number system has a base, or radix, of 8; eight different symbols are used to represent numbers. these are commonly 0,1,2,3,4,5,6, and 7. The first few octal numbers and their decimal equivalents are as shown in Table 2.2.

To convert an octal number to a decimal number, we use the same sort of literal expansion as in the binary case, except that we now have a radix of 8 instead of 2. Therefore 1,213 in octal is $(1 \times 8^3) + (2 \times 8^2) + (1 \times 8^1) +$

Table 2.2 Octal Numbers

Octal	Decimal	Octal	Decimal
0	0	11	9
1	1	12	10
2	2	13	11
3	3	14	12
4	4	15	13
5	5	16	14
6	6	17	15
7	7	20	16
10	8	21	17

$(3 \times 8^0) = 512 + 128 + 8 + 3 = 651$ in decimal. Also, 1.123 in octal is $(1 \times 8^0) + (1 \times 8^{-1}) + (2 \times 8^{-2}) + (3 \times 8^{-3})$, or $1 + \frac{1}{8} + \frac{2}{64} + \frac{3}{512} = 1\frac{83}{512}$ in decimal.

There is a simple trick for converting a binary number to an octal number. Simply group the binary digits into sets of threes, starting at the octal point, and read each set of three binary digits according to Table 2.3.

Let us convert the binary number 011101. First we break it into threes (thus 011 101), and then, converting each group of three binary digits, we get 35 in octal. Therefore 011101 binary = 35 octal. Here are several more examples:

$$110110101_2 = 665_8$$
$$11011_2 = 33_8$$
$$1001_2 = 11_8$$
$$10101.11_2 = 25.6_8$$
$$1100.111_2 = 14.7_8$$
$$1011.1111_2 = 13.74_8$$

Conversion from decimal to octal can be performed by repeatedly dividing the decimal number by 8 and using each remainder as a digit in the octal number being formed.† For instance, to convert 200_{10} to an octal representation, we divide as follows:

$200 \div 8 = 25$	remainder is 0
$25 \div 8 = 3$	remainder is 1
$3 \div 8 = 0$	remainder is 3

Therefore $200_{10} = 310_8$

Notice that when the number to be divided is less than 8, we use 0 as the quotient and the number as the remainder. Let us check this:

†This will work for any base. To convert from decimal to base 5, repeat these instructions replacing each occurrence of 8 with 5, for example.

Table 2.3 Octal-Binary Conversion

Three binary digits	Octal digit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$$310_8 = (3_{10} \times 8_{10}^2) + (1_{10} \times 8_{10}^1) + (0_{10} \times 8_{10}^0) = 192_{10} + 8_{10} = 200_{10}$$

Here is another example. We wish to convert 3964_{10} to octal.

$3{,}964 \div 8 = 495$	with a remainder of 4
$495 \div 8 = 61$	with a remainder of 7
$61 \div 8 = 7$	with a remainder of 5
$7 \div 8 = 0$	with a remainder of 7

Therefore $7574_8 = 3964_{10}$. Checking,

$$\begin{aligned} 7574_8 &= (7_{10} \times 8_{10}^3) + (5_{10} \times 8_{10}^2) + (7_{10} \times 8_{10}) + 4_{10} \\ &= (7_{10} \times 512_{10}) + (5_{10} \times 64_{10}) + (7_{10} \times 8_{10}) + (4_{10} \times 1_{10}) \\ &= 3584_{10} + 320_{10} + 56_{10} + 4_{10} \\ &= 3964_{10} \end{aligned}$$

There are several other techniques for converting octal to decimal and decimal to octal, but they are not used very frequently manually, and tables prove to be of about as much value as anything in this process. A useful octal-to-decimal listing of values is shown in Table 2.4.

An important use for octal is in listings of programs and for memory "dumps" for binary machines, thus making the printouts more compact. Also, in keeping track of the contents of different registers and in orally conveying the contents of a binary register to someone, it is very convenient to use octal characters rather than binary. If, for instance, we want to send the binary number 011101111 over the telephone, it is easier and less conducive to error to say "three-five-seven in octal" than "zero-one-one-one-zero-one-one-one-one in binary."

The hexadecimal number system is useful for similar reasons. In many computers, including the IBM System/360 series, Data General's Nova series, the Digital Equipment Corp. PDP-11, and Honeywell Inc. computers, and in most minicomputers, the memories are organized into sets of "bytes" consisting of 8 binary digits. Each byte either is used as a single entity to

Table 2.4 Octal-Decimal Table

Octal digit position/ 8^n	Position coefficients (multipliers)							
	0	1	2	3	4	5	6	7
1st (8^0)	0	1	2	3	4	5	6	7
2d (8^1)	0	8	16	24	32	40	48	56
3d (8^2)	0	64	128	192	256	320	384	448
4th (8^3)	0	512	1,024	1,536	2,048	2,560	3,072	3,584
5th (8^4)	0	4,096	8,192	12,288	16,384	20,480	24,576	28,672
6th (8^5)	0	32,768	65,536	98,304	131,072	163,840	196,608	229,376

represent a single alphanumeric character or is broken into two 4-bit pieces (We shall examine the coding of alphanumeric characters using bytes later in this chapter)

When the bytes are handled in two 4-bit pieces, these 4-bit half-bytes are often called "nibbles," and the programmer is given the option of declaring each 4-bit character as a piece of a binary number or as two binary-coded-decimal numbers For instance, the byte 00011000 can be declared a binary number, in which case it is equal to 24 decimal, or two binary-coded-decimal characters, in which case it represents the decimal number 18.

When the computer is handling numbers in binary, but in groups of 4 digits, it is convenient to have a code for representing each of these sets of 4 digits Since there are 16 possible different numbers which can be represented, we need a base-16 number system this is called the *hexadecimal number system* The digits 0 through 9 alone will not suffice, so the letters A, B, C, D, E, and F are also used (see Table 2 5)

To convert binary to hexadecimal, we simply break a binary number into groups of 4 digits and convert each group of 4 digits according to the preceding code Thus $10111011_2 = BB_{16}$, $10010101_2 = 95_{16}$, $11000111_2 = C7_{16}$, and $10001011_2 = 8B_{16}$ The mixture of letters and decimal digits may seem strange at first, but these are simply convenient symbols, just as decimal digits are.

The conversion of hexadecimal to decimal is straightforward but time-consuming. For instance, BB represents $(B \times 16^1) + (B \times 16^0) = (11 \times 16) + (11 \times 1) = 176 + 11 = 187$. Similarly,

Table 2 5 Hexadecimal Digits

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

$$
\begin{aligned}
AB6_{16} &= (10_{10} \times 16_{10}^2) + (11_{10} \times 16_{10}) + 6_{10} \\
&= (10_{10} \times 256_{10}) + 176_{10} + 6_{10} \\
&= 2560_{10} + 176_{10} + 6_{10} \\
&= 2742_{10}
\end{aligned}
$$

To convert, for instance, $3A6_{16}$ to decimal,

$$
\begin{aligned}
3A6_{16} &= (3_{10} \times 16_{10}^2) + (10_{10} \times 16_{10}) + 6_{10} \\
&= (3_{10} \times 256_{10}) + (10_{10} \times 16_{10}) + 6_{10} \\
&= 768_{10} + 160_{10} + 6_{10} \\
&= 934_{10}
\end{aligned}
$$

Again, tables are convenient for converting hexadecimal to decimal and decimal to hexadecimal Table 2 6 is useful for converting in either direction

The chief use of the hexadecimal system is in connection with byte-organized machines.

2.5 ORGANIZATION OF MEMORIES

No single memory device so far invented has both of two desirable characteristics low cost and high speed There are high-speed memories constructed of magnetic cores and integrated semiconductor circuits, but these are not inexpensive. On the other hand, magnetic tape is very inexpensive per bit, but it is slow in operating speed There is also a class of devices such as magnetic disks and drums which are "in between," being relatively inexpensive and relatively fast.

Since a computer often needs a large memory and high speed of operation and no single device will satisfy both requirements, a compromise strategy is used where the machine has several memories, each with different characteristics

Figure 2 2 shows the basic organization of memory in a large computer (Small computers often have no backup memory.) The *general registers* are made of high-speed semiconductor electronic circuits and are used to hold operands while calculations are being performed Some computers have only one general register, while other computers have as many as 32 general registers These registers are often called *accumulators*

The *high-speed memory* usually communicates directly with the general registers, and data or operands to be used are obtained from this memory and results are stored in it The high-speed memory also contains the instruction words making up the program.

The *backup memory* consists of slower memory devices such as tape or disk storage, and these have the characteristics of low price per bit[†]

[†] A bit is a *binary digit*

Table 2.6 Hexadecimal-Decimal Table

A Integer conversion

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4.096	1	256	1	16	1	1
2	8.192	2	512	2	32	2	2
3	12.288	3	768	3	48	3	3
4	16.384	4	1.024	4	64	4	4
5	20,480	5	1 280	5	80	5	5
6	24 576	6	1.536	6	96	6	6
7	28 672	7	1.792	7	112	7	7
8	32.768	8	2 048	8	128	8	8
9	36 864	9	2 304	9	144	9	9
A	40.960	A	2 560	A	160	A	10
B	45 056	B	2 816	B	176	B	11
C	49.152	C	3 072	C	192	C	12
D	53 248	D	3 328	D	208	D	13
E	57 344	E	3 584	E	224	E	14
F	61.440	F	3 840	F	240	F	15
4		3		2		1	

Hexadecimal positions

EXAMPLE: 2322_{16} is $8192_{10} + 768_{10} + 32_{10} + 2_{10}$ = 8994.0.

B Fractional conversion

HEX	DEC	HEX	DECIMAL		HEX	DECIMAL			HEX	DECIMAL EQUIVALENT			
0	0000	00	0000	0000	000	0000	0000	0000	0000	0000	0000	0000	0000
1	0625	01	0039	0625	001	0002	4414	0625	0001	0000	1525	8789	0625
2	1250	02	0078	1250	002	0004	8828	1250	0002	0000	3051	7578	1250
3	1875	03	0117	1875	003	0007	3242	1875	0003	0000	4577	6367	1875
4	2500	04	0156	2500	004	0009	7656	2500	0004	0000	6103	5156	2500
5	3125	05	0195	3125	005	0012	2070	3125	0005	0000	7629	3945	3125
6	3750	06	0234	3750	006	0014	6484	3750	0006	0000	9155	2734	3750
7	4375	07	0273	4375	007	0017	0898	4375	0007	0001	0681	1523	4375
8	5000	08	0312	5000	008	0019	5312	5000	0008	0001	2207	0312	5000
9	5625	09	0351	5625	009	0021	9726	5625	0009	0001	3732	9101	5625
A	6250	0A	0390	6150	00A	0024	4140	6250	000A	0001	5258	7890	6250
B	6875	0B	0429	6875	00B	0026	8554	6875	000B	0001	6784	6679	6875
C	7500	0C	0468	7500	00C	0029	2968	7500	000C	0001	8310	5468	7500
D	8125	0D	0507	8125	00D	0031	7382	8125	000D	0001	9836	4257	8125
E	8750	0E	0546	8750	00E	0034	1796	8750	000E	0002	1362	3046	8750
F	9375	0F	0585	9375	00F	0036	6210	9375	000F	0002	2888	1835	9375
1		2			3				4				

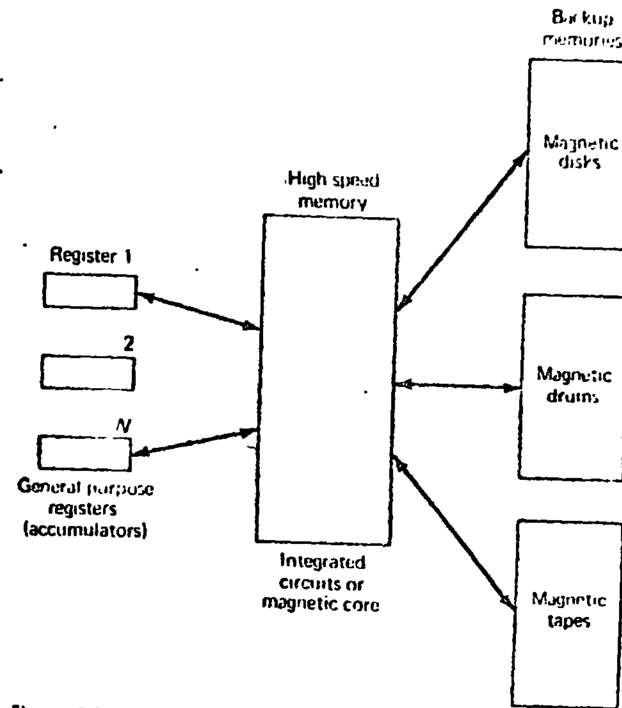


Figure 2.2 Memory organization

and high packing density. The main high-speed memory may contain from 10^5 to 10^8 bits, but the backup memories may contain 10^9 to 10^{12} bits.[†]

Computers do not handle bits one at a time. Instead, most computers perform calculations with groups of bits, each group having the same number of bits in it. A given group of bits is then called a *word*, and the number of bits in each word is called the *word length*. Thus, each computer has a basic word length.

The word length for the newer computers tends to be either 16 or 32 bits. The System/360 has a 32-bit word. The DEC PDP-11, Data General Nova; Hewlett-Packard, and Micro Data computers, with many others have 16-bit words. There are some 12-bit word computers such as the PDP-8, and some 24-bit computers including the Control Data Corporation 3100, 3200, 3300, and 3500, several computers made by the Univac Division of Sperry Rand Corp., and the Soviet K-200. Univac also has some 36-bit computers. Some of the larger CDC computers have 60-bit word lengths.

Figure 2.3 shows that the high-speed memories in computers are also organized into words of fixed lengths. The memory is divided into *n* words, where *n* generally is some power of 2, and each word is assigned an ad-

[†]A little arithmetic with high-speed memory prices of 1 cent per bit, which is quite reasonable, will show the need for slower memory devices at thousandths (disks) to hundreds of thou[illegible] of a cent per bit.

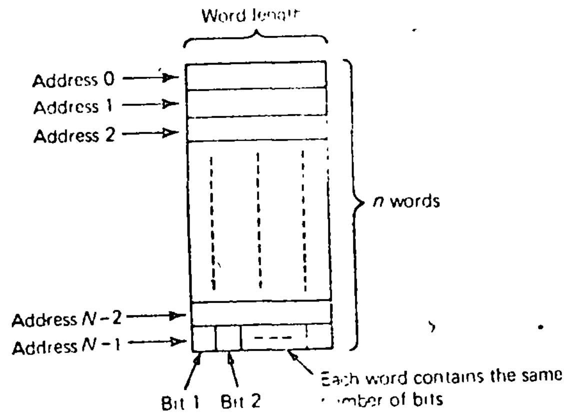


Figure 2.3 Words in high-speed memory

dress or location in the memory. Each word has the same number of bits, and if we read, for instance, the word at location 72, we shall receive a word from the memory with this number of bits

The addresses or address numbers in the memory run consecutively, starting with the address 0 and running up to the largest address Thus, at address 0 we find a word at address 1 a second word, at address 2 a third word, and so on up to the final word at the largest address

The computer can read a word from or write a word into each location in the memory For a memory with an 8-bit word, if we write the word 01001011 into memory address 17 and later read from this same address, we shall read the word 01001011 If we again read from this address at a later time (and have not written another word in), the word 01001011 will again be read This means the memory is "nondestructive read" in that reading does not destroy or change a stored word

It is important to understand the difference between the *contents* of a memory address and the address itself A memory is like a large cabinet containing as many drawers as there are addresses in memory In each drawer is a word, and the address of each word is written on the outside of the drawer If we write or store a word at address 17, it is like placing the word in the drawer labeled 17 Later, reading from address 17 is like looking in that drawer to see its contents We do not remove the word at an address when we read but change the contents at an address only when we store or write a new word

2.6 FIXED-POINT NUMBER REPRESENTATION

There are two basic techniques for representing binary numbers in a computer: the are called the *fixed-point* and the *floating-point* systems Both will be described, beginning with the fixed-point system

The most direct representation technique for numbers in a fixed-word-length computer consists of simply storing positive integers in each word in binary form Thus, in a 16-bit-per-word computer a positive integer could be stored with value from 0 to $2^{16}-1$ This is not very satisfactory, however, since it is almost invariably necessary to represent and handle both positive and negative integers This leads to setting aside the leftmost bit in each word and calling this the *sign bit* Our first computer representation system is therefore called the *sign-plus-magnitude integer system*.

Sign-Plus-Magnitude Integer System

Suppose each word consists of only 5 binary digits If the leftmost digit is used to indicate sign, then the remaining 4 digits can be used to indicate the magnitude of the integer stored It is common practice to let a 0 in the sign bit indicate a positive number and a 1 a negative number Thus, for our 5-bit word, 0.0000 would indicate (positive) zero The word 0.0001 would be "plus 1" and 0.0010 would be "plus 2." Notice that the sign bit is set apart from the magnitude bits by a . in each word This is due to custom, and, although the . has the disadvantage of looking like a binary point, it rarely leads to misunderstandings An alternate technique uses a box for the sign bit so that 0.0001 would be [0]0001, and another system uses an S over the sign bit so that 0.0010 would be written $\overset{s}{0}0010$ We shall use the . symbol to set the sign bit apart This . is not in the computer, of course, but is simply a notational convenience

In the sign-plus-magnitude integer system with 5-bit words 1.0001 is "minus 1" and 1.0010 is "minus 2." Table 2.7 shows several words and their values in decimal.

Table 2.7 Sign-Plus-Magnitude Numbers in Computer Words

Computer word	Decimal value
0.1111	+15
0.1001	+9
0.1000	+8
0.0010	+2
0.0001	+1
0.0000	0
1.0001	−1
1.0010	−2
1.0100	−4
1.1000	−8
1.1001	−9
1.1111	−15

In general, an n-bit word using sign-plus-magnitude integer representation can represent an integer from $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$

1s Complement Integer Representation System

The sign-plus-magnitude system described above was used in several early computers. It has the appeal of being quite simple to understand. Its drawback is that in order to actually add and subtract, the computer must "work harder" than when either of two other representation systems, 1s complement and 2s complement, are used. This means that the circuits for handling numbers are simplified if 1s or 2s complement systems are used, and as a result one of these is almost always adopted.

The 1s complement system again uses the leftmost bit as the sign bit. All positive numbers are represented just as in the sign-plus-magnitude system. Thus, 0.0011 would again be $+3_{10}$, and 0.0100 would be $+4_{10}$. Negative numbers are represented differently, however. The negative of each number is formed by subtracting the value of each bit in the word from 1. This effectively changes the value of each bit, for 0 from 1 is 1 and 1 from 1 is 0.† For instance, $+4_{10}$ is 0.0100; to form -4_{10}, we change each bit to get 1.1011. Similarly, $+3_{10}$ is 0.0011; to form -3_{10}, we complement each bit and get 1.1100.

Figure 2.4 shows the basic format for a 5-bit word in the 1s complement integer system. Integers from $+(2^{n-1}-1)$ to $-(2^{n-1}-1)$ can be represented by an n-bit word.

2s Complement Signed-Integer System

When the 2s complement signed-integer system is used, the sign bit again occupies the leftmost position. A 0 in this bit again indicates that the integer is positive, and 0.0001 again means $+1_{10}$; 0.0010 is again $+2_{10}$, and all positive numbers are represented as they are in the signed-magnitude or 1s complement systems. Negative numbers, however, are formed by complementing each bit and then adding 1 in the least significant position; this means forming the 1s complement and then adding 1. For instance, to form the negative, or 2s complement, of $+4_{10}$, which is 0.0100, we complement each bit, getting 1.1011, and then add 1 to get 1.1100, which is the 2s complement representation for -4_{10}. To form -3_{10}, we 2s complement 0.0011 by first forming the 1s complement 1.1100, then add 1 to get 1.1101, which is the 2s complement of 0.0011 and has value -3_{10}.

Figure 2.5 shows the 2s complement system for a 5-bit word in more detail.

†This is called *complementing*. The complement of 1 is 0, and the complement of 0 is 1.

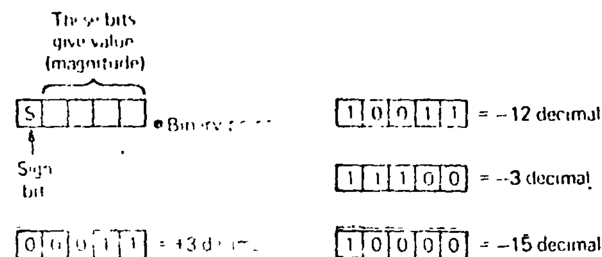


Figure 2.4 1s complement integer representation

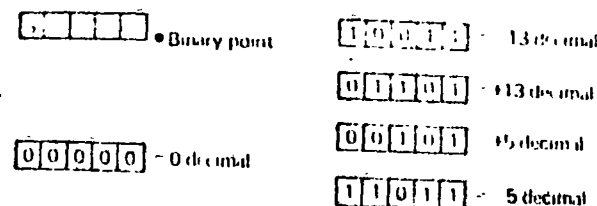


Figure 2.5 2s complement integer representation

*2.7 COMPLEMENTED NUMBER SYSTEMS

The preceding section described the 1s and 2s complement number systems. These are widely used in computers for they make it possible to add and subtract signed numbers in a simple way. To explain this, the general principle will be demonstrated using decimal numbers; then the binary case will be discussed.

In general, there are two useful complements for each number in a given number system. These are called the *true complement* and the *radix-minus-1 complement*.

True Complement

The true complement of a number represented by $a_m a_{m-1} \ldots a_0$ in a number system with a radix of r is formed by subtracting each digit of the number from $r-1$ and forming a new number of these differences. A 1 is then added to this number. Thus, the true complement of $a_m a_{m-1} \ldots a_0$ is a number $(b_m b_{m-1} \ldots b_0) + 1$

where
$$b_m = (r-1) - a_m$$
$$b_{m-1} = (r-1) - a_{m-1}$$
$$\cdots\cdots\cdots\cdots$$
$$b_0 = (r-1) - a_0$$

Thus, for the number 2102 in octal, (with a radix of 8) the true complement would be formed by letting $a_3 = 2$, $a_2 = 1$, $a_1 = 0$, and $a_0 = 2$, and using the rule above. We then get $b_3 = (8-1) - 2$, $b_2 = (8-1) - 1$, $b_1 = (8-1) - 0$, and $b_0 = (8-1) - 2$. The true complement of 2102 is therefore 5675 + 1, which is 5676.

Radix-Minus-1 Complement

The radix-minus-1 complement of a number written in a system with a radix of r is formed by simply subtracting each digit in the number from the value $r - 1$ and writing this difference in the same position of the new number. That is, a number $a_1 a_2 \ldots a_n$ with a radix r has radix-minus-1 complement $b_1 b_2 \ldots b_n$, where $b_1 = (r - 1) - a_1$, $b_2 = (r - 1) - a_2$, etc. For instance, in octal, let the number $a_2\, a_1\, a_0$ be 235, then the true complement is 542 for $r - 1$ is $8 - 1 = 7$ and $7 - 2$ is 5, $7 - 3$ is 4, and $7 - 5$ is 2. Thus, $a_2 = 2$, $a_1 = 3$, $a_0 = 5$, and $b_2 = 7 - 2$, $b_1 = 7 - 3$, and $b_0 = 7 - 5$.

For the decimal number system with a radix of 10, the two basic complements of a given number are called the 10s complement (which is the true complement) and the 9s complement (which is the radix-minus-1 complement).

The 9s complement of a given number may be formed by subtracting each digit in the number from 9. For instance, the 9s complement of 21 is 78, the 9s complement of 323 is 676, and the 9s complement of 524 is 475.

The 9s complement can be used in subtraction by means of a simple trick. To subtract one integer from another larger integer, simply form the 9s complement of the integer to be subtracted and then add this to the other number. Any carry which is generated when the two most significant digits are added must be added to the lowest digit of the result. This is called an *end-around-carry*.

For example:

```
Normal subtraction      9s complement subtraction

   45                                   45
  -32                                 + 67
  ---                                 ----
   13                                 1 12
                   end-around-carry   └→ 1
                                      ----
                                        13

   76                                   76
  -27                                 + 72
  ---                                 ----
   49                                 1 48
                   end-around-carry   └→ 1
                                      ----
                                        49
```

As has been described, the 10s complement of any number may be formed by subtracting each digit of the number from 9 and then adding 1 to the least significant digit of the number thus formed. For instance, the 10s complement of 87 is 13 and the 10s complement of 23 is 77. Subtraction may be performed by simply adding the 10s complement of the subtrahend to the minuend and discarding the final carry from the most significant digits, if a carry occurs. For instance:


```
Normal subtraction     10s complement subtraction

   89                  89      89
  -23                 -23  =  +77
  ---                         ----
   66                      ┌─1 66
                           └→the carry is dropped

   98                  98      98
  -87                 -87  =  +13
  ---                         ----
   11                      ┌─1 11
                           └→the carry is dropped
```

According to the rule given at the beginning of this section, the 2s complement of a binary number is formed by simply subtracting each digit (bit) of the number from the radix-minus-1 and adding a 1 to the least significant bit. Since the radix in the binary number system is 2, each bit of the binary number is subtracted from 1. The application of this rule is actually very simple: Every 1 in the number is changed to a 0 and every 0 to a 1. A 1 is then added to the least significant bit of the number formed. For instance, the 2s complement of 10110 is 01010, and the 2s complement of 11010 is 00110. Subtraction using the 2s complement system involves forming the 2s complement of the subtrahend and then adding this "true complement" to the minuend. For instance:

```
  11011        11011              11100       11100
 -10100  =    +01100     and     -00100  =   +11100
 ------       ------             ------      ------
  00111    ┌─ 1 00111             11000   ┌─ 1 11000
           └→ the carry is dropped         └→ dropped
```

Subtraction using the 1s complement system is also straightforward. The 1s complement of a binary number is formed by changing each 1 in the number to a 0 and each 0 in the number to a 1. For instance, the 1s complement of 10111 is 01000, and the 1s complement of 11000 is 00111.

When subtraction is performed in the 1s complement system, any end-around-carry is added to the least significant bit. For instance:

```
  11001       11001              11110       11110
 -10110  =   +01001     and     -01101  =   +10010
 ------      ------             ------      ------
  00011   ┌─ 1 00010             10001   ┌─ 10000
          └────────→ 1                   └────────→ 1
             ------                         ------
              00011                          10001
```

The representation of positive and negative integers in binary form generally involves use of the 2s complement system. Computers of many major manufacturers, including Hewlett-Packard, DEC, IBM, and Data General, use 2s complement integer representation. The DEC PDP-11, all Hewlett-Packard computers, the IBM 1130, all Data General computers, and many others have 16-bit word lengths.

In a 16-bit word computer which stores numbers in 2s complement, the decimal number +10 would be stored as 0 000000000001010, the number −10 would be 1 111111111110110 As may be seen, a computer with a 16-bit word using the 2s complement system can store integers from -2^{15}, which is −32,768, to $+(2^{15}-1)$, which is 32,767.

In general, a binary computer with an n-bit word length which stores numbers in the 2s complement system can represent numbers from -2^{n-1} to $+(2^{n-1}-1)$ For instance, the System/360 series has a 32-bit word length and integers, giving it a maximum magnitude of $2^{31}-1$, or 2,147,483 647.

It is possible to extend the range of the integers in a machine by using what is called *multiple-precision arithmetic*. In this case several registers are used to represent a single integer (a number). The basic operations of addition, subtraction, multiplication, and division are then performed by *multiple-precision routines*, which are programs that calculate on rules in several words instead of one word *Double-precision arithmetic* is arithmetic using two words per number,[†] and *triple-precision arithmetic* uses three words per number

It sometimes happens that the result of a calculation is a number exceeding the magnitude that can be stored Such a number is then said to *overflow* If we add the integer $+2^{15}$ to $+2^{15}$ in one of the 16-bit machines previously discussed, the result would be 2^{16}, too large to be represented.[‡] The computer circuits then give an overflow indication, which may be used by the operating system or other programs. They will communicate the fact that an overflow has occurred to the computer user. This process will be described later.

2.8 FLOATING POINT NUMBER SYSTEMS

The preceding sections describe representation systems whereby positive and negative integers are stored in binary words. In the representation system used, the binary point is "fixed" in that it lies at the end of each word, and so each value represented is an integer When computers calculate with binary numbers in this format, the operations are called *fixed-point arithmetic*

In science, it is often necessary to calculate with very large or very small numbers. Scientists have therefore adopted a convenient notation in which a *mantissa* plus an *exponent* are used to represent a number For instance, 4,900,000 may be written as 0.49×10^7, where 0 49 is the mantissa and 7 is the value of the exponent, or 0 00023 may be written as 0.23×10^{-3} The notation is based on the relation $y = a \times r^p$, where y is the number to be represented, a is the mantissa, r is the base of the number system ($r = 10$ for decimal, and $r = 2$ for binary), and p is the power to which the base is raised

[†]This rule is not always strictly enforced Double-precision numbers may have more or less than twice the precision in some computers

[‡]This is so because the sign bit occupies one bit in the word, leaving only 15 for magnitude

It is possible to calculate with this representation system To multiply $a \times 10^n$ times $b \times 10^m$, we form $(a \times b) \times 10^{m+n}$. To divide $a \times 10^m$ by $b \times 10^n$, we form $a/b \times 10^{m-n}$. To add $a \times 10^m$ to $b \times 10^n$, we must first make m equal to n. If $m = n$, then $a \times 10^m + b \times 10^n = (a + b) \times 10^m$. The process of making m equal to n is called *scaling* the numbers

Considerable "bookkeeping" can be involved in scaling the numbers, and there can be difficulty in maintaining precision during computations when the numbers vary over a very wide range of magnitudes. For computer usage, these problems are alleviated by means of two techniques whereby the computer (not the programmer) keeps track of the radix (decimal) point, automatically scaling the numbers. In the first, programmed *floating-point routines* automatically scale the numbers used during the computations while maintaining the precision of the results and keeping track of the scale factors. These routines are used with small computers having only fixed-point operations. A second technique lies in building what are called "floating-point operations" into the computer's hardware. The logical circuitry of the computer is then used to perform the scaling automatically and to keep track of the exponents when calculations are performed To effect this, a number representation system called the *floating-point system* is used

A *floating-point number* in a computer uses the exponential notation system described above, and during calculations the computer keeps track of the exponent as well as the mantissa A computer number word in a floating-point system may be divided into three pieces· the first is the sign bit, indicating whether the number is negative or positive, the second part contains the exponent for the number to be represented, and the third part, the mantissa

As an example, let us consider a 12-bit word length computer with a floating-point word. Figure 2 6 shows this It is common practice to call the exponent part of the word the *characteristic* and the mantissa section the *integer part*, we shall adhere to this practice

The *integer part* of the floating-point word shown represents its value in *signed-magnitude form* (rather than 2s complement, although this has been used). The characteristic is also in signed-magnitude form The value of the number expressed is $I \times 2^C$, where I is the value of the integer part and C is the value of the characteristic

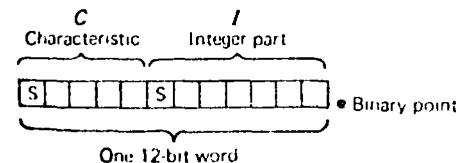


Figure 2.6 12-bit floating-point word

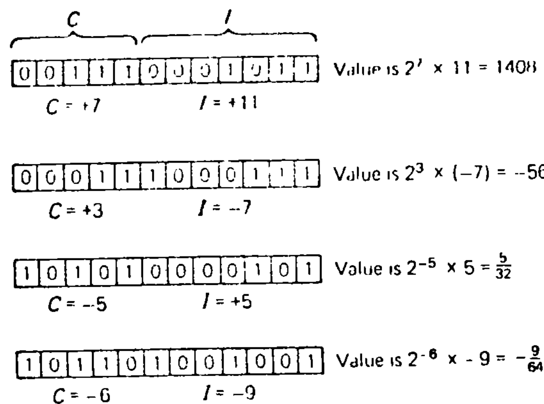


Figure 2.7 Values of floating-point numbers in 12-bit all-integer systems

Figure 2.7 shows several values of floating-point numbers in both binary form and after being converted to decimal. Since the characteristic has five bits and is in signed-magnitude form, the C in $I \times 2^C$ can have values from -15 to $+15$. The value of I is a sign-plus-magnitude binary integer of 7 bits, and so I can have values from -63 to $+63$. The largest number represented by this system would have a maximum I and would be 63×2^{15}. The least number would be -63×2^{15}.

This example shows the use of a floating-point number representation system to store "real" numbers of considerable range in a binary word.

One other widely followed practice is to express the mantissa of the word as a fraction instead of as an integer. This is in accord with common scientific usage since we commonly say that 0.93×10^4 is in "normal" form for exponential notation (and not 93×10^2). In this usage, a mantissa in decimal normally has a value from 0.1 to 0.999 . . . . Similarly, a binary mantissa in normal form would have a value from 0.5 (decimal) to less than 1. Most computers maintain their mantissa sections in normal form, continually adjusting words so a significant (1) bit is always in the position next to the sign bit.

When the mantissa is in fraction form, this section is called the *fraction*. For our 12-bit example we can express floating-point numbers with characteristic and fraction by simply supposing the binary point to be to the left of the magnitude (and not to the right as in integer representation). In this system a number to be represented has value $F \times 2^C$, where F is the binary fraction and C is the characteristic.

For the 12-bit word considered before, fractions would have values from $1 - 2^{-6}$, which is 0.111111, to $-(1 - 2^{-6})$ which is 1.111111, thus numbers from $(1 - 2^{-6}) \times 2^{15}$ to $-(1 - 2^{-6}) \times 2^{15}$ can be represented, or about +32,000 to −32,000. The smallest value the fraction part would have is now [illegible] fraction [illegible] 1.000000, which is 2^{-1} times the smallest characteristic, which is [illegible], so the smallest positive number representable is $2^{-1} \times 2^{-15}$

or 2^{-16}. Most computers use this fractional system for the mantissa, although computers of Burroughs Corporation and the National Cash Register Co. use the integer system previously described.

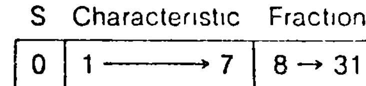
In computers using 16-bit words (including some made by DEC, Hewlett-Packard, Data General, and IBM), floating-point words are represented by two adjacent words and thus have 32 bits per word. The actual format for floating-point words for several of these computers is shown in Fig. 2.8. In these computers the fraction part F consists of 24 bits representing a 23-bit fraction and a sign bit. The exponent or characteristic consists of 8 bits. (In Hewlett-Packard computers the fraction part and the characteristic part are represented in 2s complement form, as programmed for Fortran.) Each of these computers, as programmed for Fortran, can represent magnitudes of up to 2^{127} (or about 10^{38}) and fractions of about as small as 2^{-138} (about 10^{-38}).

As an example of a computer with internal circuitry which performs floating-point operations and uses a single computer word representation of floating-point numbers, we turn to the System/360 and /370.

IBM calls the exponent part the *characteristic* and the mantissa part the *fraction*. In the /360 and /370 series floating-point data words can be either 32 or 64 bits in length. The basic formats are as follows:

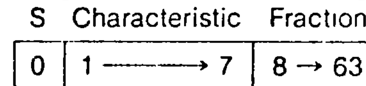
Short or single word floating-point number

S Characteristic Fraction



Long or double word floating-point number

S Characteristic Fraction



In both cases the sign bit S is in the leftmost position and gives the sign of the number. The characteristic part of the word then comprises bits 1–7 and is simply a binary integer, which we shall call I, ranging from 0 to 127. The actual value of the scale factor, or characteristic, is formed by subtracting 64 from this integer I and raising 16 to this power. Thus the

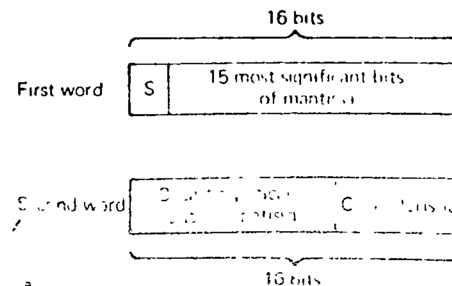


Figure 2.8 Floating point representation using two words

value 64 in bits 1–7 gives a scale factor of $16^{I-64} = 16^{64-64} = 16^0$; a 93 (decimal) in bits 1–7 gives a scale factor of $16^{I-64} = 16^{93-64}$ which is 16^{29}, and a 24 in bits 1–7 gives a 16^{-40}.

The magnitude of the actual number represented in a given floating-point word is equal to this scale factor times the fraction contained in bits 8–31 for the short number, or 8–63 for a long number. The radix point is assumed to be to the left of bit 8 in either case, so if bits 8–31 contain 1000 . . . 00, the fraction has value $\frac{1}{2}$ (decimal); that is the fraction is .1000 . . 000 in binary. Similarly, if bits 8–31 contain 11000 . . . 000, the fraction value is $\frac{3}{4}$ decimal, or .11000 . . . 000 binary.

The actual number represented then has magnitude equal to the value of the fraction times the value given by the characteristic. Consider a short number:

	Sign	Characteristic	Fraction
Floating-point no.	0	1 0 0 0 0 0 1	1 1 1 0 0 . . 0
Bit position	0	1 2 3 4 5 6 7	8 9 10 11 12 . . . 31

The sign bit is a 0, and so the number represented is positive. The characteristic has binary value 1000001, which is 65 decimal and so the scale factor is 16^1. The fraction part has value .111 binary, or $\frac{7}{8}$ decimal, and so the number represented is $\frac{7}{8} \times 16$, or 14 decimal.

Again, consider the following number:

	Sign	Characteristic	Fraction
Floating-point no	1	1 0 0 0 0 0 1	1 1 1 0 0 . . . 0
Bit position	0	1 2 3 4 5 6 7	8 9 10 11 12 . . . 31

This has value −14 since every bit is the same as before except the sign bit (The number representation system is signed magnitude.)

As a further example

Sign	Characteristic	Fraction	
0	1 0 0 0 0 1 1	1 1 0 . . . 0	$16^3 \times \frac{3}{4} = 3072$
0	0 1 1 1 1 1 1	1 1 0 . . . 0	$16^{-1} \times \frac{3}{4} = \frac{3}{64}$

*2.9 PERFORMING ARITHMETIC OPERATIONS WITH FLOATING-POINT NUMBERS

A computer obviously requires additional circuitry to handle floating-point numbers automatically. Some machines come equipped with floating-point instructions. (For computers such as DEC PDP-11/45 and others, floating-point circuitry can be purchased and added to enable them to perform floating-point operations.)

To handle the floating-point numbers, the machine must be capable of extensive shifting and comparing operations. The rules for multiplying and dividing are

$$(a \times r^p) \times (b \times r^q) = ab \times r^{p+q}$$

$$(a \times r^p) \div (b \times r^q) = a/b \times r^{p-q}$$

The computer must be able to add or subtract the exponent sections of the floating-point numbers, and also perform the mutliplication or division operations on the mantissa sections of the numbers. In addition, precision is generally maintained by shifting the numbers stored until significant digits are in the leftmost sections of the word. With each shift, the exponent must be changed. If the machine is shifting the mantissa section left, for each left shift the exponent must be decreased.

For instance, in a binary-coded-decimal computer, consider the word

0	10	0064
Sign	Exponent	Mantissa

To attain precision, the computer shifts the mantissa section left until the 6 is in the most significant position. Since two shifts are required, the exponent must be decreased by 2, and the resulting word is 0 08 6400. If all numbers to be used are scaled in this manner, the maximum precision may be maintained throughout the calculations.[†]

For addition and subtraction, the exponent values must agree. For instance, to add 0.24×10^5 to 0.25×10^6, we must scale the numbers so that the exponents agree. Thus

$$(0.024 \times 10^6) + (0.25 \times 10^6) = 0.274 \times 10^6$$

The machine must also follow this procedure. The numbers are scaled as was described, so that the most significant digit of the computer mantissa section of each word contains the most significant digit of the number stored. Then the larger of the two exponents for the operands is selected, and the other number's mantissa is shifted and its exponent adjusted until the exponents for both numbers agree. The numbers may then be added or subtracted according to these rules.

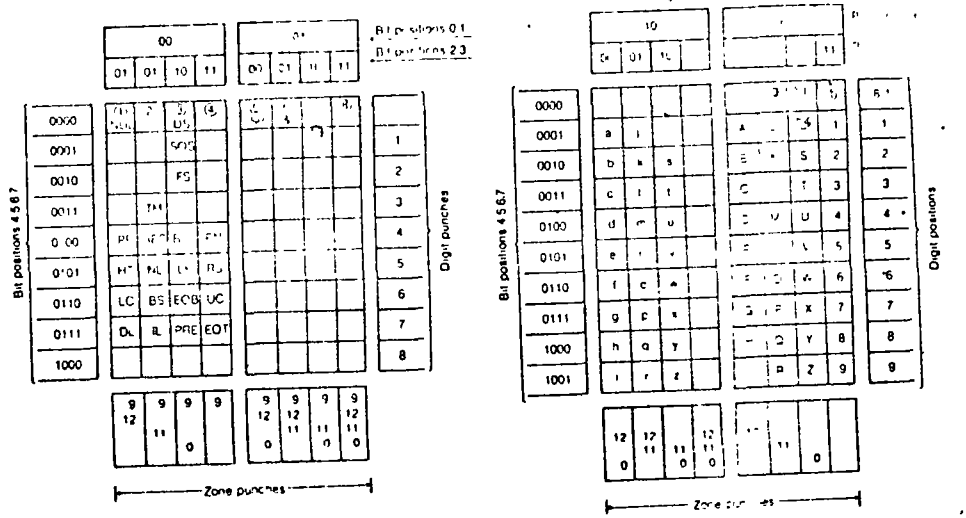
$$(a \times r^p) + (b \times r^p) = (a + b) \times r^p$$

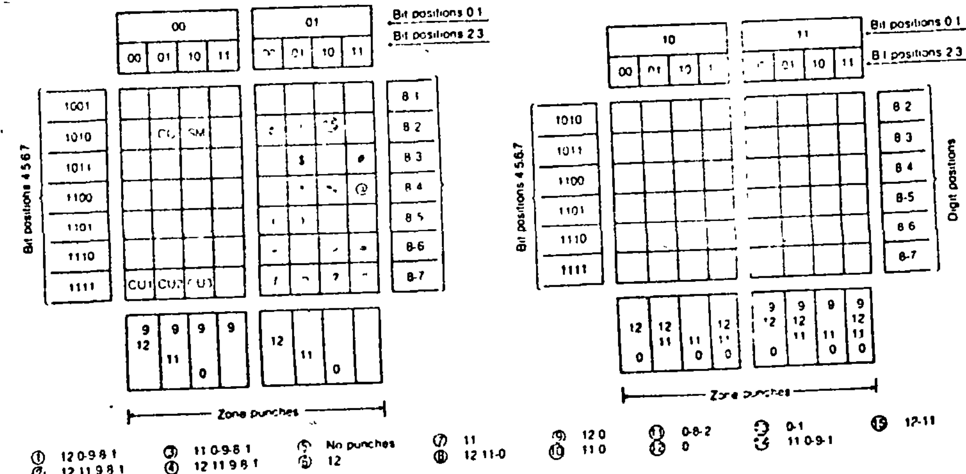
$$(a \times r^p) - (b \times r^p) = (a - b) \times r^p$$

2.10 ALPHANUMERIC CHARACTERS

Computers must handle not only numbers but also alphabetic and special symbols. To make this possible, special codes have been designed

[†]A floating point number with its leftmost digit nonzero is said to be *normalized*. Shifting the mantissa [illegible] until a nonzero digit is reached is called *normalization*. For the System/360 and /370 a nonzero digit is [illegible] into one of the first 4 bits since the scale factor is 16^I, not just I and shifts are made four at a time





① 12 0-9 8 1 ② 12 11 9 8 1 ③ 11 0-9 8 1 ④ 12 11 9 8 1 ⑤ No punches ⑥ 12 ⑦ 11 ⑧ 12 11-0 ⑨ 12 0 ⑩ 11 0 ⑪ 0-8-2 ⑫ 0 ⑬ 0-1 ⑭ 11 0-9-1 ⑮ 12-11

Control Character

NUL	Null	BS	Backspace	EOB	End of block
PF	Punch off	IL	Idle	PRE	Prefix
HT	Horizontal tab	CC	Cursor control	SM	Set mode
LC	Lower case	CU2	Reserved for customer use	CU3	Reserved for customer use
DL	Delete	DS	Digit select	PN	Punch on
CU1	Reserved for customer use	SOS	Start of significance	RS	Reader stop
TM	Tape mark	FS	Field separator	UC	Uppercase
RES	Restore	BYP	Bypass	EOT	End of transmission
NL	New line	LF	Line feed	SP	Space

Special Graphic Characters

¢	Cent sign	*	Asterisk	>	Greater than sign
.	Period, decimal point	)	Right parenthesis	?	Question mark
<	Less than sign	;	Semicolon	:	Colon
(	Left parenthesis	¬	Logical NOT	#	Number sign
+	Plus sign	-	Minus sign, hyphen	@	At sign
|	Vertical bar, logical OR	/	Slash	'	Prime, apostrophe
&	Ampersand	,	Comma	=	Equal sign
!	Exclamation point	%	Percent	"	Quotation mark
$	Dollar sign	_	Underscore		

Example	Type	Bit pattern Bit position 01 23 4567	Hole pattern Zone punches	Digit punches
PF	Control character	00 00 0100	12 9	4
%	Special graphic	01 10 1100	0	8 4
R	Upper case	11 01 1001	11	9
a	Lower case	10 00 0001	12 0	1
	Control character	00 11 0000	12 11-0 9	8 1

Figure 2.9 Extended binary-coded ...

whereby characters are represented or coded by fixed numbers of bits An alphanumeric character is a single digit (0 1, ... 9), a letter, or, for most codes, a special character such as ?, $, ',), (, etc An *alphanumeric code* is (for the computer industry) a means of representing alphanumeric characters with binary digits If we are to represent decimal digits and letters, at least 6 binary digits are required per character The EBCDIC code used 6 bits per character and has been used in a number of computers This code is shown in Fig 2.9

Another popular code is the American Standard Code for Information Interchange, which is referred to as ASCII. (This is probably the most used code at this time) Each alphanumeric character is represented by 8 bits, and there are a number of special characters and control characters. This code is shown in Fig 2.10 Since 8 bits are required for each character,

Least Significant Digits (columns); Most Significant Digits (rows)

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NULL	SOH	STX	ETX	EOT	ENQ	ACK	BELL	BS	HT	LF	VT	FF	CR	SO	SI
0001	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SS	ESC	FS	GS	RS	US
0010																
0011																
0100	SP	!		#	$	%	&		(	)	*	+		-		/
0101	0	1	2	3	4	5	6	7	8	9				-	.	?
0110																
0111																
1000																
1001																
1010	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1011	P	Q	R	S	T	U	V	W	X	Y	Z	[		]		
1100																
1101																
1110	\	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
1111	p	q	r	s	t	u	v	w	x	y	z	{	|	}		DEL

NULL	NULL	VT	Vertical tabulation	NAK	NACK negative acknowledge
SOH	Start of heading	FF	Form feed	SYN	SYNC-synchronous idle
STX	Start of text	CR	Carriage return	ETB	End of transmission
ETX	End of text	SO	Shift out	CAN	CNCL cancel
EOT	End of transmission	SI	Shift in	EM	End of medium
ENQ	Enquiry	DEL	Delete	SS	Start of special sequence
ACK	Acknowledge	DLE	Data link escape	ESC	Escape
BELL	BELL audible signal	DC1	Device control	FS	File separator
BS	Backspace	DC2	Device control	GS	Group separator
HT	Horizontal tabulation	DC3	Device control	RS	Record separator
LF	Line feed	DC4	Device control	US	Unit separator

Figure 2.10 The American Standard Code ...

the 16-bit-word computers commonly handle two characters per word, and the 32-bit-word machines store 4 characters per word. Thus, to store the name "John Jones" in ASCII in a 16-bit-word computer would require five words, and in a 32-bit computer would require three words. The term *byte* is commonly used to refer to an 8-bit set of bits. A single byte, therefore, can store a single character. To facilitate the handling of characters, the memories of some computers are arranged so that bytes can be addressed individually.

Since data are commonly read in and printed in alphanumeric form, special programs must be written for inputting and outputting these characters. Computers commonly have instructions which are especially designed for handling 6- or 8-bit alphanumeric characters.

EXERCISES

1 Convert the following decimal numbers to equivalent binary numbers.
   (*a*) 17 (*b*) 15 (*c*) 19 (*d*) 5 (*e*) 25 (*f*) 32 (*g*) 64 (*h*) 127 (*i*) 254

2 Convert the following decimal numbers to equivalent binary numbers
   (*a*) 12 (*b*) 0.125 (*c*) 0.775 (*d*) 4.625 (*e*) $3\frac{3}{8}$ (*f*) $2\frac{5}{16}$

3 Convert the following binary numbers to equivalent decimal numbers.
   (*a*) 1011 (*b*) 1001 (*c*) 1111 (*d*) 1110 (*e*) 11011 (*f*) 1010101

4 Convert the following binary numbers to equivalent decimal numbers.
   (*a*) 0.011 (*b*) 0.0101 (*c*) 1.1010 (*d*) 101.101 (*e*) 0110.0111 (*f*) 1101.001101

5 Perform the following additions and check by converting the binary numbers to decimal
   (*a*) 1001.1 + 1011.01 (*b*) 100101 + 100101 (*c*) 0.1011 + 0.1101 (*d*) 1011.01 + 1001.11

6 Perform the following additions and check by converting the binary numbers to decimal
   (*a*) 1101.1 + 1011.1 (*b*) 101101 + 110110 (*c*) 0.0011 + 0.1110 (*d*) 1100.011 + 1011.011

7 Perform the following subtractions in binary and check by converting the numbers to decimal and subtracting
   (*a*) 1011 − 1000 (*b*) 1101 − 1011 (*c*) 1011.1 − 111.1 (*d*) 1011.01 − 1011.1 (*e*) 111.1 − 111.11 (*f*) 1101.1 − 1110

8 Perform the following subtractions in the binary number system
   (*a*) 128 − 32 (*b*) $\frac{1}{8} - \frac{1}{16}$ (*c*) $2\frac{1}{8} - 4\frac{1}{32}$ (*d*) $31 - \frac{7}{8}$ (*e*) $63 - 31\frac{1}{16}$ (*f*) 129 − 33

9 Perform the following multiplications and divisions in the binary number system.
   (*a*) 16 × 8 (*b*) 31 × 7 (*c*) 23 × 2.5 (*d*) 15 × 7.625 (*e*) 6 ÷ 3 (*f*) 16 ÷ 4.

10 Perform the following multiplications and divisions in the binary number system.
   (*a*) 14 × 2.75 (*b*) 18 ÷ 9 (*c*) $256\frac{1}{2} \div 128\frac{1}{4}$ (*d*) 31.5 ÷ 15.75 (*e*) $\frac{3}{8} \div 3$ (*f*) $1125 - 3\frac{3}{8}$

11 Convert the following decimal numbers into both their 9s and 10s complements.
   (*a*) 97 (*b*) 79 (*c*) 0.82 (*d*) 0.18 (*e*) 297.64 (*f*) 334.73

12 Convert the following binary numbers into both their 1s and 2s complements.
   (*a*) 110 (*b*) 101 (*c*) 1101 (*d*) 1110 (*e*) 10110 (*f*) 11011

13 Perform the following subtractions using both 9s and 10s complements
   (*a*) 8 − 4 (*b*) 16 − 8 (*c*) 198 − 124 (*d*) 28.5 − 23.4 (*e*) 27.6 − 23.4 (*f*) 0.55 − 0.42

14 Perform the following subtractions using both 9s and 10s complements.
   (*a*) 948 − 234 (*b*) 347 − 263 (*c*) 349.5 − 245.3 (*d*) 412.7 − 409.2

15 Perform the following subtractions of binary numbers using both 1s and 2s complements
   (*a*) 101 − 100 (*b*) 110 − 11 (*c*) 110 − 0.1 (*d*) 0.11 − 0.101 (*e*) 0.111 − 0.11 (*f*) 11.11 − 0.1

16 How many different binary-coded-decimal numbers can be stored in a register containing 12 switches using an 8, 4, 2, 1 code?

17 How many different binary-coded-decimal numbers can be stored in 16 switches? (Assume two-position or ON-OFF switches.)

18 Using the 1s complement number system, perform the following subtractions.
   (*a*) 0.1001 − 0.0110 (*b*) 0.1110 − 0.0110 (*c*) 0.01111 − 0.01001 (*d*) 11011 − 11001 (*e*) 1110101 − 1010010

19 Perform the following subtractions in the binary number system using 2s complements.
   (*a*) 1011 − 1101 (*b*) 111 − 1101 (*c*) 1011.11 − 101.01 (*d*) 111.1 − 110.1

20 Convert the following hexadecimal numbers to decimal numbers

CHAPTER 1

INTRODUCTION TO MINICOMPUTATION

A NEW TOOL OF EXTRAORDINARY POWER

1-1. Where Computers Fit. Scientists and engineers describe, measure, interpret, and predict the outside world in terms of idealized **mathematical models**, which relate numerical quantities and truth values through various mathematical operations. **Computers are physical systems designed to implement mathematical models** and to automate their manipulation. Professional workers are likely to meet computers in the following roles:

1. **Numerical problem solving and data processing:** This ranges from little slide-rule and calculator jobs to big number-crunching projects and includes design calculations, statistics, genetics calculations, bookkeeping, etc. The end product may be scientific or clerical *description*, but, in the long run, calculations usually serve for *making decisions*.
2. **Storing, retrieving, sorting, and updating data:** This is by no means restricted to numerical data only.
3. **Computer simulation:** We use the convenient, easy-to-change "live mathematical model" for *experiments* which might be slow, expensive, unsafe, or impossible with the real-world system or situation being simulated. Computer simulation serves the purposes of design, systems research, education, training, and play; simulation experiments or tests sometimes involve parts of real systems.
4. **"Real-time" or "on-line" computing devices** serve as components of control and instrumentation systems to:
   (*a*) Implement desired mathematical relations between physical variables (e.g., function generation, filtering, prediction, optimization)

(*b*) Control timing and logical sequencing of operations and experiments.

The latter types of operations are often combined with on-line record keeping (data logging) and data processing.

5. Real-time timing, switching, coding, and data storage in communication systems, especially in communications between digital computers and/or computer terminals.

1-2. The Role of Small Computers. Conventional number-crunching calculations are traditionally performed by large digital computers operating in a batch mode, i.e., efficiently fed with a more or less continual and orderly sequence of things to do. In this type of application, experience appears to indicate that "throughput" (defined as the number of computer operations, in some specified mix, per unit time) increases better than proportionally with computer cost (*Grosch's law*), so that large computer systems are economical.

Other computer applications involve several users who would not like to wait for batch-processed results but need quick *"conversational" input and output* from multiple computer terminals. Again, *control and instrumentation work* is timed by the demands of real-world events. Large digital machines subject to such random service requests cannot afford to wait idly until action is required: they must be *time-shared* among multiple programs. Time-sharing operations can utilize the resources of ever larger (and thus presumably more efficient) computer systems. Time sharing also involves serious *overhead costs* resulting from communications with remote interfaces, from greatly complicated system programming, and from the many computer operations needed to swap and protect programs. *We can, then, find applications where multiple small computers can neatly replace or complement large machines.*

We will (quite arbitrarily) define a minicomputer as a digital computer whose "minimum configuration" (4,000 words of memory, teletypewriter) costs under $20,000 and which usually employs short computer words (to 18 bits, Sec. 1-3) to represent data and computer instructions (see also Secs. 1-5 and 2-1); minicomputer cost is usually roughly proportional to word length. As we shall see, 16 bits can represent numerical *data* with enough precision for many applications, but clever utilization of the short *instruction* words is the central problem of minicomputer system design (Sec. 2-5 and Chap. 6). An n-bit instruction word can specify at most 2^n different instructions. $2^{16} = 65{,}536$ looks like a very large number of possible instructions, but many of these instructions must specify the source or destination of an operand in a computer memory having perhaps 8,000 locations. This need for address specification greatly reduces the effective number of different one-word minicomputer instructions.

Nevertheless, even 8- and 12-bit minicomputers with fairly primitive instruction sets are very versatile, since multiple instructions can implement extremely complex operations. Such machines now replace hard-wired special-purpose logic in many real-time applications such as operation sequencing, timing, production testing, and data logging. Custom-designed hardware is then replaced by the quantity-produced minicomputer, which can be programmed and reprogrammed for a huge variety of different applications and new conditions.

Minicomputers are especially suitable for operations involving external real-world devices (Fig. 1-2 and Chap. 7) because:

1. Many jobs of this type do not require elaborate processor circuits.
2. We want no big expensive central processor standing idle during input/output operations.

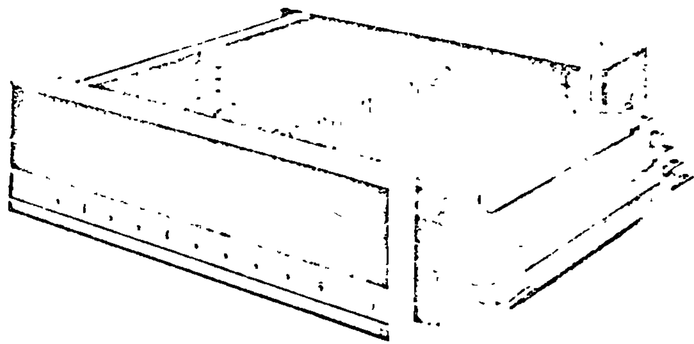
For precisely the same reasons, minicomputers can also relieve large digital computers of input/output and communications-handling chores.

Many of the more recent small processors are in no sense primitive (Chaps. 2 and 6). The truly revolutionary advance and acceptance of the newer minicomputers stems from the mass production of new integrated circuits, which have radically reduced processor and memory costs. We actually have a twofold effect:

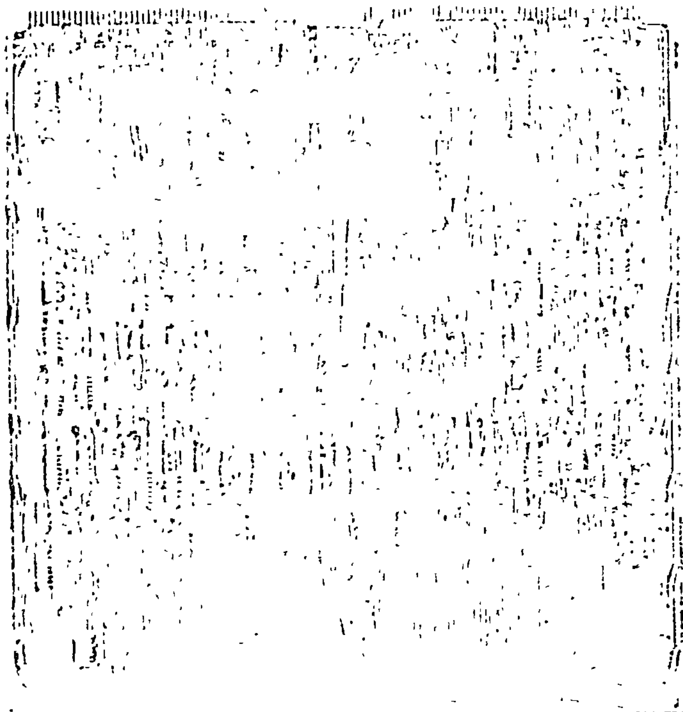
1. Medium-scale integration (MSI) of multiple logic functions on small silicon chips permits inexpensive construction of very fast and remarkably sophisticated miniprocessors (Fig. 1-1).
2. Inexpensive and much faster core and semiconductor memories make it less painful to use extra instruction words for improved instruction sets and addressing schemes (Sec. 2-7 and Chap. 6).

These two developments have given astonishing capabilities to the new small machines. While the majority of minicomputers continue to serve as *special-purpose* computers in control, instrumentation, and communications, an increasing proportion are employed in *general-purpose* computation and simulation. Minicomputers work especially well with conversational terminals, graphic displays, and all kinds of instruments. Operating inefficiencies can be tolerated; a small computer can "belong" to a small group of researchers or engineers rather than to a computer-center bureaucracy, and it is possible to modify programs (and even hardware!) without collapsing a large organization.

The situation is *not* one-sided. A reasonable end-user installation might require not only a $12,000 minicomputer but between $10,000 and $70,000 worth of computer *peripherals* (tape drives, disks, displays, printer, card reader—these are, unfortunately, not grown on monolithic silicon chips). Maintenance must be provided or paid for. Altogether, the economics of



(a)



(b)

Fig. 1-1*a* to *c*. Figure 1-1*a* shows a complete 16-bit digital computer for desk-top or rack mounting. The entire central processor, built with medium-scale-integrated-circuit chips, fits a single etched-circuit board, and so does each 4K memory module (*b* and *c*). The small machine, which can control (or receive information from) hundreds of external devices, has an instruction-cycle time of 800 nsec with a core memory or 300 nsec with a semiconductor memory. (*Data General Corporation.*)

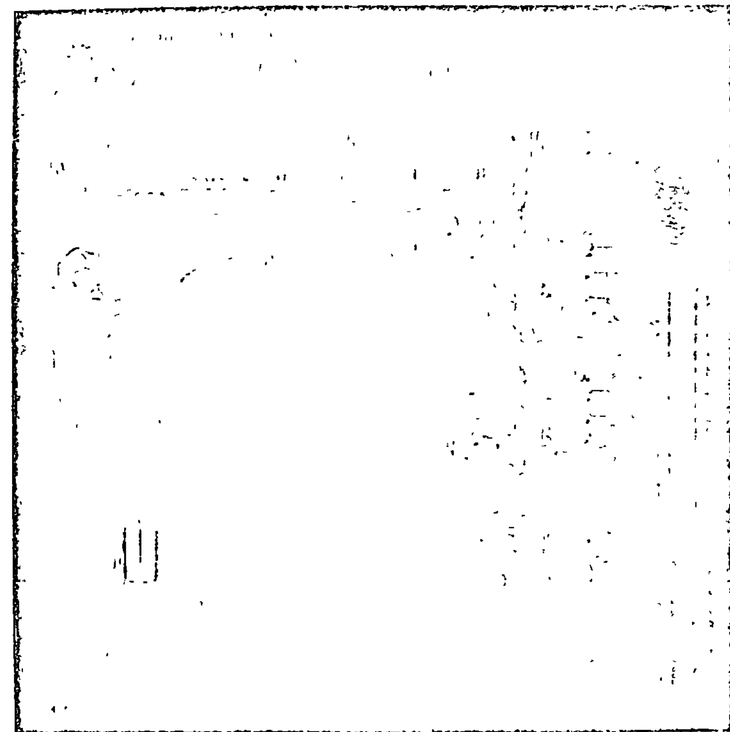


Fig. 1-1*c*.

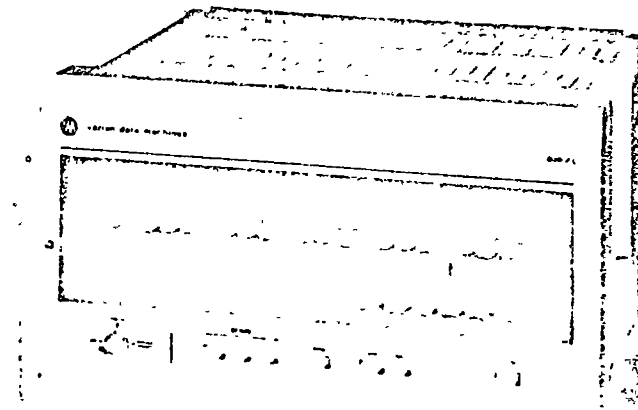


Fig. 1-1*d*. With two accumulators, an index register, and one of the most comprehensive instruction sets available, the 16-bit Varian Data Machines 620L/100 costs only $5,400 for the basic processor and 4K words of 0.95-μsec memory; each additional 4K words costs $2,300. (*Varian Data Machines.*)

multiple minicomputers versus large time-shared multiprocessors are always good for an extended argument and depend on the specific utilization of specific installations. Even so, the question merits frequent reexamination since, while minicomputers and their peripherals are becoming cheaper and more powerful, the same is true for the new multiprocessor computer utilities and their formidable system software. Data communication, moreover, is sure to be improved over the pitiful telephone-based systems of the early 1970s.

In any case, the inexpensive, small, and light minicomputer, with its dramatic versatility, is surely the world's finest toy for innovators and experimenters. It opens unheard-of horizons in control and instrumentation. On experimenters' desks or wheeled carts, in large and small factories, and in ships or aerospace vehicles, minicomputers permit intelligent automation of all sorts of operation-sequencing and data-gathering operations and generate convenient displays for human operators. Physical interfacing of small digital computers and much real-world apparatus is quite easy (Chap. 5). The larger job of computer programming for a wide variety of applications is simplified by new system and application software (Chaps. 3 and 4).

DIGITAL-COMPUTER REPRESENTATION OF DATA AND TEXT

1-3. Binary and Digital Variables. While an analog computer represents problem variables by continuously variable physical quantities such as voltages or currents (Fig. 1-3*a*), a digital computer represents problem variables by physical quantities capable of taking only discrete and countable sets of values. Thus, the original "digital-computer user" long ago employed his fingers to count and add external objects, first up to five and then up to ten. The overwhelming majority of electronic digital computers, however, implements a binary representation in terms of basic variables which can take only two different states called (logical) 0 and 1. Most frequently, the state 1 is indicated by the presence of a voltage, usually 3 or 4 volts, on a line associated with a variable, while logical 0 is indicated by the absence of that voltage (Fig. 1-3*b*). Many other pairs of voltage levels, such as 0 and −3 volts or −1.75 and −0.5 volts, are also employed to represent 0 and 1.

Such binary variables, which involve only the presence or absence of a signal, are especially easy to generate, transmit, and store reliably. We pay for this great convenience, however: most problem situations or variables admit a much greater variety of possible states than just two and must, therefore, be labeled (represented) in terms of ordered *combinations* of binary variables. We must, then, develop binary codes which associate problem

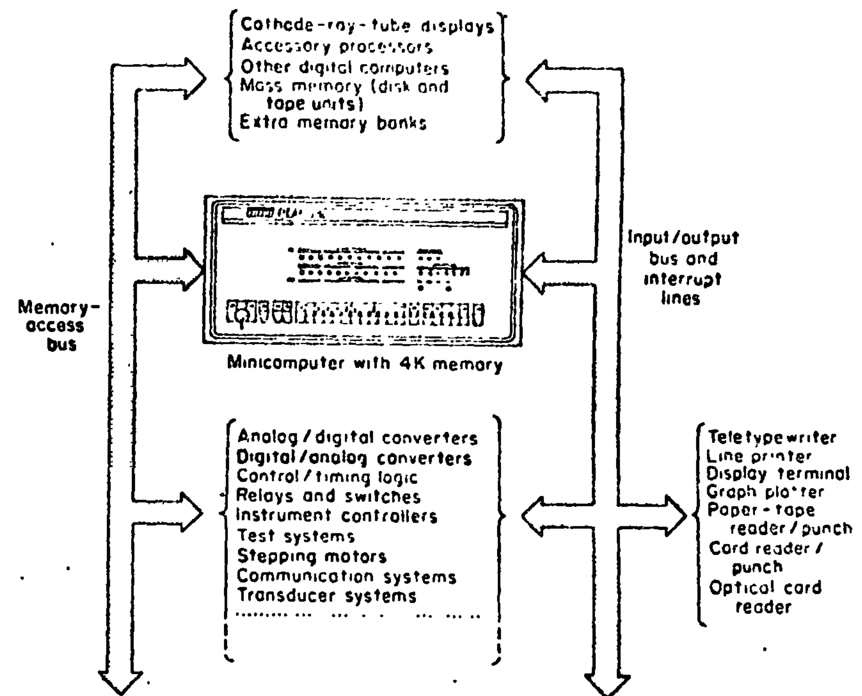


Fig. 1-2. The minicomputer input/output world.

states, messages, or numerical quantities with corresponding digital words, i.e., ordered sets of binary variables. Such codes may differ for different applications. In Fig. 1-3*c*, *four* binary variables specify which of four circuits is energized, while in Fig. 1-3*d two* binary variables control the same situation.

It is, of course, especially important to represent real numbers in terms of binary variables. In general, a real integer m will require at least $\log_2 m$

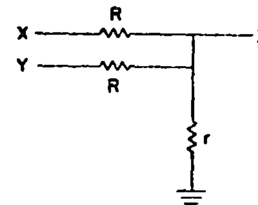


Fig. 1-3*a*. Simple analog computation. The summing network produces the output voltage $Z = \alpha(X + Y)$, where X and Y are input voltages and $\alpha = (R/r + 2)^{-1}$

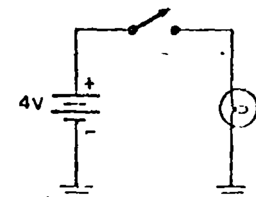


Fig. 1-3*b*. Logic states represented by voltage levels in an elementary digital circuit.

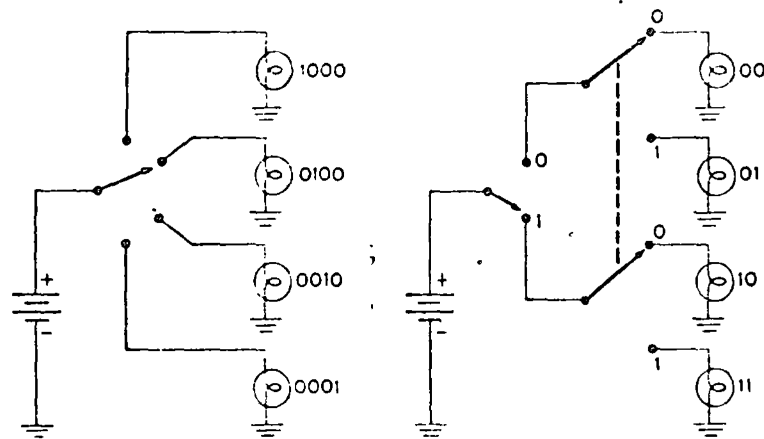


Fig. 1-3c and d. Two different binary representations of a switch setting energizing one of four circuits

binary units of information or **bits**, plus a **sign bit** to tell whether the integer is positive or negative (Secs. 1-4 and 1-6).

It is common practice to designate an entire set of binary variables (which may or may not represent a numerical quantity) as a single **digital variable.** The different bits of such a digital variable may appear on parallel bus lines (**parallel representation**) and may be stored in a *register* like that of the toggle switches in Fig. 1-4*a*. The different bits of a digital variable could also follow each other in time as consecutive samples of a voltage waveform which can take the values corresponding to 0 or 1 (serial representation, Fig. 1-4*b*). Parallel representation, which can transmit all the bits of a word at the same time, is clearly faster and is most frequently employed in modern digital computers. Serial representation, on the other hand, simplifies long-distance data communication.

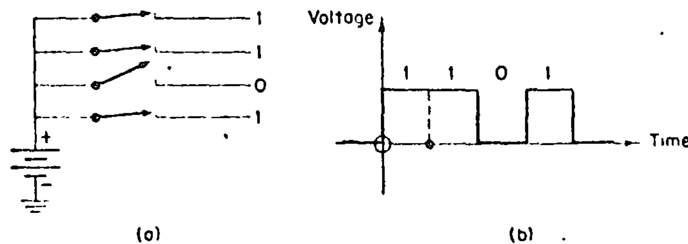


Fig. 1-4. Representation of the digital word 1101 by simultaneous levels on a parallel bus (*a*) and by [illegible] serial pulses (*b*).

1-4. Digital-computer Representation of Numbers and Characters. (a) Binary Numbers. The n-bit binary word $(a_0, a_1, \ldots, a_{n-1})$, where a_k is either 0 or 1, can be interpreted as a one-to-one representation (binary code) for the nonnegative integers

$$X = 2^{n-1}a_0 + 2^{n-2}a_1 + \cdots + a_{n-1} \qquad 0 \le X \le 2^{n-1} - 1 \quad (1\text{-}1)$$

The integers [Eq. (1-1)], in turn, constitute a numerical code for the n-bit words $(a_0, a_1, a_2, \ldots, a_{n-1})$, whose original interpretation may not be numerical, e.g., a set of truth values. Table 1-1 and Sec. 1-9 further describe the n-bit binary codes for negative integers and for fractions most commonly employed in digital-computer arithmetic.

(b) Octal and Hexadecimal Numbers. Binary words are convenient for machines but not for people. To obtain a nice shorthand notation, we split a given binary word into *3-bit groups* (starting with a_{n-1}) and write the binary number corresponding to each group as an octal digit between 0 and 7:

	a_0 ↘				↙ a_{n-1}
binary word	11	111	001	000	010
octal word	3	7	1	0	2
	↗ A_0				↖ A_{m-1}

The resulting **octal word** $(A_0, A_1, A_2, \ldots, A_{m-1})$ represents the integer [Eq. (1-1)] in the form

$$X = 8^{m-1}A_0 + 8^{m-2}A_1 + \cdots + A_{m-1}$$
$$0 \le X \le 2^{n-1} - 1;\ m < \frac{n}{3} + 1 \quad (1\text{-}2)$$

The code we have just defined will describe every binary word as a *nonnegative octal integer* [Eq. (1-2)], even though the binary word may represent a negative number, a fraction, or a nonnumerical quantity such as a set of truth values or a text character. There is little need to learn octal *complement codes* for negative numbers since minicomputer assembly languages (Sec. 3-5) which accept negative octal integers automatically translate ordinary sign-and-magnitude notation, e.g.,

$$-400_8 = -256_{10}$$

into binary code (see also Sec. 4-2). Computer *output* is usually in decimal form, except for some debugging and troubleshooting programs (Sec. 3-17). It is, however, useful to know the octal code for *nonnegative pure fractions* encoded in binary forms as $(a_0, a_1, a_2, \ldots, a_{n-1})$ (a_k = 0 or 1), with a binary point implied ahead of the most significant bit a_0 (Table 1-2).

TABLE 1-1. Binary Codes Representing Real Integers X by n-bit Words $(a_0, a_1, a_2, \ldots, a_{n-1})$

Each binary digit a_k is either 0 or 1. a_0 is the most significant bit (MSB), and a_{n-1} is the least significant bit (LSB).

1. Nonnegative Integers: $X = 2^{n-1}a_0 + 2^{n-2}a_1 + \cdots + a$ where $0 \le X \le 2^n - 1$.

EXAMPLE (3 bits):

DECIMAL	BINARY	DECIMAL	BINARY
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

This is the conventional binary code used in most minicomputers. Many other codes exist. In particular, in the *Gray code* (Ref. 6) only one binary digit changes each time X is incremented; this is useful for encoding certain instrument outputs.

Binary decimal conversion is easiest with *decimal/octal tables* (Table A-3), or use the "doubling and dabbling" recursion (Ref. 4) $X = X_{n-1}$, where $X_0 = a_0$, $X_i = 2X_{i-1} + a_i$, and $i = 1, 2, \ldots, n-1$; e.g.,

$$a_i \to 1 \quad 0 \quad 1 \quad 1$$

$$X_i \to 1 \quad 2 \quad 5 \quad 11 \qquad X = 11$$

2. Signed Integers (positive, negative, or zero). The sign bit a_0 is 0 for $X \ge 0$ and 1 for $X < 0$

(*a*) **Sign-and-magnitude Code:** $X = (-1)^{a_0}(2^{n-2}a_1 + 2^{n-3}a_2 + \cdots + a_{n-1})$, where $1 - 2^{n-1} \le X \le 2^{n-1} - 1$. There are *two* binary representations of 0: 000 . . . and 100 . . . . This can cause complications, e.g., in statistical work. This code is often used in digital voltmeters.

(*b*) **1s-complement Code:** $X = (1 - 2^{n-1})a_0 + 2^{n-2}a_1 + 2^{n-3}a_2 + \cdots + a_{n-1}$, where $1 - 2^{n-1} \le X \le 2^{n-1} - 1$. Negative integers X are coded into unsigned integers: $(2^n - 1) + X = (2^n + X) - 1$. There are *two* binary representations of 0: 000 . . . and 111 . . . . *One obtains the code for $-X$ very simply by complementing each bit.* This code is used in a few arithmetic units.

(*c*) **2s-complement Code:** $X = -2^{n-1}a_0 + 2^{n-2}a_1 + 2^{n-3}a_2 + \cdots + a_{n-1}$, where $-2^{n-1} \le X \le 2^{n-1} - 1$. Negative integers X are coded into unsigned integers $2^n + X$. It has a unique 0 and simple arithmetic. *To obtain code for $-X$, complement every bit and add 1 LSB.* It is used in binary counters and in almost all minicomputers.

EXAMPLES (4-bit codes)

DECIMAL		SIGN-AND-MAGNITUDE		1s COMPLEMENT		2s COMPLEMENT	
+	7	0	1 1 1	0	1 1 1	0	1 1 1
+	6	0	1 1 0	0	1 1 0	0	1 1 0
+	3	0	0 1 1	0	0 1 1	0	0 1 1
+	2	0	0 1 0	0	0 1 0	0	0 1 0
+	1	0	0 0 1	0	0 0 1	0	0 0 1
+	0	0	0 0 0	0	0 0 0	} 0	0 0 0
−	0	1	0 0 0	1	1 1 1		
−	1	1	0 0 1	1	1 1 0	1	1 1 1
−	2	1	0 1 0	1	1 0 1	1	1 1 0
−	3	1	0 1 1	1	1 0 0	1	1 0 1
−	6	1	1 1 0	1	0 0 1	1	0 1 0
−	7	1	1 1 1	1	0 0 0	1	0 0 1
−	8	–				1	0 0 0

TABLE 1-2. Binary Codes Representing Real Fractions X by n-bit Words $(a_0, a_1, a_2, \ldots, a_{n-1})$.

Each binary digit a_k is either 0 or 1. a_0 is the most significant bit (MSB), and a_{n-1} is the least significant bit (LSB).

1. Nonnegative Fractions: $X = \frac{1}{2}a_0 + \frac{1}{2^2}a_1 + \cdots + \frac{1}{2^n}a_{n-1}$, where $0 \le X \le 1 - \frac{1}{2^n}$.

EXAMPLE (3 bits):

DECIMAL	BINARY	DECIMAL	BINARY
0.000	000	4/8 = 0.500	100
1/8 = 0.125	001	5/8 = 0.625	101
2/8 = 0.250	010	6/8 = 0.750	110
3/8 = 0.375	011	7/8 = 0.875	111

2. Signed Fractions (positive, negative, or zero). The sign bit a_0 is 0 for $X \ge 0$ and 1 for $X < 0$.

(*a*) **Sign-and-magnitude Code:** $X = (-1)^{a_0}\left(\frac{1}{2}a_1 + \frac{1}{2^2}a_2 + \cdots + \frac{1}{2^{n-1}}a_{n-1}\right)$, where $\frac{1}{2^{n-1}} - 1 \le X \le 1 - \frac{1}{2^{n-1}}$. There are *two* binary representations of 0: 000 . . . and 100 . . . . This may cause complications, e.g., in statistical work.

(*b*) **1s-complement Code:**

$$X = \left(\frac{1}{2^{n-1}} - 1\right)a_0 + \frac{1}{2^2}a_2 + \cdots + \frac{1}{2^{n-1}}a_{n-1},$$

where $\frac{1}{2^{n-1}} - 1 \le X \le 1 - \frac{1}{2^{n-1}}$. There are *two* binary representations of 0: 000 . . . and 111 . . . . *One obtains the code for $-X$ very simply by complementing each bit.* This code is used in some arithmetic units.

(*c*) **2s-complement Code:**

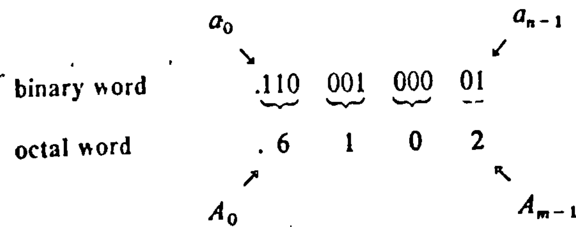
$$X = -a_0 + \frac{1}{2}a_1 + \frac{1}{2^2}a_2 + \cdots + \frac{1}{2^{n-1}}a_{n-1},$$

where $-1 \le X \le 1 - \frac{1}{2^{n-1}}$. It has a unique 0 and simple arithmetic. *To obtain code for $-X$, complement every bit and add 1 LSB.* This code is used in almost all minicomputers.

EXAMPLES (4-bit codes):

DECIMAL	SIGN-AND-MAGNITUDE		1s-COMPLEMENT		2s-COMPLEMENT	
+7/8 = +0.875	0	1 1 1	0	1 1 1	0	1 1 1
+6/8 = +0.750	0	1 1 0	0	1 1 0	0	1 1 0
+3/8 = +0.375	0	0 1 1	0	0 1 1	0	0 1 1
+2/8 = +0.250	0	0 1 0	0	0 1 0	0	0 1 0
+1/8 = +0.125	0	0 0 1	0	0 0 1	0	0 0 1
+0	0	0 0 0	0	0 0 0	} 0	0 0 0
−0	1	0 0 0	1	1 1 1		
−1/8 = −0.125	1	0 0 1	1	1 1 0	1	1 1 1
−2/8 = −0.250	1	0 1 0	1	1 0 1	1	1 1 0
−3/8 = −0.375	1	0 1 1	1	1 0 0	1	1 0 1
−6/8 = −0.750	1	1 1 0	1	0 0 1	1	0 1 0
−7/8 = −0.875	1	1 1 1	1	0 0 0	1	0 0 1
−1	—	—	—	—	1	0 0 0

We again start 3-bit groups at the implied binary point, i.e., proceeding *to the right* from a_0:



so that

$$X = \frac{1}{2}a_0 + \frac{1}{2^2}a_1 + \cdots + \frac{1}{2^n}a_{n-1}$$

$$= \frac{1}{8}A_0 + \frac{1}{8^2}A_1 + \cdots + \frac{1}{8^n}A_{m-1} \qquad 0 \leq X \leq 1 - \frac{1}{2^n}; \quad m \leq \frac{n}{3} \quad (1\text{-}3)$$

Decimal-octal-decimal conversion is defined by Eqs. (1-2) and (1-3) but is usually done with the aid of *conversion tables* (Appendix). Octal-number representations work perfectly well even if the given word length n is not divisible by 3. Note, however, that the octal-*integer* code for nonnegative pure binary fractions is identical with the octal-*fraction* code if and only if the word size is divisible by 3. For this reason, our Appendix presents an octal-fraction conversion table as well as an octal-integer conversion table.

Hexadecimal notation similarly divides each binary word into *4-bit groups* labeled with hexadecimal digits (Table 1-3):

$a_0 \rightarrow$ 1 0 | 1 0 0 1 | 1 1 1 0 | 0 1 0 0 | 0 0 1 0

2 | 9 | E | 4 | 2 (integer)

Octal-integer arithmetic, useful for "manual" work with binary operations (design, programming, see also Sec. 4-2) is easy to learn for those used to decimal arithmetic. We simply carry or borrow at 8 instead of 10 and learn a simple multiplication table. Especially for occasional use, octal numbers are probably easier to live with than hexadecimal numbers. But the latter are widely accepted in applications involving communications and on IBM 360/370 computer systems. This is because representation of the 8-bit words or partial words (bytes) used for alphanumeric characters (Sec. 1-4*d*) requires *three* octal digits but only *two* hexadecimal digits:

$$10110010_2 = 262_8 = B2_{16}$$

Conversion and arithmetic tables for both systems will be found in the Appendix [illegible]bles A-1 to A-6).

TABLE 1-3. Hexadecimal Notation.

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

(c) Parity Checking. In the course of a digital program, thousands or millions of digital words are transferred between the processor, the computer memory, and external devices. To improve reliability at the expense of some extra circuits, we can augment each n-bit word with an extra (redundant) **parity bit** which is made to equal 1 if and only if the number of 1s in the n information bits is odd. We can then check for **even parity** (even number of 1s) over all $n + 1$ bits to detect errors in 1, 3, 5, . . . bits (including the parity bit) and stop or repeat the operation as needed. Errors in 2, 4, . . . bits will remain undetected but are much less likely than 1-bit errors. Related checking methods apply to transfers of long lists of words (Sec. 3-10).

Parity checks for memory and interface transfers are recommended for critical applications, especially where a computer system is unattended. Note that simple parity checking does *not* check arithmetic or logic errors. Many end-user minicomputers operate satisfactorily without memory-transfer parity checks.

(d) Alphanumeric-character Codes. Computer input/output and digital data transmission, manipulation, and storage require binary coding of **alphanumeric-character strings** representing text, commands, numbers, and/or code groups. We use one binary word, or a byte, for each character. 4 bits ($2^4 = 16$) are enough to encode the 10 numerals 0 to 9 (*binary coding of decimal numbers*, Sec. 1-4*e*). The 26 letters of the alphabet (uppercase only) and the 10 numerals, plus some mathematical and punctuation symbols, can be squeezed into a 6-bit code ($2^6 = 64$) if resources are scarce; Table A-11 shows an example of such a code. Most minicomputer applications will employ a 7-bit code with an added 8th bit for parity checking (Sec. 1-4*c*). Table A-9 shows the ASCII code (American Standard Code for Information

Interchange), which admits uppercase and lowercase letters, numerals, standard symbols, control characters for printers and communication links (tab, line feed, form feed, rubout, end-of-message, etc.) and still has room for extra agreed-on symbols and control characters ($2^7 = 128$). A similar 8-bit code is the EBCDIC code used by the IBM Corporation.

8- and 16-bit minicomputers neatly handle one or two ASCII-character bytes in a computer word. Perforated paper tape also has eight-hole columns fitting 8-bit bytes (Fig. 3-5*a*). 12- and 18-bit machines must pack successive 8-bit characters into multiple words through rather uncomfortable packing

TABLE 1-4. Some BCD Codes.
(See Ref. 6 for special applications.)

Decimal	8, 4, 2, 1	Excess-3 (8, 4, 2, 1, code for $x + 3$)	2, 4, 2, 1
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0010
3	0011	0110	0011
4	0100	0111	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

operations (Fig. 1-17*d*); 6-bit character sets are more convenient for such machines but may not have enough characters.

(c) Binary-coded-decimal (BCD) Numbers and Other Number Codes. Table 1-4 shows some binary-coded-decimal (BCD) codes which express numerical data in terms of strings of 4-bit character codes corresponding to decimal digits. As an example, the 8, 4, 2, 1 BCD code encodes each decimal digit into the corresponding binary integer:

9	2	1	7	8	3
1001	0010	0001	0111	1000	0011

The numbers 8, 4, 2, 1 are the "weights" assigned to the binary bits defining each decimal digit. Some business-oriented computers employ BCD-coded arithmetic circuits, but this is not economical for general-purpose minicomputers (4 bits can specify 16 binary numbers, but only 10 BCD numbers). Thus, BCD circuits serve mainly in numerical displays, printers, and counters used directly by 10-fingered bipeds.

A large number of other number codes, both with and without redundant check bits, have been used. In particular, the *Gray code* (*reflected code*, Ref. 6) serves in some analog-to-digital converters (especially shaft encoders) where it is desirable to switch only 1 bit at a time during up or down counting operations.

Conversions between different coding schemes are important computer operations and are implemented both by hard-wired logic and by computer programs. Coding schemes for punched cards and for punched tapes are illustrated in Fig. 3-5.

1-5. Choice of Word Length and Data Format. (a) Word Length. Existing minicomputers are 8-bit, 12-bit, 16-bit, or 18-bit machines; we will arbitrarily eliminate 24-bit computers from the minicomputer classification. Intuitively, the number of bits quoted refers to the length of the most frequently used data word and thus to the number of bits in the main arithmetic registers. This interpretation has become somewhat blurred because software and/or microprogramming easily permits, say, an 8-bit computer to operate with composite 16-, 24-, or 32-bit words. Such an 8-bit machine may well have one or more 16-bit registers and can use single-word or multiple-word instructions. Again, modern 16-bit minicomputers can often address and fetch 8-bit half-words (bytes) as well as 16-bit words. We will speak of an n-bit computer if the *main data paths* (buses) connecting memory, processor circuits, and external devices are parallel n-bit paths (not counting extra bits used for parity checks and memory protection, Sec. 2-15). Advertising literature should be read somewhat critically in this respect.

Since computation with, say, a 4K memory can take 12 bits for addressing alone, most minicomputers with meaningful instruction sets require some double-word instructions, usually implied or disguised by indirect or relative addressing (Sec. 2-7). Depending on the application, longer word length may mean fewer double-word instructions and thus save memory and time. Clever design of short-word instruction sets is the central problem of minicomputer architecture and will be discussed in Chap. 6. We now consider the choice of *data-word* length.

In minicomputers serving largely as *logic controllers* rather than as arithmetic processors, word length need not be determined by numerical precision. Where speed is not important, any number of, say, relay closures can be controlled and/or sensed through *successive* 8-bit words. But there are also applications where an 18-bit word length (rather than 8, 12, or 16 bits) is just the thing to simplify control interface, program, and memory requirements.

In judging the data-word length to be used in fixed-point numerical computation, remember that minicomputers do not perform a true roundoff to the least significant digit. Instead, they effectively reduce the missing

digit to a zero (they "chop" or "truncate" the missing digit), so that the resulting 2s-complement number *will never be larger than the correct quantity*. It follows that even with 18-bit data words, fixed-point sums of, say, 1,000 terms, such as are frequently encountered in numerical integration or statistical averaging, *must* be computed with *double-precision* arithmetic if we actually want 18-bit accuracy; up to 10 of the least significant bits might be meaningless.[1]

(b) 8-bit Machines. 8 bits (i.e., a resolution of 1 in 256) will not permit very accurate single-precision arithmetic, although very useful logic operations (say in industrial controllers) are possible. Multiword instructions and operations, however, permit powerful 16-, 24-, and even 32-bit computations (at reduced speed) with many 8-bit machines, especially with microprogramming (Sec 6-13). Another very important application of 8-bit minicomputers is the manipulation, storage, recognition, and recoding of 8-*bit alphanumeric characters;* note that 8 bits are just right for an ASCII character with parity bit or for two BCD digits (Sec. 1-4).

(c) 12-bit Machines. 12-bit data words can accommodate the 1 in 4,000 resolution of medium-accuracy instruments (within 0.1 percent of half-scale and sign), although the results of 12-bit arithmetic will rarely have 12-bit accuracy.

As minimum-size data processors, 12-bit computers (more specifically the Digital Equipment Corporation's PDP-8 series) spearheaded the minicomputer revolution with enormous success at a time when the additional logic required for a 16-bit machine was still fairly expensive. The success of the PDP-8 has produced so much valuable software that new PDP-8-type 12-bit machines are produced not only by DEC but also by other manufacturers, with prices reduced to below $5,000 for the processor and a 4K-word memory.

(d) 16-bit Machines. Since the advent of low-cost integrated-circuit processor logic, 16-bit minicomputers have become the predominant type. Longer 16-bit instruction words permit the design of exceedingly sophisticated minicomputer architectures (Chap. 6). The second significant advantage is the ease with which two 8-bit ASCII bytes can be packed into a single 16-bit word; separate byte addressing and manipulation is possible in many 16-bit machines (Sec. 2-13).

(e) 18-bit Machines. The most successful 18-bit minicomputers have been the Digital Equipment Corporation's PDP-7/9/15 series, which have relatively simple instruction sets and employ the extra word length for direct addressing of as much as 8K of memory. Other computer designers prefer to use extra instruction bits for addressing multiple processor registers

[1] The situation is somewhat better in statistical averaging because we can subtract the expected value of the chopping error out of our result. Note, however, that the chopping-error *varianc*[illegible] adds to the variance of our statistical estimate

(Sec. 2-8). ASCII-character packing is either clumsy or wasteful with 18-bit words, but suitable packing and unpacking routines exist. Cathode-ray-tube or *xy*-recorder displays of fair resolution (512 by 512 points) can be very conveniently driven with 18-bit data words packed with 9-bit X and Y coordinate values; this arrangement halves both refresh memory and refresh time (Sec. 7-9).

(f) Data Formats. Figure 1-17 illustrates typical *data formats* used to code fixed-point binary numbers, floating-point numbers, and alphanumeric characters into 8-bit, 12-bit, 16-bit, and 18-bit words. *Instruction formats* are shown in Sec. 2-5 and in Chap. 6.

DIGITAL OPERATIONS: LOGIC AND ARITHMETIC

1-6. Logic Operations. The reasons for the explosive success of computers with binary variable representation are not only the ease of binary-data storage and transmission but also the remarkable simplicity, reliability, and

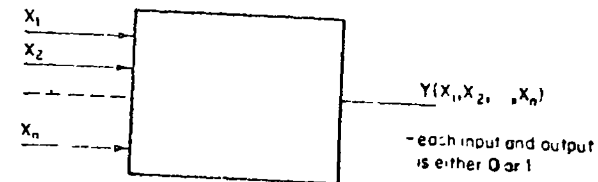


Fig. 1-5. Generation of a Boolean function Y of n inputs $X_1, X_2, \ldots, X_n$

low cost of the basic *operations* on binary variables. Figure 1-5 shows a "black box" whose output Y is a binary (Boolean) function $F(X_1, X_2, \ldots, X_n)$ of n binary input variables $X_1, X_2, \ldots, X_n$. Since each input can take only two different values, *there are 2^{2^n} different Boolean functions of n inputs*. We can characterize each Boolean function by a simple table (truth table) showing the function values for all possible combinations of argument (input) values (Fig. 1-7).

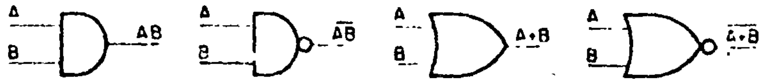
We would like to implement many different operations like that of Fig. 1-5 with electrical circuits; inputs and outputs will be voltage levels corresponding to 0 and 1 (Sec 1-1). Fortunately, *all Boolean functions can be obtained through combinations of simpler functions*. The simple one- and two-input functions of Fig. 1-6 will be more than sufficient, and all can be realized with readily available integrated circuits (logic inverters and gates).

The elementary Boolean operations of complementation (inversion), logical addition (union, ORing), and logical multiplication (intersection, ANDing) combine according to the rules of Boolean algebra listed in Table 1-5*a*. The table also illustrates how these rules are used to [illegible] in useful

TABLE 1-5a. Very Little Logic Goes a Long Way: Gate Circuits (Combinatorial Logic).

1. Basic Gates and Truth Tables. The basic *logic gates* implement simple functions of binary variables. Each gate function is defined explicitly by a *truth table* listing the gate output for all combinations of inputs.

NAND and NOR gates also serve as *logic inverters* complementing a single input (1 becomes 0, and vice versa).



AND

B \ A	0	1
0	0	0
1	0	1

NAND

B \ A	0	1
0	1	1
1	1	0

OR

B \ A	0	1
0	0	1
1	1	1

NOR

B \ A	0	1
0	1	0
1	0	0

NOTE: In some types of logic, gate outputs can be ORed together.

2. Inverters. NAND and NOR gates also serve as logic inverters for complementing a single input. We use the following inverter symbols

Some gates have *two complementary outputs*, and some logic modules provide gates with *inverting inputs*.

3. The Rules of Boolean Algebra. When we proceed to combine simple logic functions into more complicated functions of more variables, we find that the combinations satisfy the following *rules of Boolean algebra*. These rules are established by a simple combination of the basic truth tables. The rules may be applied to simplify logic circuits (logic optimization).

$A + B = B + A$, $AB = BA$ (COMMUTATIVE LAWS)

$A + (B + C) = (A + B) + C$, $A(BC) = (AB)C$ (ASSOCIATIVE LAWS)

$A(B + C) = AB + AC$, $A + BC = (A + B)(A + C)$ (DISTRIBUTIVE LAWS)

$A + A = AA = A$ (IDEMPOTENT PROPERTIES)

$A + B = B$ if and only if $AB = A$ (CONSISTENCY PROPERTY)

$A + 0 = A$ $\quad A1 = A$

$A0 = 0$ $\quad A + 1 = 1$

$A(A + B) \equiv A + AB \equiv A$ (LAWS OF ABSORPTION)

$\overline{(A + B)} \equiv \bar{A}\bar{B}$, $\overline{(AB)} \equiv \bar{A} + \bar{B}$ (DUALITY, OR DE MORGAN'S LAWS)

$\bar{\bar{A}} \equiv A$ $\quad \bar{1} = 0$ $\quad \bar{0} = 1$

$A + \bar{A}B \equiv A + B$ $\quad AB + AC + B\bar{C} \equiv AC + B\bar{C}$

TABLE 1-5a. Very Little Logic Goes a Long Way: Gate Circuits (Combinatorial Logic) (Continued)

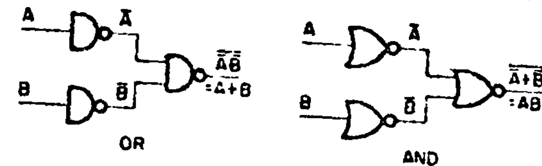
Every Boolean function is either identical to 0 or can be expressed as a unique sum of minimal polynomials (canonical minterms) $Z_1 Z_2 \cdots Z_n$, where Z_i is either X_i or $\bar{X}_i$ *(canonical form of a Boolean function)*.

In view of de Morgan's laws, *every Boolean function not identically 0 can also be expressed as a unique product of* canonical maxterms $Z_1 + Z_2 + \cdots + Z_n$, where Z_i is either X_i or $\bar{X}_i$. There are altogether 2^n minterms and 2^n maxterms.

These canonical forms show that *every* Boolean function can, in principle, be implemented with *two levels* of logic gates (either ORing of AND-gate outputs or ANDing of OR-gate outputs). But the number of gates and/or connections needed might be reduced decisively if we admit some intermediate levels at the expense of extra time delay.

4. Examples of Combinatorial Logic. *Combinatorial logic* involves only gates (including inverters), no memory or delays.

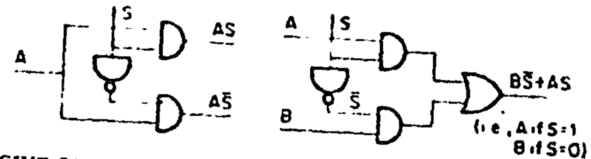
(a) NAND/NOR and NOR/AND Conversion (by de Morgan's theorem).



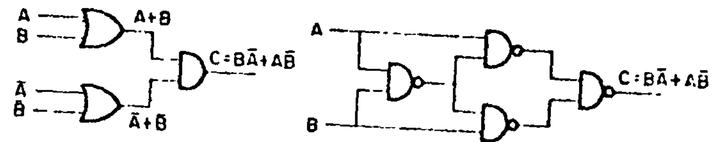
These conversion rules are useful when we have to work with specific commercially available components.

NOTE: *All* combinatorial logic *can* be implemented with NAND gates alone, or with NOR gates alone.

(b) Single-pole/Double-throw Switches.



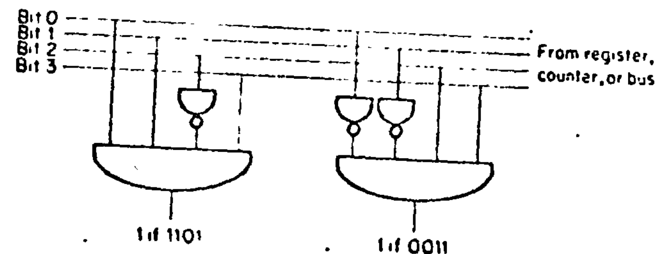
(c) EXCLUSIVE OR (XOR, Modulo-two Adder).



NOTE: $C = 0$ indicates that $A = B$ (coincidence detection).

Many other implementations exist.

(d) Recognition Gates (Decoding Gates) for selecting devices identified by a binary address code, for presetting counters, etc.



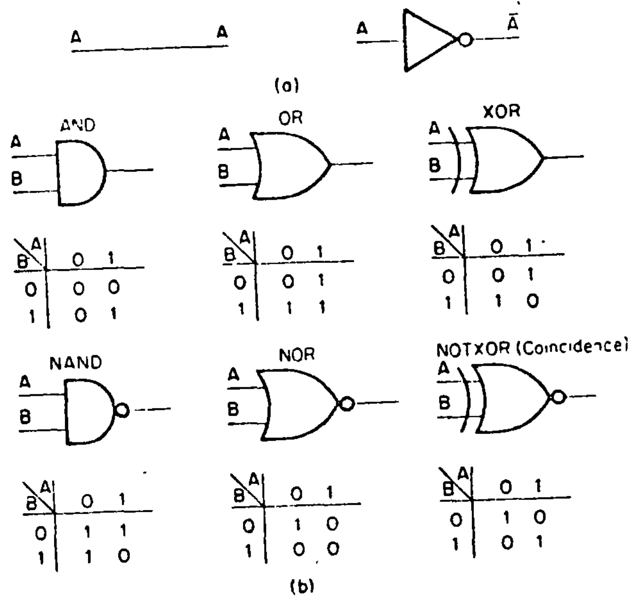


Fig. 1-6. Figure 1-6a shows Boolean functions of a *single* input. Figure 1-6b defines the most important functions of *two* inputs by simple truth tables and shows commonly used symbols for the corresponding logic gates

Boolean functions with simple gates. In particular, AND gates and inverters alone, NAND gates alone, or NOR gates alone can perform *all* Boolean operations. This is of great practical importance because some types of solid-state logic make it easier to implement NAND gates, while others lead to a preference for OR and NOR gates. Many commercially available logic systems also offer logic gates with more than two inputs, which are often convenient (Fig. 1-7).

A **flip-flop** is a 1-*bit memory device* for storing a binary variable; **flip-flop registers** are ordered sets of flip-flops for storing digital words. Table 1-5*b* defines each of the most useful flip-flop types by the method of data entry and shows two important applications (see also Secs. 1-7 and 5-3). Figure 1-8 shows how appropriately timed control pulses are used to parallel-transfer the contents of a flip-flop register to other registers.

Digital-computer *arithmetic circuits* will be designed as logic circuits operating on the bits of binary-number inputs to produce desired binary output numbers, with inputs, outputs, and intermediate results stored in flip-flop registers (Sec. 1-9).

Techniques for simplifying logic circuits (i.e., minimizing the number of gates and flip-flops, gate inputs, interconnections, and/or crossovers) form the subj of logic optimization for digital-system design (Refs. 1 to 5). Optimization of a large digital system, such as a complete computer, is

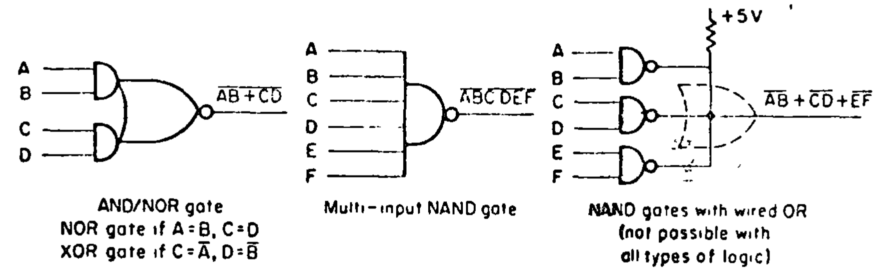
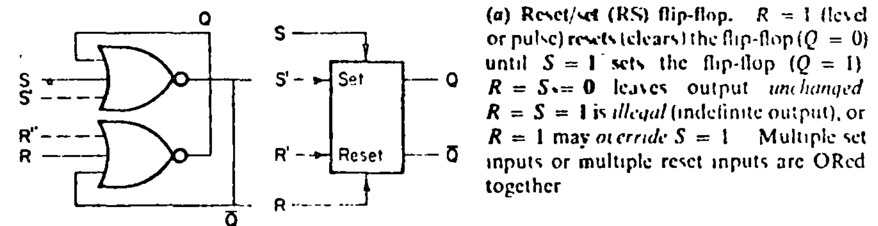


Fig. 1-7. Some multi-input gates available in integrated-circuit form. Many other types exist

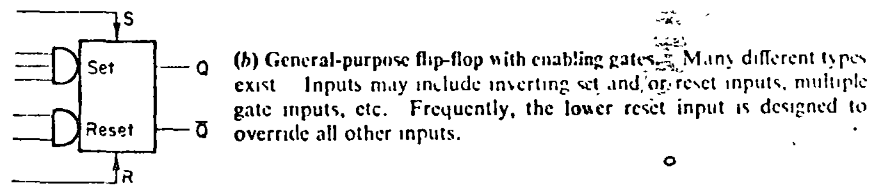
often itself done with the help of a digital computer. On the other hand, a researcher or engineer who merely wants to use a small digital computer, and to interface it to some real-world instruments and controls, will seldom require formal logic optimization. *All we usually require is the material in Table 1-5, some reasonable common sense, and a nice collection of tried logic circuits we can adapt and modify.* Manufacturers' catalogs and application

TABLE 1-5*b*. Very Little Logic Goes a Long Way: Flip-flop Circuits.

1. Flip-flops A flip-flop, like the familiar toggle switch, will stay in a given output state (0 or 1) even after inputs have been removed. Flip-flops thus implement *memory* for binary variables and permit *data storage* and *automatic sequential operations* (That is, logic states can determine the sequence of future logic states, as in data transfers, counting, etc.) Although a somewhat bewildering variety of different flip-flops are sold, all are derived from a few simple types. Specifically, the basic **reset/set (RS) flip-flop** retains its **output** state through regenerative feedback until a new reversing input is applied. Other types **of** flip-flops add different input-gating circuits

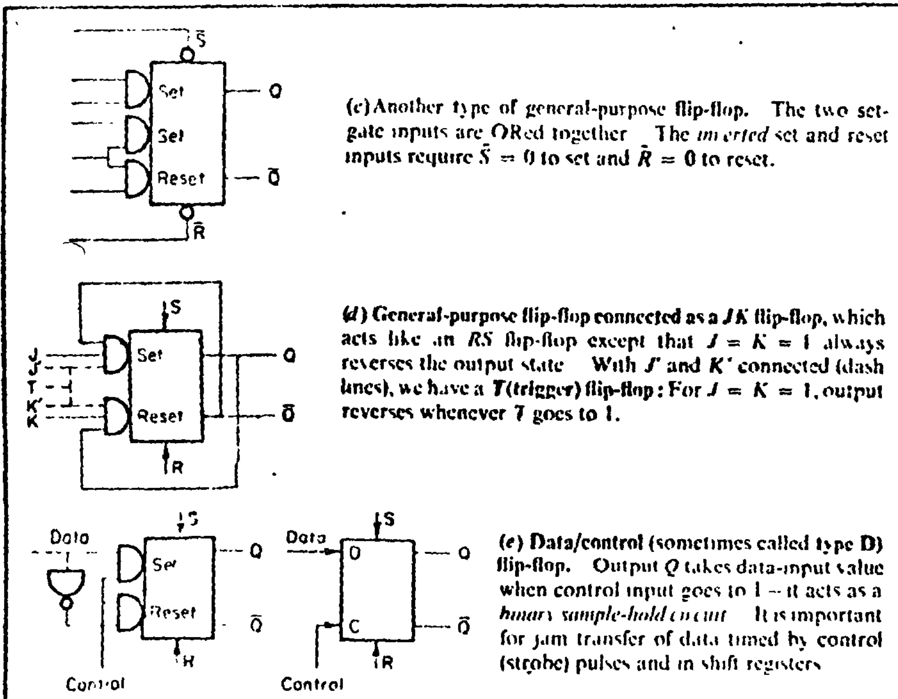


(*a*) **Reset/set (RS) flip-flop.** $R = 1$ (level or pulse) **resets** (clears) the flip-flop ($Q = 0$) until $S = 1$ sets the flip-flop ($Q = 1$). $R = S = 0$ leaves output *unchanged*. $R = S = 1$ is *illegal* (indefinite output), or $R = 1$ may *override* $S = 1$. Multiple set inputs or multiple reset inputs are ORed together



(*b*) **General-purpose flip-flop with enabling gates.** Many different types exist. Inputs may include inverting set and/or reset inputs, multiple gate inputs, etc. Frequently, the lower reset input is designed to override all other inputs.

· In some general-purpose flip-flops (diode/transistor logic, DTL), gates have *ac-coupled inputs*, which set or reset the flip-flop when a voltage *step* (either up or down, depending on the type) is gated by a logic level



(*c*) Another type of general-purpose flip-flop. The two set-gate inputs are ORed together. The *inverted* set and reset inputs require $\bar{S} = 0$ to set and $\bar{R} = 0$ to reset.

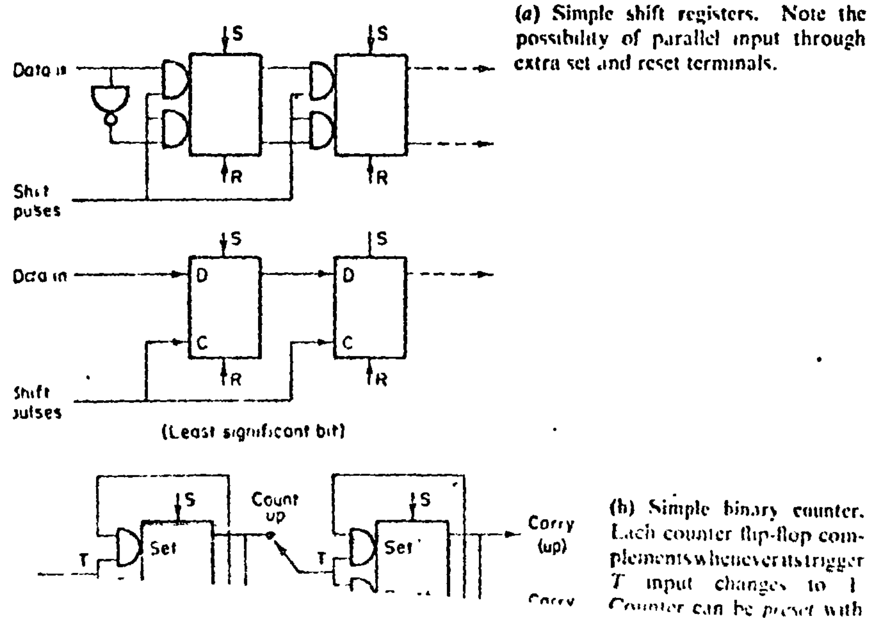
(*d*) General-purpose flip-flop connected as a *JK* flip-flop, which acts like an *RS* flip-flop except that $J = K = 1$ always reverses the output state. With J' and K' connected (dash lines), we have a ***T*** **(trigger) flip-flop:** For $J = K = 1$, output reverses whenever *T* goes to 1.

(*e*) Data/control (sometimes called **type D**) **flip-flop.** Output *Q* takes data-input value when control input goes to 1 – it acts as a *binary sample-hold circuit*. It is important for jam transfer of data timed by control (strobe) pulses and in shift registers.

Dual-rank (master-slave) type D flip-flops are designed to establish a definite time interval between input and output steps.

Consult manufacturers' logic manuals for exact logic, fanout, logic-level tolerances, noise immunity, pulse duration, and step rise time required to trigger flip-flops, etc.

2. Important Flip-flop Circuits.



(*a*) Simple shift registers. Note the possibility of parallel input through extra set and reset terminals.

(*b*) Simple binary counter. Each counter flip-flop complements whenever its trigger *T* input changes to 1. Counter can be *preset* with

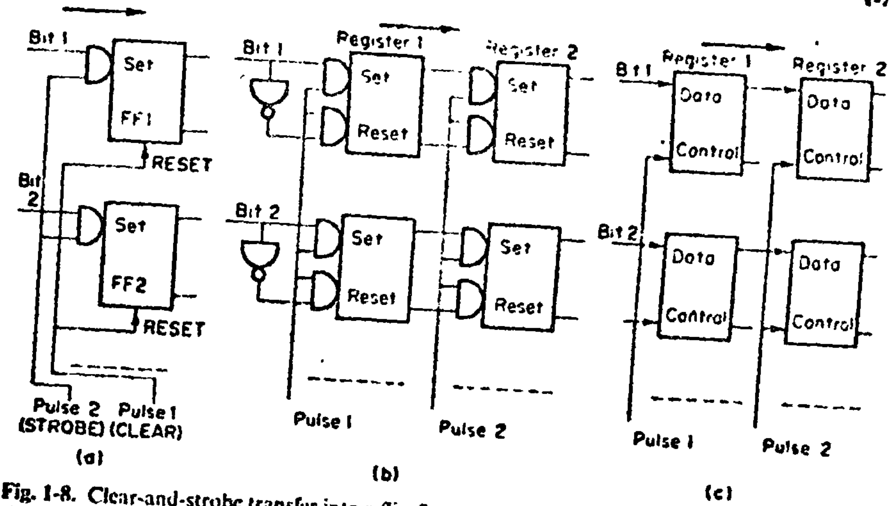


Fig. 1-8. Clear-and-strobe transfer into a flip-flop register (*a*) and jam transfer between registers (*b*) and (*c*). If the dual transfers in (*b*) or (*c*) are to take place simultaneously, we can use dual-rank flip-flops to make sure that the old output of the first register is transferred before it is updated.

notes should be consulted for special tricks and precautions applicable to specific types of commercially available logic. Digital-computer interface logic will be discussed in Chap. 5.

1-7. A General Finite-state Machine. If we agree to admit logic-state changes only at discrete clocked time intervals $0, \Delta t, 2\Delta t, \ldots$, then every sequential machine can be built from *N* type D flip-flops (Table 1-5*b*) plus combinatorial logic (e.g., AND gates, OR gates, and inverters; or NAND gates; or NOR gates), as shown in Fig. 1-9. Each flip-flop output equals its logic-level

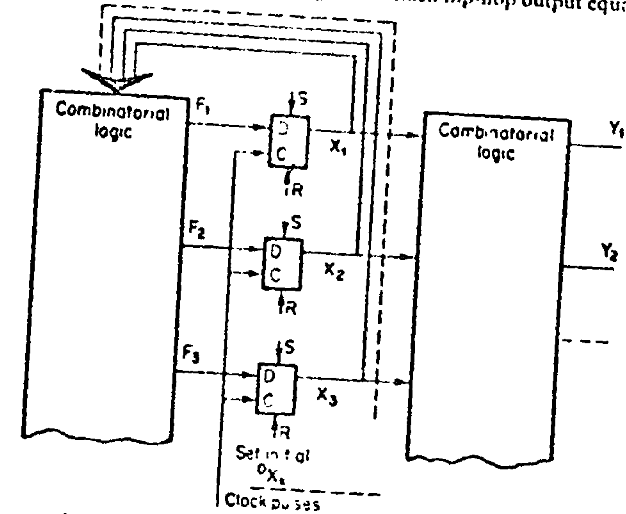


Fig. 1-9. A general clocked sequential machine. The given logic and the initial register contents ${}^0X$ determine all subsequent flip-flop states ${}^kX \equiv ({}^kX_1, {}^kX_2, \ldots)$ and outputs ${}^kY \equiv ({}^kY_1, {}^kY_2, \ldots) = Y({}^kX, k)$ through the recursion relations

$${}^{k+1}X = F({}^kX, k) \qquad k = 0, 1, 2, \ldots$$

input at the time of the last clock-pulse upswing. Thus the N flip-flop outputs

$$X_i = X_i(k\,\Delta t) \qquad i = 1, 2, \ldots, N$$

are Boolean state variables defining the state of our system during the kth clock interval. Given the N initial values $X_i(0)$, *all future states are determined by the N recursion relations (Boolean difference equations, state equations)*

$$X_i[(k+1)\Delta t] = F_i[X_1(k\,\Delta t), X_2(k\,\Delta t), \ldots, X_N(k\,\Delta t), k] \qquad i = 1, 2, \ldots, N,\ k = 0, 1, 2, \ldots$$

where each F_i is a Boolean function of the X_i. The M system outputs $Y_j(k\,\Delta t)$ are also Boolean functions of the state variables $X_i(k\,\Delta t)$ and may, like the F_i, depend explicitly on the time variable k.

In an actual digital computer, the state-determining flip-flops in Fig. 1-9 will be grouped into processor and memory registers containing numerical and control information.

1-8. Fixed-point Arithmetic and Scaling. Minicomputer data registers hold 8-bit, 12-bit, 16-bit, or 18-bit data words. It is also possible to concatenate two or more such words for **double or higher precision** (Fig. 1-17). We have seen how a binary word can represent an integer (Table 1-1) or a fraction (Table 1-2). In principle, a binary computer word $(a_0, a_1, a_2, \ldots, a_{n-1})$ could also represent a nonnegative binary number of the more general form

$$X = 2^r\left(\frac{1}{2}a_0 + \frac{1}{2^2}a_1 + \cdots + \frac{1}{2^n}a_{n-1}\right) \qquad 0 \le X \le 2^r - 2^{r-n} \qquad (1\text{-}4)$$

with a binary point implied ahead of a_r (if $r < 0$, we imply 0 digits $a_r, a_{r+1}, \ldots, a_{-1}$ between the binary point and a_0, as in $X = 0.00101$). An analogous generalization applies to signed (positive or negative) numbers. With such representations, we must keep track of the exponent r determining the *binary-point location* throughout the computation; in particular, terms in a sum or difference must have the same r. **Floating-point arithmetic programs** or circuits (Secs. 1-10 and 6-12) employ some or all the bits in an extra register to specify the exponent r and compute exponents separately at each step of the computation at considerable expense in either computing time or special hardware.

With fixed-point arithmetic, it is best to consider all numerical quantities in computer registers and memory as either integers or pure fractions (Tables 1-1 and 1-2). We propose to employ integers (which may be positive, negative, or zero) only to represent actual real integers used in counting, ordering, and addressing operations. All other real numerical quantities X in the computer will be regarded as signed or unsigned pure fractions (-1 machine unit $< X < +1$ machine unit) proportional to corresponding quantities x occurring in the given problem:

$$X = [a_x\ \ x] \qquad (1\text{-}5)$$

Each bracketed quantity $[a_x\ \ x]$ is a scaled machine variable representing

the corresponding problem variable x in the computer. *It is convenient to restrict the scale factors a_x to integral powers of 2.*

For best accuracy in fixed-point computations, we try to pick each scale factor a_x as the largest (positive, negative, or zero) integral power of 2 which will still keep the machine variable $[a_x\ \ x]$ between -1 and $+1$:

$$a_x = \frac{1}{2^r} < \frac{1}{\max|x|} \qquad (1\text{-}6)$$

Unfortunately, bounds for $\max|x|$ are not always known ahead of time, so that we may pick too small or too large scale factors. Too small scale factors waste computer precision. Too large scale factors cause overflow of the corresponding computer variables, which makes the computation **invalid.** Digital computers have flip-flops (flags) which set to indicate overflow in arithmetic operations. *These flags will not stop the computation by themselves but must be tested by programmed instructions* (Secs. 2-10 and 2-11).

To scale mathematical relations for any given problem, we simply express each problem variable x in terms of the corresponding scaled machine variable $[a_x\ \ x]$:

$$x = \frac{1}{a_x}[a_x\ \ x] \qquad (1\text{-}7)$$

Our scaling procedure is best exhibited through an example.

EXAMPLE: Scale

$$y = ax + bx^2$$

given $\qquad a = 10 \qquad b = 0.05 \qquad -7 \le x \le 19$

Since multiplication consumes more computer time than fixed-point addition, we rewrite

$$y = x(a + bx) = xz$$

We must scale the intermediate result $z = a + bx$ as well as y; substitution yields $|z| < 11 < 16$ and $|y| < 256$. Now we simply replace a, b, x, z, and y by

$$16\left[\frac{a}{16}\right], \quad \frac{1}{16}[16b], \quad 32\left[\frac{x}{32}\right], \quad 16\left[\frac{z}{16}\right], \quad \text{and} \quad 256\left[\frac{y}{256}\right]$$

where the bracketed quantities are machine variables between -1 and $+1$. We thus find the scaled machine equation

$$\left[\frac{y}{256}\right] = 2\left[\frac{x}{32}\right]\left\{\left[\frac{a}{16}\right] + \frac{1}{8}[16b]\left[\frac{x}{32}\right]\right\}$$

which is easily checked against the given problem equation through cancellation of scale factors. Note that *our computation involves only scaled machine variables and multiplying factors 2^r ($r = 0, \pm 1, \pm 2, \ldots$) corresponding to simple signed-shift operations* (Sec. 1-9b).

Our scaling procedure, as it were, keeps track of the correct exponents r in Eq. (1-4) outside of the computer.

Although fixed-point computation requires us to *program* with scaled variables, *we may not have to bother with the job of scaling reams of machine input and/or output data.* Entering and printing arabic numerals requires some computation (translation to and from binary numbers) in any case, and it is usually readily possible to incorporate scaling operations in such input/output programs (Sec. 5-27).

1-9. Some Binary-arithmetic Operations. (a) Addition, Subtraction, and Overflow. As discussed in Sec. 1-8, *we will consider all fixed-point binary numbers as signed or unsigned integers and pure fractions;* 2s-complement coding is most common.

The half-adder (modulo-2 adder) of Fig. 1-10*a* is a logic circuit for adding *one-digit* binary numbers and is seen to involve an XOR circuit. For *multidigit* addition, e.g.,

19	1 0 0 1 1
09	0 1 0 0 1
28	1 1 1 0 0

carries

each bit-by-bit addition can generate a carry bit, which must be added to the next-higher-order digit. This is accomplished by the full-adder scheme of Fig. 1-10*b*. Figure 1-10*c* shows a complete three-digit binary adder made up of three full-adders.

Such adders will produce correct results with signed numbers (2s- or 1s-complement code) if we follow these simple rules:

1. With 2s-complement arithmetic, simply add as though words represented nonnegative numbers, and disregard sign-bit carries.
2. *With 1s-complement arithmetic*, add the sign-bit carry (if any) to the least significant digit (*"end-around" carry*).

EXAMPLES.

DECIMAL (Integer)	2s-COMPLEMENT CODE	1s-COMPLEMENT CODE
6	0 | 1 1 0	0 | 1 1 0
−7	1 | 0 0 1	1 | 0 0 0
−1	1 | 1 1 1	1 | 1 1 0
6	0 | 1 1 0	0 | 1 1 0
−4	1 | 1 0 0	1 | 0 1 1
2	← 0 | 0 1 0	0 | 0 1 0
	discarded carry	"end-around" carry

In simple adders like that of Fig. 1-10*c*, low-order carries must propagate ("ripple through") all the way to the highest-order bit before the sum output is complete. To save time, one could, in principle, compute the result bit of ...der as a Boolean function of all summand bits of the same and

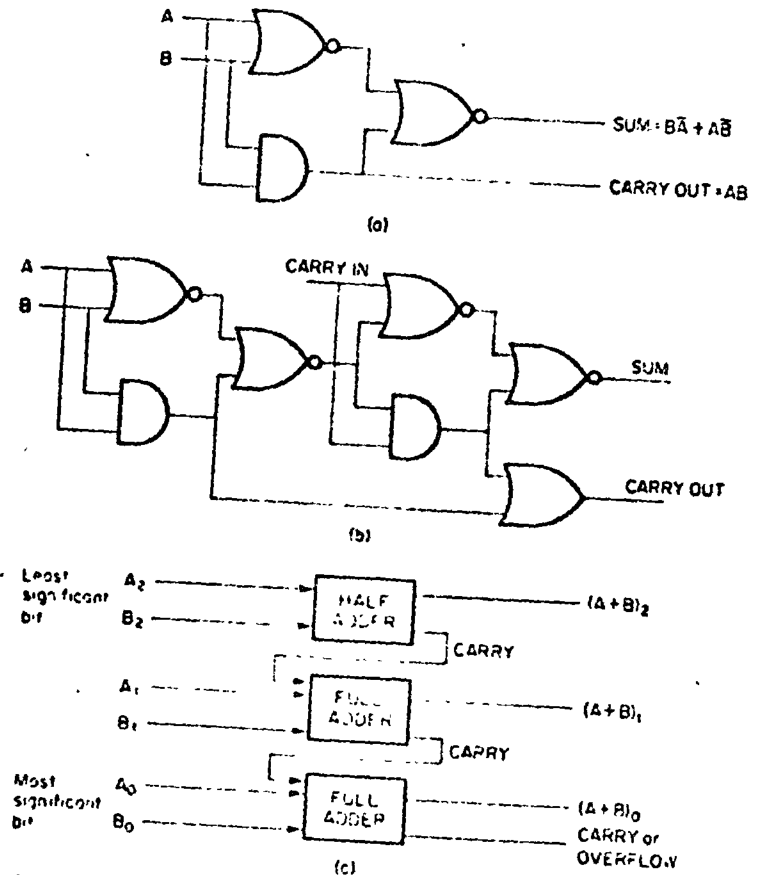


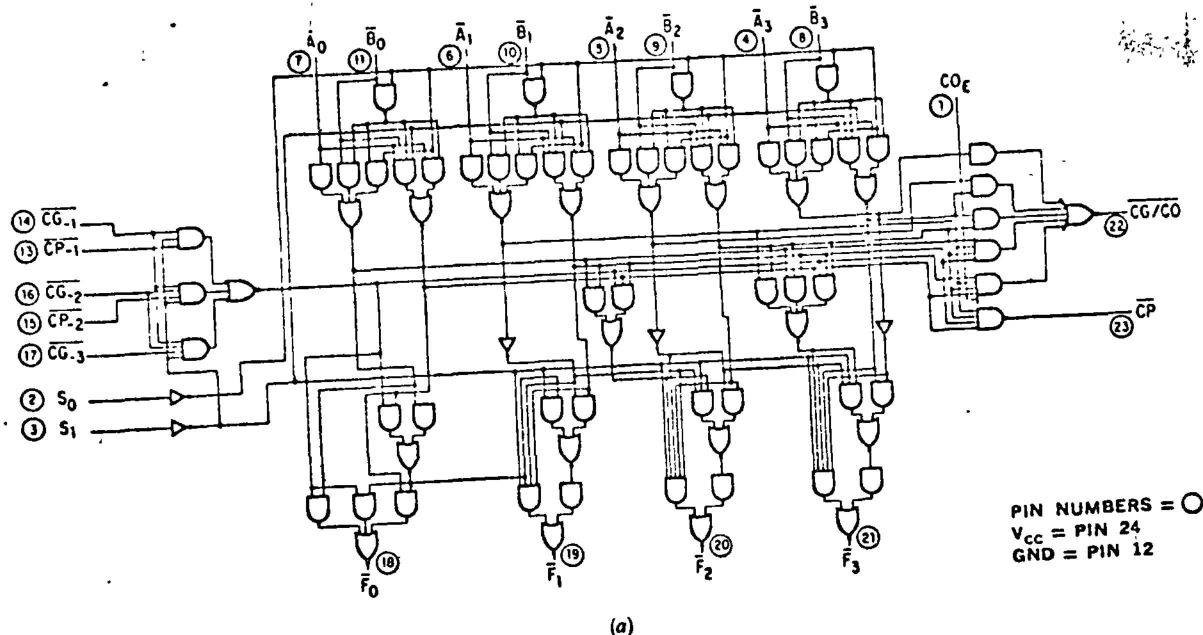
Fig. 1-10. Half-adder (*a*), full-adder (*b*), and a 3-bit adder with simple ripple-through carry propagation (*c*)

lower orders within two gate-delay times. Practical carry-lookahead circuits constitute various tradeoffs between circuit simplicity and speed (Refs. 1 and 3, and Fig. 1-11).

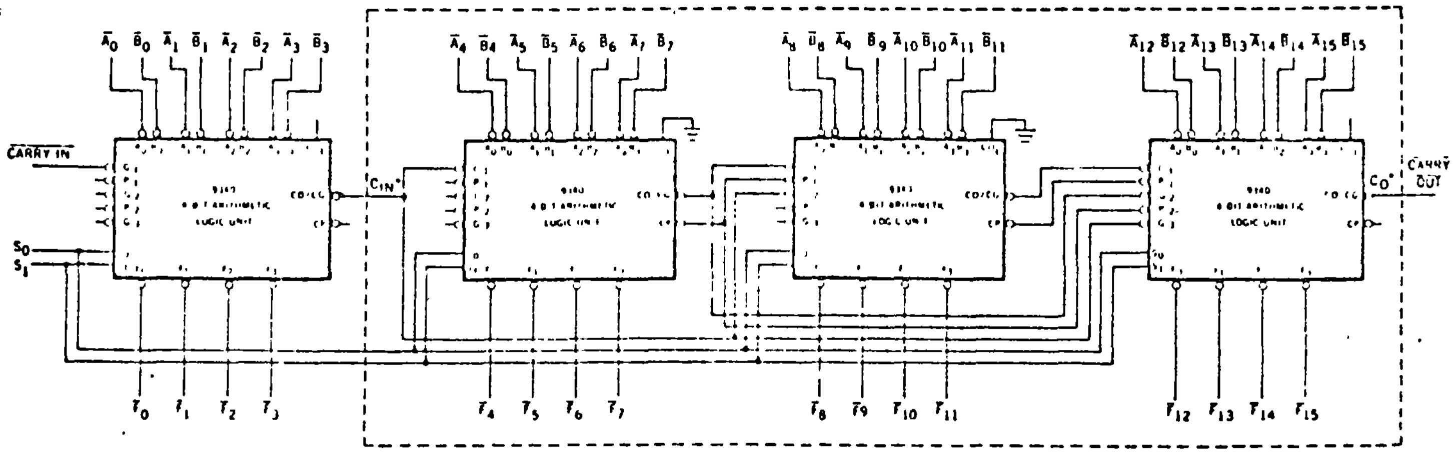
Minicomputer adders usually add a number in a processor arithmetic register (accumulator) to a number taken from memory (or from another register) and place the result into the accumulator (hence its name).

Fixed-point addition of two numbers produces arithmetic overflow if and only if:

1. Both terms of the sum have identical signs but the computed sum has a different sign.
2. Or, equivalently, addition produces a carry out of the sign bit *or* out of the most significant bit but *not both* (that is, the EXCLUSIVE OR of these carries is 1).



(a)



For 1's complement arithmetic connect $\overline{\text{Carry In}}$ to $\overline{\text{Carry Out}}$
For 2's complement arithmetic, connect $\overline{\text{Carry In}}$ to S_0

(b)

Fig. 1-11. Complete 4-bit TTL arithmetic logic unit on a single integrated-circuit chip (a) and a 16-bit minicomputer arithmetic logic unit with group carry lookahead using four such chips (b). A 12-bit arithmetic logic unit is also shown in dash lines. Bits marked $\bar{A}_i$, B_i on two input buses are combined to form output-bus bits F_i. Two function-control bits S_0, S_1 determine the function:

00	SUBTRACT	01	XOR
10	ADD	11	AND

Maximum delay is 35 nsec for 4 bits, 49 nsec for 16 bits. Shifting would be done with register-gate circuits (*Fairchild Semiconductor*.)

The same reasoning applies to subtraction if we regard differences as sums of positive and/or negative numbers. Logic circuits can test summand and sum signs and set an overflow-flag flip-flop. Many minicomputers, however, do not have a true overflow flag but only a **carry flag (accumulator-extension or link flip-flop)**, which is complemented by carries from the highest sum bit. Overflow tests for negative numbers then require several programmed instructions (Sec. 4-8c).

Binary subtraction can utilize modified adder circuits (*half-subtractors* and *full-subtractors* with *negative carries* or "*borrows*," Ref. 1), or we may negate the subtrahend (Tables 1-1 and 1-2) and add.

Figure 1-11*a* illustrates the logic design of a complete 4-bit arithmetic/logic unit, which can implement the bit-by-bit AND and XOR functions as well as addition and subtraction. The entire circuit is a single integrated-circuit chip. Figure 1-11*b* shows how such circuits combine into 12-bit and 16-bit arithmetic/logic units.

(b) Shifting (see also Sec. 2-10*b*). The definition of binary-number codes (Tables 1-1 and 1-2) implies that *shifting each digit of an unsigned* 1s*-complement or* 2s*-complement number* 1 *bit to the right will multiply the number by* $\frac{1}{2}$, *provided that the new leftmost bit equals the old sign bit or is* 0 *for unsigned numbers.* The old least significant bit is lost (chopped rather than rounded off).

Conversely, *each* 1*-bit shift to the left will multiply the original number by* 2, *provided that the new rightmost bit is made* 0 *for unsigned and* 2s*-complement numbers and equals the original sign bit for* 1s*-complement numbers.* Such multiplication by 2 will produce *overflow* if and only if the most significant bit of the given number was 1 for positive numbers and 0 for negative numbers.

EXAMPLES (4-bit 2s-complement code):

0110 represents $+6$ (or $+\frac{6}{8}$) 1010 represents -6 (or $-\frac{6}{8}$)

0011 represents $+3$ (or $+\frac{3}{8}$) 1101 represents -3 (or $-\frac{3}{8}$)

0110 and 1010 cannot be shifted left without overflow in this code (sign bit and most significant bit differ)

Digital computers employ shift operations for multiplication by integral powers of 2, and also to move partial words (bytes) in character-handling operations. Shifting could be accomplished with a shift register (Table 1-5*b*), but in most computers gate circuits like those in Fig. 1-12 move each bit of a word "sideways" during parallel register-to-register transfers.

(c) Binary Multiplication. One ordinarily computes the product of two *n*-bit binary numbers *A* and *B* as a 2*n-bit number*, so that no information is lost. This works nicely for *unsigned* integers or fractions,

$$\left.\begin{matrix} 3 \times 3 = 9 \\ \frac{3}{4} \times \frac{3}{4} = \frac{9}{16} \end{matrix}\right\} \text{ is represented by } 11 \times 11 = 1001$$

and also for *signed integers*, say in 2s-complement code:

$(-3) \times 3 = -9$ is represented by $101 \times 011 = 110111$
$(-4) \times (-4) = +16$ is represented by $100 \times 100 = 010000$

But if the *n*-bit multiplier inputs *A*, *B* and the 2*n*-bit multiplier output are interpreted as signed *fractions* (Table 1-2), *then the multiplier output is* $\frac{1}{2}AB$ (*not* AB). Thus, in 2s-complement code:

$101 \times 011 \rightarrow 110111$ represents $\frac{1}{2}(-\frac{3}{4}) \times \frac{3}{4} = -\frac{9}{32}$
$100 \times 100 \rightarrow 010000$ represents $\frac{1}{2}(-1) \times (-1) = +\frac{1}{2}$

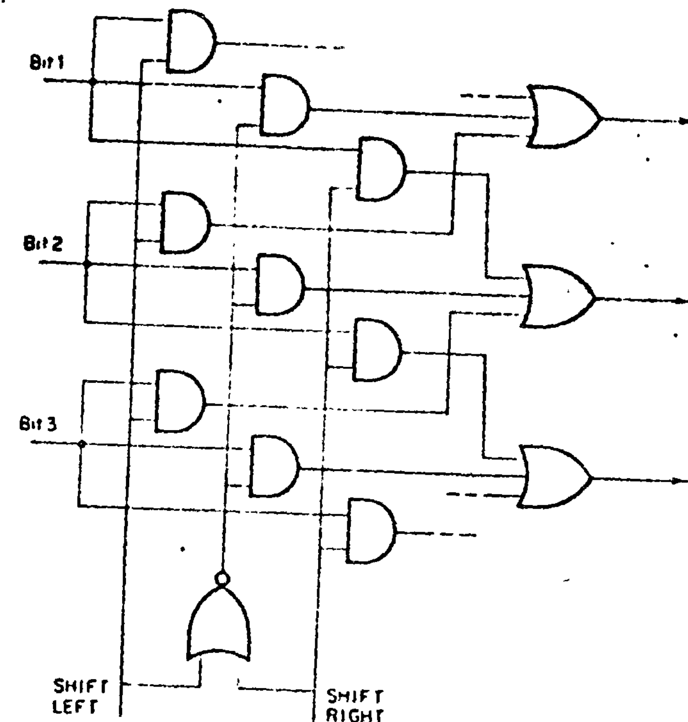


Fig. 1-12. Shifting (multiplication by 2 or $\frac{1}{2}$) with register gates (multiplexer chips)

A fast multiplier for short words can use logic or table lookup to form product digits, but this is too expensive for minicomputer arithmetic. Instead, we proceed as in pencil-and-paper multiplication. We multiply the multiplicand by each multiplier digit in turn to form *partial product*; these are then multiplied by successive powers of 2 (i.e., shifted) and added.

For simplicity, let us consider multiplication of unsigned integer. Instead of shifting partial products, the computer adds the most significa partial product into a cleared 2*n*-bit register (actually two *n*-bit registers

shifts the register contents to the left, adds the next partial product, etc. With binary numbers, each multiplier bit is either 0 or 1, so that each partial product simply adds either 0 or the given multiplicand. These operations are accomplished either through successive computer instructions (multiplication subroutine, software multiplication) or more quickly by hard-wired logic.

EXAMPLE ($3 \times 5 = 15$)

```
 011 × 101
 011
  000
   011
 001111
```

(d) Division. Division subroutines or hardware employ a *double-length dividend* and a *one-word divisor*. The result will be a *one-word quotient* plus a *one-word remainder*.

We again consider only unsigned integers or unsigned fractions. We begin by comparing the divisor with the high-order half of the dividend: the division *overflows* (and is stopped as unsuccessful) unless the divisor is larger. No quotient bit is entered at this point.

The entire two-word dividend is next shifted 1 bit to the left, and the contents of the most significant register are again compared with the divisor. If it is still larger, we enter 0 as the most significant quotient bit and shift again; if not, we enter 1 and subtract the divisor into the most significant dividend register and shift. We continue in this way (much as in pencil-and-paper division) until all quotient bits are computed. At this point, *the most significant dividend register will contain the remainder; the least significant dividend register contains the (integral part of the) quotient.*

EXAMPLE ($15 \div 7 = 2\frac{1}{7}$, 3-bit words):

```
(a) 0 0 1 1 1 1          (b) 0 1 1 1 1 0          (c) 1 1 1 1 0 0
      1 1 1                   1 1 1                    1 1 1
  no overflow, shift          shift                   subtract and shift

              Quotient  0 1 0     Remainder  0 0 1
```

For division of *signed numbers*, consult your computer manual as to the specific format used. Most frequently, the correctly signed quotient is left in the least significant dividend register, while the remainder, again in the most significant dividend register, is an unsigned number preceded by the sign of the *dividend*.

To compute *scaled-fraction quotients* X/Y of N-bit scaled fractions X, Y (Sec. 1-8) or, for that matter, of N-bit integers X, Y with an N-bit computer, simply place X into the *most significant* dividend register and clear the least significant dividend register. The desired signed or unsigned fractional quotient will be correctly produced in the form $2^{-N}(2^N X/Y)$

1-10. Floating-point Arithmetic. (a) Floating-point Data Representation. Binary floating-point arithmetic represents each real number X by *two* binary numbers, an N-bit signed binary fraction (mantissa) A and an M-bit signed binary integer (exponent) R so that

$$X = A \times 2^R \tag{1-8}$$

The mantissa is usually represented in sign-and-magnitude code (Table 1-2). The exponent can be a 2s-complement integer (Table 1-1); some floating-point number representations do not use the exponent R itself but employ, instead, a nonnegative **biased exponent** or **characteristic**

$$R' = R + B \tag{1-9}$$

where B is an agreed-on positive integer. Typical minicomputer floating-point formats are shown in Fig 1-17.

Floating-point number representation is not unique since, for instance,

$$0.10100_2 \times 2^9 = 0.01010_2 \times 2^{10} = 0.00101_2 \times 2^{11}$$

The first form, where the most significant binary digit of the (nonnegative) mantissa is a 1, is often defined as the **normalized** form of the floating-point number. A number is also considered normalized if the exponent is as small as possible and the absolute value of the mantissa is still less than $\frac{1}{2}$.

Some computer manufacturers (e.g., IBM, Interdata, Data General) use a *hexadecimal* floating-point representation defined by

$$X = A \times 16^R \tag{1-10}$$

where A is a binary fraction and R is a binary integer (usually expressed in hexadecimal code, Sec. 1-4*b*). In this case, a properly normalized mantissa will have at least the magnitude $\frac{1}{16}$; i.e., at least one of the four most significant bits of the positive fraction is a 1.

Fortunately, most minicomputer users meet *binary* floating-point formats only when troubleshooting. Input/output is almost always in *decimal* floating-point format, usually in the E format familiar to FORTRAN users:

$$0.2734\ \mathrm{E} + 02 = 0.2734_{10} \times 10^2$$

The floating-point representation [Eq. (1-8)] covers the range

$$-2^{2^{M-1}-1} < X < 2^{2^{M-1}-1} \tag{1-11}$$

This range is usually so very large ($2^{2^{M-1}-1} > 10^{38}$ for $M = 8$, see also Fig. 1-17) that *no scaling is necessary with most practical problems*. This truly dramatic advantage over fixed-point computation is paid for with either extra computing time or more expensive hardware, and usually also with *reduced precision* per bit used to represent X (M b[illegible] are used for the exponent, which does not contribute "significant dig[illegible]). Roundoff

errors can be multiplied by large factors 2^R in some parts of a computation, so that a multidigit floating-point result may be less accurate than it looks. In particular, *minicomputer two-word floating-point formats have uncomfortably short mantissas*. Unless you know exactly what you are doing, we recommend *three- or four-word formats* with 16- and 18-bit machines (Fig. 1-17). This applies especially where many terms are added, as in numerical integration and averaging.

Overflow of the floating-point range [Eq. (1-11)] is possible, but rare. Underflow (normalized exponent R below $1 - 2^{M-1}$) will return $X = 0$ in most systems.

(b) Floating-point Operations. *Floating-point addition and subtraction* require the computer to perform the following—fairly involved—operations:

Compare exponents
Shift mantissa of smaller term so that both terms have equal exponents
Add (or subtract) mantissas
Normalize result; check for overflow, return 0 on underflow

With floating-point arithmetic, *multiplication and division* are rather simpler than addition and subtraction:

Enter with normalized data; multiply (or divide) mantissas—exit if 0
Add (or subtract) exponents; check for overflow, return 0 on underflow
Normalize result

All these operations must be implemented with software (subroutines), with a microprogram (Sec. 6-13), or with optional hardware (floating-point arithmetic unit). *We must also provide for the additional operations of "floating" fixed-point numbers and "fixing" floating-point numbers, and for decimal input/output*. Suitable assembly-language subroutines will be found in computer manufacturers' software listings; floating-point hardware is discussed in Refs. 1 and 6.

MEMORY AND COMPUTATION

1-11. Introduction. A computer memory is needed to store data and instruction sequences; in addition, a finite instruction set makes it necessary to compute and store intermediate results. In effect, each computer memory consists of a large number of binary storage registers (memory locations) each capable of storing a complete computer word, plus circuits to address a program-selected memory location for reading or writing a word.

To access a memory location, we place its number (memory address) in the memory address register. A "tree" of decoding gates (Table 1-5*a*) connected to the memory address register will then direct logic signals to read the [illegible] memory data register (memory buffer

register) or write a word *from* the memory data register into the selected memory location. The access time is the time needed to select and read. The write time required to select and write is often called cycle time, a memory cycle being the time required to read/erase *and* write/rewrite in a magnetic-core memory (Sec. 1-12).

The main memory of a minicomputer usually has between 1,000 and 32,768 words of magnetic-core, semiconductor, or plated-wire storage with effective cycle times between 250 and 8,000 nsec (Secs. 1-12 and 1-13). As a compromise, inexpensive 800- to 3,000-nsec main memories can be supplemented by very fast (50- to 200-nsec) *intermediate storage* (scratchpad memories) in the form of flip-flop-register or semiconductor memories. In addition, we often add slow but inexpensive *mass storage* in the form of magnetic disks, drums, and tape (Chap. 3). These can store large programs and data blocks (up to millions of words) which are (one hopes) not immediately needed at all times, but which can be transferred to and from the main memory as need arises. We thus have a *hierarchy of storage systems*.

1-12. Core Memories. Most minicomputer main memories are core memories, which store individual bits by magnetizing toroidal ferrite cores in the 1 or 0 direction through a write-current pulse (Fig. 1-13*a*). To read the information stored in such a core, one pulses the core in the 0 magnetization direction; if a 1 had been stored in the core, the resulting flux reversal would cause an output current pulse in a *sense wire* threaded through the core (Fig. 1-13*b*). To *select* only the cores associated with a specific word in the memory, we implement the read and write currents through superposition of select and inhibit currents in two or three wires threading each core (coincident-current selection, Figs. 1-13*a* and *b*). Figure 1-13*c* illustrates a typical core-memory word selection scheme (*3D scheme*), and Fig. 1-13*d* shows the wiring of a typical bit plane; the sense wire is threaded through all cores in the plane in a pattern designed to cancel the effects of the half-select current pulses associated with unselected cores. At the expense of a little extra switching logic, *the same bit-plane wire can be used for both inhibiting and sensing*, so that only *three* wires need to be threaded through each core. Reference 3 describes two different word/bit arrangements for core memories (2D and 2½D).

Core storage is nonvolatile; i.e., the memory continues to store its contents even when computer power is off. Full-cycle times of typical minicomputer core memories are between 650 nsec and 8 μsec, half of which is the access time. As shown in Fig. 1-13*b*, core-memory readout is destructive; i.e., *reading clears the addressed memory location*; ordinarily, the word thus read into the memory data register is rewritten into core during the second half of the READ memory cycle. The cycle time of even a 650-nsec core memory is quite long compared to the 50- to 100-nsec clock-interval times

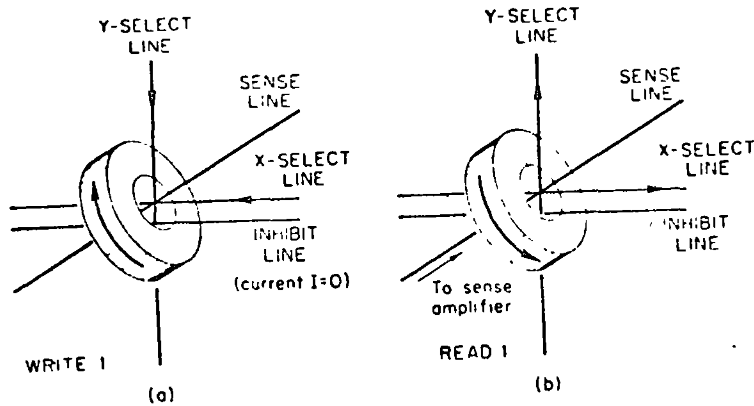


Fig. 1-13a and b. Coincident-current writing and reading/erasing in a typical minicomputer core memory. (a) WRITING. We start with all cores magnetized in the 0 direction. Writing a 1 into a given core (i.e., a given bit of a selected word) depends on *three* current pulses. Each of the two *select currents* X, Y must be one-half of the required magnetizing current (word selection) *and* the *inhibit current i* (which would oppose 1 magnetization) must be 0 (bit setting). The inhibit line, common to all *words*, belongs to a specific memory-data-register *bit*. (b) READING/ERASING. The X and Y select wires are both pulsed with current in the 0 magnetization direction. This produces no change if the core is already magnetized in the 0 direction. If a 1 was stored, it will be erased, and the flux reversal will cause a 1 pulse in the sense line. The latter, common to all *words*, sets a specific memory-data-register *bit* via a sense amplifier.

of integrated-circuit arithmetic units. In larger digital machines with multiple core-memory banks, one can partially circumvent core-access delays by taking successive memory words from *different* memory banks; this permits rewriting in one bank to be *overlapped* with logic operations and memory accesses to other banks. Such memory-bank overlapping is rarely used with minicomputers, which may have only a single memory bank altogether, but *the possibility should not be overlooked* (Sec. 6-9).

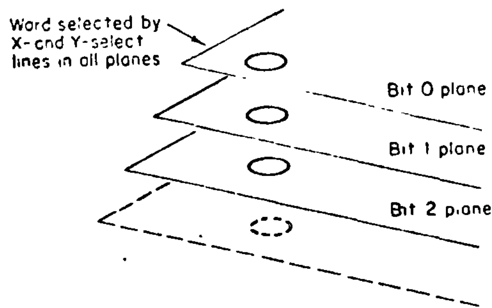


Fig. 1-13c. ... cores belonging to a given *word* have the same *X*, *Y* position in each bit plane. ... one memory-data-register *bit*

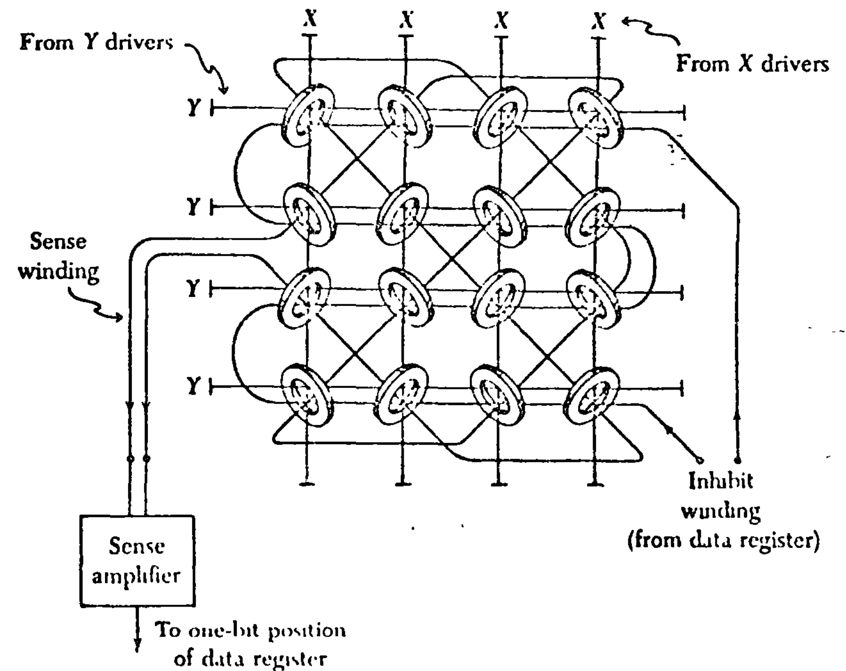


Fig. 1-13d. One bit plane of a 16-word 3D core-memory unit (*H. Hellerman, Digital Computer System Principles, McGraw-Hill, New York, 1967*)

Core-magnetization retention at temperatures much above 85°F requires either high-temperature core material or an automatic increase of core-driving currents with temperature. Typical minicomputer core memories can work at ambient temperatures up to 110 to 130°F; *you should check this specification carefully against your application requirements.*

With new monolithic sense and driver amplifiers and clever core-stringing techniques, core-memory manufacturers are still holding their own against the onslaught of new solid-state memories. Typical 16-bit 1-μsec minicomputer memories, including selection, driving, and sensing electronics, cost on the order of 2 to 5 cents/bit in 1K banks.

1-13. Semiconductor and Plated-wire Memories. With the advent of medium-scale and large-scale integrated circuits permitting fair manufacturing yields, all-electronic solid-state memories have become a reality. Semiconductor memories are smaller than equivalent core memories, they can be faster and consume less power; and, in the long run, semiconductor memories will probably be substantially cheaper (Ref. 23).

Bipolar-transistor static memories, which are essentially multiple flip-flop registers plus read/write/selection circuits, are directly compatible with

transistor/transistor or emitter-coupled logic and are very fast: read and write times below 50 nsec are readily possible. No rewriting after reading is needed (nondestructive readout, NDRO). Bipolar memories are fairly complex integrated circuits and are still expensive (of the order of 20 cents/bit in 1K banks). They are, therefore, used mostly in small "scratchpad" memories. Prices are expected to decrease to below 2 cents/bit as integrated-circuit yields improve.

MOSFET (metal-oxide-silicon field-effect transistor) semiconductor memories involve simpler integrated-circuit patterns and are cheaper than bipolar memories. While older MOSFET circuits needed level-changing amplifiers to supply large logic-voltage swings, some newer MOSFET memories are TTL-compatible. MOSFET memories also come as static (flip-flop-register) memories but usually as dynamic memories. In a dynamic memory, each bit is stored in what amounts to a shift register whose output is fed back to the input through a clock-gated MOSFET *refresh amplifier*, so each stored bit is recirculated and regenerated, say, 1,000 times/sec (Fig. 1-14). The refresh amplifier can be time-shared among 16 to 32 memory cells. Simple silicon-substrate MOSFET memories are slower than bipolar memories (and slower than some core memories). Typical access times are between 300 nsec and 2 μsec with nondestructive readout. 1-μsec dynamic MOSFET memories cost about 1 to 3 cents/bit in quantities of 1,000, and prices can be expected to decrease to below 1 cent/bit.

The era of solid-state computer memories is still in its beginning. One may anticipate massive developments, both with respect to better yields (and thus much lower memory costs) and in the development of new integrated memory circuits. In particular, different types of MOSFET circuits (complementary MOSFETs, sapphire and garnet substrates) are under active development and can be expected to lead to substantially faster MOSFET memories. Compared to core memories, semiconductor memories have the advantage of *nondestructive readout*. On the other hand, semiconductor memories are volatile; i.e., memory contents are destroyed when computer power is turned off. In sufficiently critical applications, one must provide an emergency power source, such as a trickle-charged battery which, when a power failure is sensed, can take over memory operation for a time sufficient to transfer the entire contents of the memory onto an auxiliary magnetic storage medium (disk or tape).

Plated-wire memories are magnetic memories which utilize small zones of magnetizable thin films plated onto wires, rather than magnetic cores, for bit storage. Plated-wire memories permit fast access (access times as low as a few hundred nanoseconds) with nondestructive readout, are nonvolatile, and have been the subject of considerable hopes and expectations. In fact, excellent plated-wire memories are commercially available. But, although batch-production methods have been developed, quality control is not

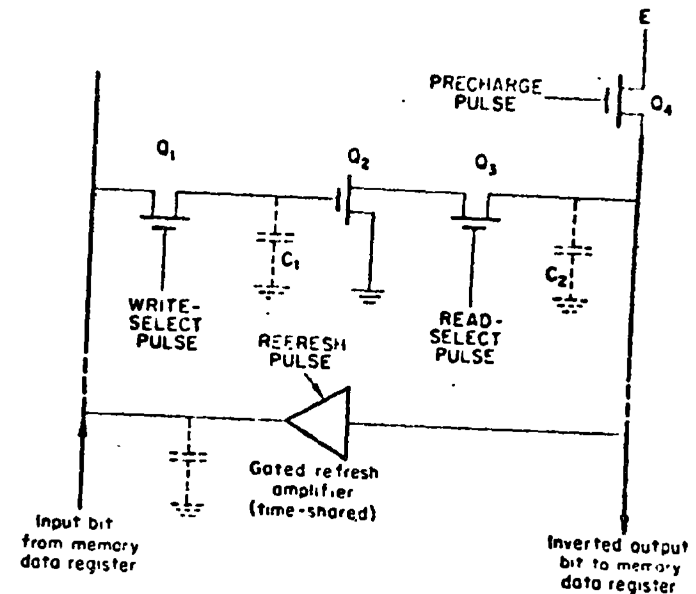


Fig. 1-14. A dynamic MOSFET memory. Q_1, Q_2, and Q_3 form a 1-bit memory cell. PRECHARGE, WRITE-SELECT, READ-SELECT, and REFRESH pulses are associated with consecutive phases of a four-phase clock. Each pulse, if enabled by the cell-selecting logic, will transfer a bit from one parasitic capacitance to the next in a clockwise direction. PRECHARGE charges C_2 by turning Q_4 ON and OFF. WRITE-SELECT turns Q_1 ON and OFF; if a 1 is written, C_1 is charged. READ-SELECT turns Q_3 ON and OFF; if a 1 is stored, Q_2 turns ON and discharges C_2 so that the (inverted) output is 0. The gate refresh amplifier (essentially another similar memory cell) inverts the selected output bit and transfers it back to its cell input once every millisecond or so.

simple. As a result, plated-wire memories are not cheap (5 to 10 cents/bit) and have been applied mostly in higher-priced digital computers (especially in aerospace-vehicle computers); MOSFET memories seem to have overtaken plated-wire circuits in the low-cost minicomputer field. This situation may or may not be changed by future improvements in plated-wire-memory fabrication.

1-14. Read-only Memories. Read-only memories (ROMs), whose contents are usually checked out and written once and for all at the time of manufacture and then cannot be overwritten, are used to store frequently reused program sequences or bit patterns:

1. *Complete special-purpose programs*, especially in industrial logic-sequence controllers replacing old-fashioned relay-ladder logic
2. *Important library subroutines* for special arithmetic sequences, control or emergency routines, scale or format transformations, etc.

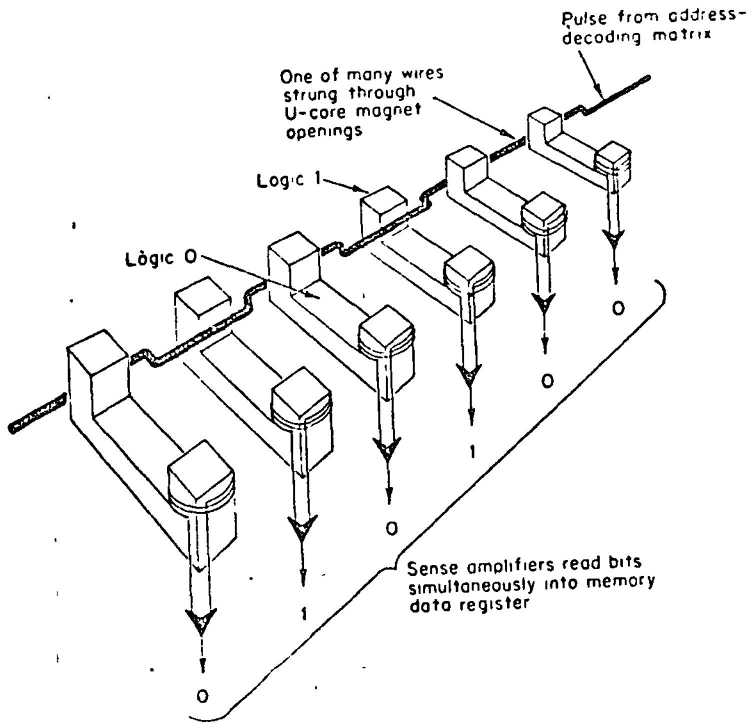


Fig. 1-15. Magnetic read-only memory (ROM). Each U core may be given a top to improve its magnetic circuit after the word wires have been strung

3. *System programs* such as bootstrap loaders (Sec. 3-4*b*), input/output subroutines (Sec. 5-30), and even simple compilers
4. Special *directories* and *function tables*
5. *Microprograms* (firmware) for implementing or emulating special instructions or instruction sequences (Sec. 6-13)

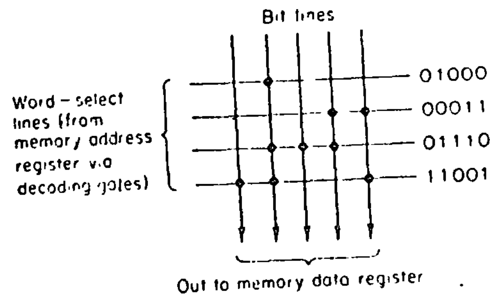


Fig. 1-16. Principle of a crossbar-matrix read-only memory. Each crosspoint connection is made thr· ·h a diode or MOSFET gate

(a) Fixed-point Formats

Single precision (8, 12, 16, or 18 bits)
0,1 — 7, 11, 15, or 17
Sign — MSB — LSB

Double precision (16-bit words shown)
0,1 — 15, 0 — 15
Sign — MSB — LSB

(b) Floating-point Formats Two 16-Bit Words
(All yield about 6 decimal places, other formats are possible)

Honeywell
0,1 — 8,9 — 15, 0 — 15
Mantissa sign — MSB — MSB
Exponent + 128 (8 bits) — 23-bit fraction

Hewlett-Packard
15 14 — 0, 15 — 8, 7 — 1 0
Mantissa sign — MSB — MSB — Exponent sign
23-bit fraction — Signed exponent (7 bits and sign, 2s complement)

IBM, Interdata, Data General (can be 4 8-bit bytes)
0,1 — 7,8 — 15, 0 — 7,8 — 15
Mantissa sign — MSB — MSB
Hexadecimal exponent + 64 (7 bits) — 24-bit fraction

(c) Digital Equipment Corporation Three-word Floating-point Formats
(12-bit PDP-8 series, 18-bit PDP-15)

0,1 — 11 or 17, 0,1 — 11 or 17, 0 — 11 or 17
Sign — MSB — Sign — MSB — LSB
Signed exponent (2s complement) — Signed mantissa (sign-and-magnitude)

Note: use of an entire word for the exponent is wasteful but simplifies software floating-point arithmetic.

With 12-bit words, 23-bit mantissa fraction yields 6 decimal places
With 16-bit words, 31-bit mantissa fraction yields 8 decimal places
With 18-bit words, 35-bit mantissa fraction yields 9 decimal places

(d) Multiple-precision Floating-point Formats
Similar to (b), with extra words added for longer mantissa fractions

(e) Packing of 8-bit ASCII Characters

16-bit words
0 — 7,8 — 15
character 1 — character 2

12-bit words
0 — 7,8 — 11, 0 — 3,4 — 11
character 1 — character 2 — character 3

Fig. 1-17. Typical minicomputer data formats

6. *Bit-pattern generators* such as *character generators* for displays and *test-signal generators* (see also Sec. 7-10)

Users may value the ROM's reliability and assurance against accidental overwriting of important stored information. Another potential advantage is that nondestructive instruction readout from a ROM can be partially overlapped with instruction execution since no rewriting is necessary (Sec. 6-5). On the debit side, simple read-only memories do not make allowances for programming afterthoughts. Stored programs or data must be checked out very carefully through preliminary runs with an ordinary memory.

An important application of read-only memory is to logic design, for the ROM can produce complicated multi-input Boolean functions (Sec. 1-6) through simple table lookup. Such table-lookup functions can be used in the sequential-machine setup of Fig. 1-9 to generate complex sequential patterns (Ref. 22).

Figure 1-15 shows a **magnetic ("woven-wire") ROM.** Read-select logic pulses one of many word-wires strung inside and outside an n-bit array of U cores and causes an output pulse in those U-core windings wired for a 1. We see that memory contents are established by each word-wire stringing pattern. Access times of such magnetic ROMs vary between 300 and 2,000 nsec.

Semiconductor ROMs can use the dynamic-storage technique (Sec. 1-13), but most semiconductor ROMs are essentially crossbar matrices (Fig. 1-16), whose crosspoint connections are established by diode or MOSFET OR-gate connections. The storage pattern is established through selective erasure of crosspoint connections or semiconductors during manufacture or during installation ("field-programmable" ROMs). Typically, access times vary between 50 nsec and 1 μsec, and costs are decreasing from 2 cents/bit. Some semiconductor ROMs can be reprogrammed in the field through new metallic connections or even through electrical signals.

REFERENCES AND BIBLIOGRAPHY

1. Chu, Y.: *Digital-computer Design Fundamentals*, McGraw-Hill, New York, 1962.
2. ———: *Introduction to Computer Organization*, Prentice-Hall, Englewood Cliffs, N.J., 1970.
3. Hellerman, H.: *Digital-computer-system Principles*, McGraw-Hill, New York, 1967.
4. Baron, R. C., and A. T. Piccirilli: *Digital Logic and Computer Operations*, McGraw-Hill, New York, 1968.
5. Hill, F. J., and G. R. Peterson: *Introduction to Switching Theory and Logic Design*, Wiley, New York, 1968.
6. Huskey, H. D., and G. A. Korn: *Computer Handbook*, McGraw-Hill, New York, 1962.
7. Klerer, M., and G. A. Korn: *Digital Computer User's Handbook*, McGraw-Hill, New York, 1967.
8. Korn, G. A.: *Random-process Simulation and Measurements*, McGraw-Hill, New York, 1966.
9. ——— and T. M. Korn: *Mathematical Handbook for Scientists and Engineers*, 2d ed., McGraw-Hill, New York, 1968.

10. Korn, G. A., and T. M. Korn: *Electronic Analog and Hybrid Computers*, 2d ed., McGraw-Hill, New York, 1971.
11. *An Introduction to HP Computers*, Hewlett-Packard Corporation, Cupertino, Calif., 1970.
12. *Small Computer Handbook*, Digital Equipment Corporation, Maynard, Mass. (issued yearly).
13. Special Issue on Minicomputers, *IEEE Comput. Group News*, July–August 1970.
    Minicomputers: An Introduction, by T. Storer
    Minicomputer Architecture, by W. H. Roberts
    Minicomputer I/O and Peripherals, by E. Holland
    Small-computer Software, by H. W. Spencer et al.
14. Epstein, A., and D. Bessel: Minicomputers Are Made of This, *Comput. Decis.*, August 1970.
15. Federoff, A. M.: Minicomputers Are Used for This, *Comput. Decis.*, August 1970.
16. French, M.: Survey of Small Digital Computers, *EEE*, February 1970.

Semiconductor Memories

17. Semiconductor Memories Turn the Corner, *Electron. Des. News*, February 1970.
18. Selecting and Applying Semiconductor Memories, *EEE*, June, July, and August 1970.
19. Boysel, L., et al.: Random-access MOS Memory, *Electronics*, February 1970.
20. Boyle, A. J.: MOS Course: Read-only Memories, *Electr. Eng.*, July and September 1970.
21. Hoff, M. E.: MOS Memory and Its Applications, *Comput. Des.*, June 1970.
22. Kvamme, F.: Standard Read-only Memories Simplify Complex Logic Design, *Electronics*, January 1970.
23. Semiconductor Memory Survey, *Electronics*, August 23, 1972.

CHAPTER 2

"BASIC" MINICOMPUTERS AND INSTRUCTION SETS

INTRODUCTION AND SURVEY

This chapter outlines the design and operation of a "basic" minicomputer illustrating the common features of many small machines. The principal ingredients of such a system—memory, registers, buses, and arithmetic/logic unit—are introduced in Secs. 2-1 to 2-5. Sections 2-6 to 2-14 then list the most important machine instructions in the repertoire of a "basic" single-address minicomputer, discuss their implementation in terms of the system block diagram, and mention the most important applications of some instructions. Sections 2-13 to 2-15 describe useful options available with small digital computers. In Chap. 4, we will meet the basic computer instructions again; specifically, we will then see how they can be combined into practical assembly-language programs. Input/output will be further discussed in Chaps. 3 and 5. More advanced instruction sets and architectures for a new generation of minicomputers will be described in Chap. 6.

THE BASIC SINGLE-ADDRESS MACHINE

2-1. Instruction Sets and Stored Programs. From a very general point of view, *the essential objective of any digital computation is to obtain digital output words*

$$Y1 = F_1(X1, X2, \ldots)$$
$$Y2 = F_2(X1, X2, \ldots) \qquad (2\text{-}1)$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$

from input words $X1, X2, \ldots$. Both input and output words will be ordered sets of 0s and 1s in suitably addressed computer registers and/or memory cells. Words may represent various types of numbers and alphanumeric-character strings or simply describe problem-logic states.

The desired relationships [Eq. (2-1)] may be numerous and enormously complicated. They must be broken down into elementary mathematical relations implemented by a (we hope small) set of computer instructions. It will, then, be necessary to supply additional registers or memory locations for storing intermediate results from elementary operations. The basic digital computer is, moreover, designed to perform all the various elementary arithmetic/logic operations *successively* with *the same* arithmetic/logic system (unlike, for instance, a conventional analog computer, which has separate adders, multipliers, etc., for separate operations). The resulting sequence of elementary instructions designed to implement a desired computation is called a **program.**

Short, simple, or repetitive digital programs can be implemented by hard-wired controllers (e.g., rotating-cam operation of control switches, read-only memories), by patch-cord systems, or by punched-tape or punched-card readers. Practical digital-computer programs, however, often require extremely large numbers (thousands and even millions) of instructions. It is also often desirable to change a digital-computer program very quickly and even to modify programs while they are being executed.

These considerations make it expedient to code the instructions themselves **into digital-computer words which, like the data words, are stored in sequences in the computer memory.** Operation of the resulting **stored-program digital computer** will now involve **alternate reading (fetching) of instruction words and execution of the corresponding instructions.**

The machine will most often read instructions in sequence, but is capable of *branching* to a different group of instructions as a result of decisions made in the course of the computation. The same group of instructions (subroutine, loop) may be traversed again and again for repeated and iterative operations.

Since the instruction words are, just like data words, simply sets of 0s and 1s to the computer, its arithmetic/logic circuits can, if we wish, *modify* instructions in accordance with intermediate results. The extraordinary power of the modern digital computer is not simply due to its speed and memory capacity but also to the flexibility of the stored-program concept: branching, looping, and instruction or data-address modification permit us to create dramatically complicated programs from very simple instruction sets.

2-2. Single-address Computers. Perhaps the most "natural" computer instruction might, say, add two numbers A and B taken from memory by

specifying ADD WORD (addressed by) A AND WORD (addressed by) B; PUT RESULT INTO MEMORY LOCATION (addressed by) C. But specification of three separate addresses would make the instruction word too long (even for a large digital computer, not to speak of a minicomputer). We can, however, implement the above operation in terms of several simpler instructions each referencing only a single memory address:

LOAD INTO ACCUMULATOR (the word addressed by) A
ADD INTO ACCUMULATOR (the word addressed by) B
STORE ACCUMULATOR (in memory location addressed by) C

The "basic" minicomputer discussed in this chapter, then, will be a single-address machine whose instructions move data between a single suitably addressed memory location and a specified processor register, or possibly between two such registers. There will also be some instructions which do not reference memory at all (e.g., COMPLEMENT ACCUMULATOR). We remark, however, that the possibility of using simplified two-address instructions and zero-address (stack) instructions in minicomputers is of the greatest interest and will be discussed in connection with more advanced designs in Chap. 6.

2-3. The "Basic" Minicomputer. Figure 2-1 illustrates the typical organization of a small digital computer. The machine has all the ingredients of Secs. 1-6 to 1-13, viz.,

1. A core or semiconductor memory, which will store instructions and data
2. A set of processor registers (flip-flop registers), viz.,
   (*a*) **Memory buffer register** (memory data register): contains the instruction or data word currently leaving or entering the memory
   (*b*) **Memory address register:** contains the address of the currently addressed memory location
   (*c*) **Program counter:** contains the address of the instruction to be executed
   (*d*) **Instruction register:** contains the current instruction
   (*e*) **General-purpose register** (accumulator, arithmetic register) or registers; and, possibly, an index register (Sec. 2-7)
   (*f*) **One-bit registers ("flags"):** indicates overflow, carry, sign bit, etc., resulting from past or current operations
3. An **arithmetic/logic unit:** logic circuits to combine words from two registers by addition, subtraction, bit-by-bit ANDing, etc., and to complement, shift, etc., single words
4. **Control logic:** decodes the 0s and 1s of the instruction currently in the instruction register to generate logic levels and time pulses, which:
   (*a*) Gate (steer) words between processor registers
   (*b*) Determine the function of the arithmetic/logic circuits

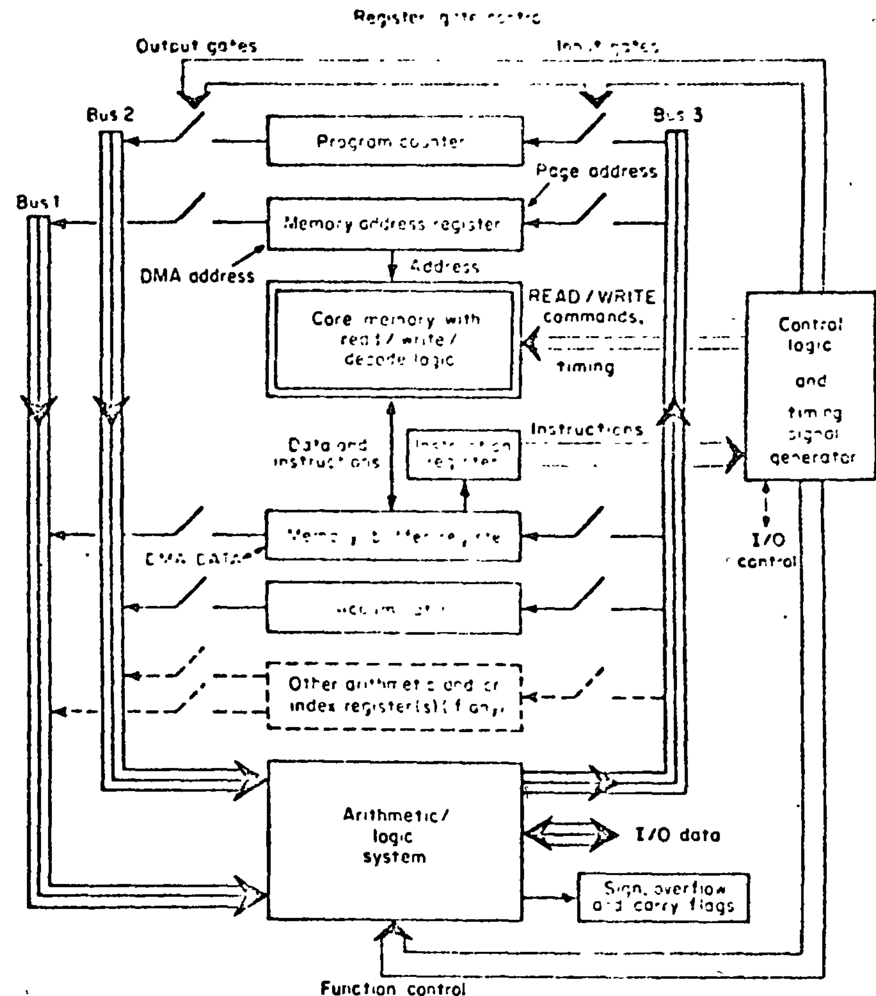


Fig. 2-1. Organization of a "basic" single-address minicomputer.

In Fig. 2-1, there are three buses for transfers between registers, always via the arithmetic/logic unit. This is a practical compromise: some extra register-to-register paths would permit more concurrent register transfers and speed computation, but we would pay for more complex interconnections and logic.

Finally, we must have input/output connections through the arithmetic/logic unit or through register gates.

2-4. Processor Operation. (a) Instruction Fetching. If we assume that a suitable program and data are in memory, processor operation proceeds

in clock-timed steps (microoperations):

1. The *program counter*, which we assume to be preset to the address of the first instruction (manually as in Sec 3-4 or by a preceding computer program), transfers this address to the *memory address register*.
2. The instruction word thus addressed is read into the *memory buffer register* and from there into the *instruction register*.

These instruction-fetch microoperations require one memory half-cycle (READ half-cycle, Sec. 1-12). A *core* memory must next restore the instruction word while its address is still in the memory address register (RESTORE half-cycle). Even with a core memory, operations which do not involve the memory can "overlap" the restoring operation. Hence most non-memory-reference instructions can be executed in a single memory cycle, which is commonly referred to as the FETCH cycle.

(b) Instruction Execution. As soon as the instruction word is in the instruction register its 0s and 1s are decoded to control processor operations on register and memory words and input/output operations.

When each instruction is completed, the program counter must contain the address of the next instruction in the program. Most instructions simply *increment* the program counter to produce the next instruction address, while *branching* instructions load the program counter with a nonconsecutive address (Sec. 2-9). The computer (unless halted) then proceeds with a new FETCH phase.

2-5. Instruction Sets. The timing-and-control block in Fig. 2-1 produces a sequence of timed pulses available for jam transfers, sensing, and clearing of words in registers, memory, and arithmetic circuits (see also Table 1-5*b* and Sec 1-6). In a conventional n-bit computer, the instruction register has at most n bits, which can be decoded to furnish up to 2^n gate-level combinations for steering pulses and/or words. We can, thus, have up to 2^n different one-word instructions (including input/output instructions). Unfortunately, each instruction referencing an operand or result in memory (memory-reference instructions, e.g., LOAD ACCUMULATOR, STORE ACCUMULATOR, JUMP, etc.) must address one of, say, 8,000 memory locations, and this *alone* requires 13 bits. To have a meaningful variety of instructions, even 18-bit machines do not use more than 13 bits for direct addressing, and most minicomputers use only 7 or 8 bits. It follows that *every minicomputer must sometimes and somehow employ multi-word instructions*; the extra word or words may be implied by an effective-address computation (Sec. 2-7). It is of at least academic interest that any [illegible] digital-computer programs can be implemented with very small instruction sets, but this requires many more instructions, which are costly

in terms of memory, time, and programming effort. Altogether, minicomputer design depends crucially on elegant compromises in coding broadly useful instruction sets into short instruction words (see also Chap. 6).

Table 2-1 lists the most common instructions used with minicomputers of the "basic" type shown in Fig. 2-1. Figure 2-2 illustrates instruction-word formats for such machines.

WHAT INSTRUCTIONS DO

2-6. Register-storing Instructions. Referring to Fig. 2-1, an instruction

STORE ACCUMULATOR IN	(effective address)
STORE ACCUMULATOR NO. 2 IN	(effective address)
STORE INDEX REGISTER NO. 2 IN	(effective address)

transfers the contents of the specified processor register to the memory data register via buses 2 and 3. Concurrently, the effective memory address implied by the instruction word is determined (Sec. 2-7) and loaded into the memory address register. The machine then deposits the register word into the effectively addressed memory location, whose previous contents are *lost*. The contents of the source register are *unchanged*.

The arithmetic/logic unit (Fig. 2-1) acts simply as a data-transfer path joining buses 2 and 3. Since both instruction fetching *and* execution require memory read/write operations, register-storing instructions require *two memory cycles*.

2-7. Addressing Modes and Index Registers (see also Sec. 4-9). **(a) Direct, Relative, and Indexed Addressing.** Memory-reference instructions like the register-storing instructions of Sec. 2-6 may have to address 4,000, 8,000, and even as many as 64,000 memory locations. But the minicomputer memory-reference-instruction formats of Fig. 2-2 have only 7 to 13 bits available for addressing since we need some operation-code bits to distinguish different memory-reference instructions. It is, therefore, necessary to *compute* an effective address by combining the instruction-word address bits with another digital word previously loaded into another processor register or memory location. This means, of course, that we may need two or more words to specify the memory-reference instruction completely. Every minicomputer employs two or more of the following addressing modes:

1. **Direct addressing on "page 0":** m address bits in the instruction word directly address memory locations 0 through $2^m - 1$.

With, say, $m = 8$, the resulting 256-word page does not go far, but we often use memory locations on page 0 for special purposes (interrupt trap locations, autoindex locations, etc.).

TABLE 2-1. Basic Instructions for Single-address Computers.

[illegible] most frequently supported instructions are shown in fat print (see also Secs. 2-6 to 2-12)

	Memory-reference Instructions (they may be indexed; most need two processor cycles, more with indirect address)	Non-memory-reference Instructions (one processor cycle)	Programmed Input/Output (to or from addressed device, two to four processor cycles)
Move word	**LOAD**, **STORE** (word or byte; specify register); STORE ZERO	**MOVE** (register to register); LOAD IMMEDIATE	**READ** (device-to-register); **WRITE** (register-to-device)
Arithmetic [illegible]	**ADD**; **SUBTRACT**; MULTIPLY, DIVIDE (usually optional)	**INVERT**; **INCREMENT**; **ROTATE SHIFT***; ADD, SUBTRACT (register-to register); ADD CARRY; SUBTRACT CARRY; SWAP BYTES	—
[illegible]	**AND**; **OR**; XOR	**CLEAR REGISTER***; SET REGISTER; **COMPLEMENT REGISTER***; CLEAR BYTE	Issue logic levels and/or timed pulses
Program control [illegible]		**HALT**; **NO OPERATION**	
[illegible] branch	EXECUTE, **JUMP** (need one cycle only); **JUMP AND SAVE**	**SKIP***	—
[illegible] condition [illegible]	EXECUTE ON CONDITION; **JUMP ON CONDITION**; **JUMP AND SAVE ON CONDITION**	**SKIP ON CONDITION***	**SKIP ON FLAG** (sense line)
[illegible] word	**SKIP IF REGISTER DIFFERS**; SKIP IF BYTE DIFFERS	—	
[illegible]	**INCREMENT MEMORY, SKIP IF ZERO**; DECREMENT MEMORY, SKIP IF ZERO	INCREMENT REGISTER, DECREMENT REGISTER (and set flag or skip if 0)	—
[illegible] condition (prepare for branch test)	—	CLEAR*, SET* (register, half register, sign bit, overflow flag and/or carry flag); **COMPLEMENT FLAG**	**CLEAR FLAG**; CLEAR ALL FLAGS; CHANGE INTERRUPT PRIORITIES

* [illegible] instructions [illegible] often be combined

2. **Direct addressing with a page register:** A processor page register loaded by an *extra instruction* adds enough high-order bits to the instruction address bits to address all of memory in terms of 2^m-word pages.
3. **Double-word direct addressing:** The second of *two consecutive instruction words* is or contains the address. 16-bit machines can address all of memory in this way. 8-bit minicomputers always use double-word addressing, but will still need paging and/or relative addressing, or more words.
4. **Direct addressing on the current page:** The m address bits (*page address*) are *augmented by the high-order bits of the current program-counter reading*, which constitute the *current page number*.
5. **Addressing relative to the program counter (relative addressing):** The instruction-word address bits are interpreted as a signed integer, which is *added to the current program-counter reading* to determine the effective address.

Current-page and relative addressing require no page-setting instruction, and experience indicates that many programs mostly reference memory locations close to the current program-counter reading. *Relative addressing simplifies program relocation* (Sec. 4-18).

6. **Addressing relative to an index register:** The instruction-word address bits are interpreted as a signed or unsigned integer, which is *added to the contents of a specified* index register to determine the effective address.

Index registers are extra processor registers holding a full computer word ("base address"). They can be cleared, loaded, incremented and/or decremented by special instructions (Sec. 2-10). Index addition takes no extra time or, at most, a fraction of a memory cycle. Index modification is, therefore, a good way to address different elements of *data structures* (Secs. 4-9 to 4-11). Index registers can also be used as temporary storage registers and may or may not serve as accumulators as well.

Most minicomputer indexing operations involve only one index register at a time.

(b) Indirect Addressing, Preindexing, and Postindexing. Indirect addressing, specified by an instruction-word bit, means that the address found by paged, double-word, relative, or indexed addressing does not itself contain the desired operand but rather its effective address in memory. Our memory-reference instruction, then, addresses a memory location whose contents serve as a *pointer* to the desired operand.

Some minicomputers permit **multilevel indirect addressing**; i.e., an indirect-address bit in the pointer indicates that it points not to the final operand but to yet another pointer.

Indirect addressing is the key to vitally important programming techniques. Since an indirectly addressed operand is a function of its pointer, we can

implement *table lookup* (function and directory tables) and modify pointers to access different elements of *data structures* (Secs. 4-9 to 4-11). Indirect addressing can be combined with indexing:

1. **Preindexing:** The instruction-word address bits, interpreted as a signed or unsigned integer, are added to the contents of a specified index register to determine the *pointer address*.
2. **Postindexing:** The indirect address contains an index bit or bits specifying an index register. *The contents of this index register are added to the indirect address to form the effective address.* The pointer stays in memory without change.

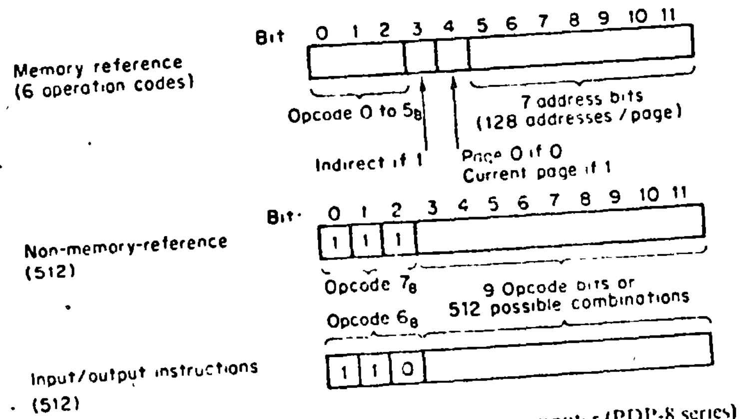


Fig. 2-2a. Instruction formats for a simple 12-bit minicomputer (PDP-8 series)

(c) Autoindexing. Some minicomputers have 8 to 16 special memory locations on page 0 called autoindex registers. When one of these autoindex registers is indirectly addressed, it produces the effective address by automatically incrementing or decrementing its contents. Autoincrementing or autodecrementing is a cheap way to address successive elements of arrays (Sec. 4-9); no index register needs to be loaded, and the only price paid is the extra memory cycle required for indirect addressing. Note also that no instruction-word bit is needed to produce autoindexing, but the autoindex-register locations cannot be used for ordinary indirect addressing. As an alternative, an extra instruction-code bit can be used to specify incrementing or decrementing of *any* indirect address.

(d) Microoperations Determining the Effective Address. We have already noted that minicomputers always offer a *choice* of several addressing modes. Opcode bits in the memory-reference-instruction words are used to select the addressing mode (*direct/indirect, return to page 0, relative,* and/or *choice of index register*). Figure 2-2 shows typical instruction formats. Good ... instructions like STORE ACCUMULATOR IN A and

automatically effect paged, relative, indirect, or double-word addressing to get around page boundaries (Sec. 4-2).

Referring to Fig. 2-1, a *direct-address instruction* transfers its address bits from the memory data register to the memory address register during the EXECUTE phase of the first instruction cycle by way of bus 1, the arithmetic/logic unit, and bus 3. Program-counter bits (for current-page addressing) or page-register bits are transferred at the same time. *Relative-address and index instructions* employ the arithmetic/logic unit to add program-counter or index-register contents via bus 2. Larger computers may have a separate adder for address computations. *Double-word and indirect-address instructions* use an extra memory cycle to transfer the pointer from

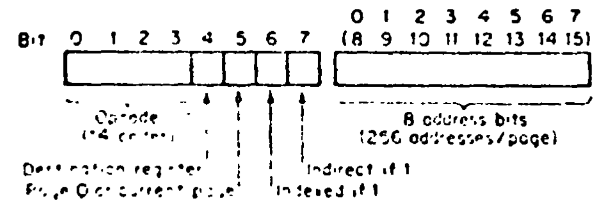


Fig. 2-2b. This could be a one-word memory-reference instruction for a 16-bit minicomputer or a two-word memory-reference instruction for an 8-bit machine (see also Fig. 6-8)

memory into the memory data register and then to the memory address register (with postindexing, if any). *Autoindexing* requires no more time than ordinary indirect addressing but needs some extra logic.

NOTE: Double-word instructions must increment the program counter *twice*. Also, in 8-bit minicomputers, indirect addresses can be double 8-bit words, so that indirect addressing can take *two* extra memory cycles.

(e) Immediate Addressing. One-word immediate-addressing instructions are not really memory-reference instructions, although instruction formats are similar. In each immediate-address instruction, e.g.,

```
LOAD ACCUMULATOR IMMEDIATE                (integer)
ADD INTO ACCUMULATOR NO 2 IMMEDIATE       (integer)
```

what looks like the address bits is interpreted as a signed or unsigned integer operand. One-word immediate-address instructions are completed in a single memory cycle and are, therefore, handy for setting up index registers, counters, etc. Some minicomputers admit two-word two-cycle immediate-address instructions with the operand stored as the second of two instruction words.

2-8. Memory-to-register Operations: (a) How They Work Such operations are fundamental to single-address arithmetic. Referring to Fig. 2-1,

each of these instructions will fetch the contents of a suitably addressed memory location into the memory data register and then via bus 1 to the arithmetic/logic system. With appropriate bus gating, the arithmetic/logic unit can now combine the memory word with a word fetched from a processor register (accumulator or index register specified by the instruction) via bus 2. The result is loaded into *the same* processor register via bus 3. The original register contents are lost.

Since both instruction fetching *and* execution require memory read/write operations, each of these instructions typically takes *two memory cycles*. If a memory-to-register instruction reads from a core memory, the original word is immediately rewritten (Sec. 1-12) so that *memory contents are unchanged*. Memory contents are obviously also unchanged in memories with non-destructive readout (read-only magnetic memories, semiconductor memories).

(b) Register Loading. The simplest memory-to-register instructions merely *load* a specified register with a word taken from memory:

LOAD (INTO) ACCUMULATOR	(effective memory address)
LOAD (INTO) ACCUMULATOR NO. 1	(effective memory address)
LOAD (INTO) INDEX REGISTER NO. 2	(effective memory address)

Effective addresses are specified as in Sec. 2-6.

Some minicomputers (e.g., PDP-8) omit the LOAD instruction and load the accumulator by adding or ORing into it after first clearing the register. A few minicomputers also have the useful two-cycle instruction INTERCHANGE ACCUMULATOR AND MEMORY.

(c) Memory-to-register Arithmetic: Overflow and Carry Flags (see also Sec. 2-10). The most important (and often the only) memory-to-register arithmetic operation is *addition:*

ADD INTO ACCUMULATOR NO. 2	(effective memory address)

The sum will overflow if and only if both terms have identical signs *and* the sign of the sum turns out to be different. We can detect 2s-complement overflow by combining the carries from the sign bit and the most significant bit in an XOR gate: overflow occurs if there is a carry from one of the two but not both.

2s-*complement arithmetic* (Sec. 1-9) is usually implied, but consult your manual. PDP-15, for instance, has *both* 1s-complement *and* 2s-complement addition (ADD and TAD).

Some minicomputers permit *subtraction of a memory word from the contents of a register:*

SUBTRACT INTO ACCUMULATOR NO. 1	(effective memory address)

and a few have *inverse subtraction:*

SUBTRACT INTO ACCUMULATOR—THEN INVERT RESULT (effective memory address)

(That is, multiply the original difference by −1.) 2s-complement arithmetic is implied.

After an addition or subtraction, the processor sets special flip-flops ("flags")

1. If *arithmetic overflow* (Sec. 1-9*a*) occurs (overflow flag), and/or
2. If there is a carry out of the most significant (sign) bit in the register (carry flag, extend flip-flop, link)

Carry flags are useful in positive-integer arithmetic, 1s-complement arithmetic, and double-precision arithmetic (Secs. 2-14 and 4-11).

Check your computer manual carefully: Some minicomputers have only an overflow flag or only a carry flag (PDP-8 series). PDP-15 indicates carries and 1s-complement overflow with the same flag, but has no 2s-complement overflow flag. In any case, note that *minicomputer overload flags will not by themselves halt or change the computer program*. It is up to the programmer to "test" the flag with suitable skip or branch instructions; the programmer must also be sure to *clear* overload and carry flags before they are needed.

Memory-to-register *multiplication* and *division* require multiple (extended) arithmetic registers. Most minicomputer manufacturers sell hardware for these operations as special *options*, which will be described in Sec. 2-14. The "bare" processor can perform multiplication and division as subroutines involving addition and shifting.

(d) Memory-to-register Logic. The *memory-to-register logic instructions*

AND INTO ACCUMULATOR	(effective memory address)
OR INTO ACCUMULATOR NO. 1	(effective memory address)
XOR INTO ACCUMULATOR NO. 1	(effective memory address)

perform the indicated operations *bit by bit* on corresponding pairs of bits from register and memory, with the result left in the register. Thus, if an 8-bit accumulator and the effectively addressed 8-bit memory word contain

	01110101
and	11110001

respectively, then

AND produces the accumulator contents	01110001
OR produces the accumulator contents	11110101
XOR produces the accumulator contents	10000100

(See also Sec. 1-6.) In practice, AND is used to replace selected bits of a register word with 0s set up in a *mask* word in memory. OR will similarly replace selected bits with 1s. XOR produces 0 bits wherever the two original words agree (coincidence check). Some applications will be discussed in Sec. 4-11.

2-9. Operations on Words in Memory. The two-cycle instruction

STORE ZERO IN (effective address)

clears the effectively addressed memory location *without* affecting the contents of any accumulator. Minicomputers without a STORE ZERO instruction must clear and deposit an accumulator.

The two-cycle instruction[1]

INCREMENT, SKIP IF ZERO (effective address)

(ISZ) moves the contents of the effectively addressed memory location into the arithmetic/logic unit via the memory data register. The number is incremented and returned to its memory location (again via the memory data register). If the incremented result is 0 (i.e., if incrementing causes a carry), then the program counter is made to skip (i.e., it is incremented by 2 rather than by 1). *This conditional skip lets the program branch in the manner of Sec. 2-11c.*

The STORE ZERO and INCREMENT, SKIP IF ZERO instructions are used to clear and increment a counter in the effectively addressed memory location. To implement a counter preset to a count of N, we store $-N$ in a memory location and then ISZ until the program branches after N ISZs (Sec. 4-9). Some computers have a similar DECREMENT, SKIP IF ZERO instruction.

2-10. Operations on Register Words and Flag Bits. (a) Register Arithmetic/Logic. Instructions like

CLEAR ACCUMULATOR
COMPLEMENT ACCUMULATOR NO. 2
INCREMENT INDEX REGISTER

move the contents of a specified register into the arithmetic/logic unit and back again to perform the indicated operation in one memory cycle (FETCH cycle, Sec. 2-4*a*. See also Fig. 2-1). COMPLEMENT ACCUMULATOR produces the 1s-complement negative of a signed number in the register. INVERT ACCUMULATOR is the same as NEGATE ACCUMULATOR or COMPLEMENT AND INCREMENT ACCUMULATOR and produces the 2s-complement negative (see also Tables 1-1 and 1-2).

Most minicomputers have an instruction or instructions like

MOVE ACCUMULATOR NO. 2 TO ACCUMULATOR NO. 1
MOVE ACCUMULATOR TO INDEX REGISTER

which move (transfer) register contents in one cycle via buses 2 and 3 (Fig.

[1] In some machines

2-1); the contents of the source register remain unchanged. Most minicomputers do not permit register-to-register addition or subtraction. Some machines have a one-cycle instruction to INTERCHANGE ACCUMULATOR CONTENTS.

The LOAD IMMEDIATE operation, which loads the instruction address bits into a specified register, was already discussed in Sec. 2-7*c*. Similar addition, subtraction, AND, and OR operations may also be implemented.

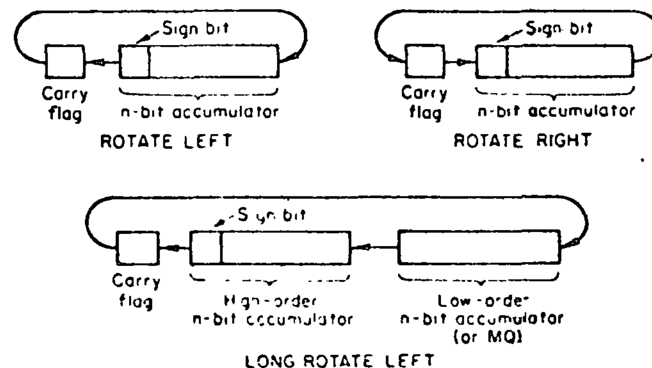


Fig. 2-3. ROTATE/SHIFT operations. The carry-flag flip-flop can be set, reset, or equated to the sign bit before each 1-bit rotation.

(b) Rotate/Shift Operations. One-cycle instructions like

ROTATE ACCUMULATOR LEFT
ROTATE ACCUMULATOR NO. 1 RIGHT

rotate (circulate) the contents of the specified register and the carry bit by 1 bit, as shown in Fig. 2-3. Some minicomputers also admit 2-bit rotations. Physically, the register bits go to the arithmetic/logic unit via bus 2, are "shifted sideways" by means of gates (Sec. 1-9*b*), and return to the register via bus 3.

Rotation has three important applications:

1. Individual bits of a register word (which might indicate the logical states in some external device, Sec. 5-8) can be rotated into the sign-bit and/or carry-bit position for tests and branching (Sec. 2-11).
2. Partial words or bytes can be moved, packed, and unpacked in connection with input/output operations (see also Sec. 4-11).
3. With the carry bit appropriately cleared or set, rotations act as arithmetic shifts implementing multiplication or division by 2 (Sec. 1-9 and Tables 1-1 and 1-2).

Specifically, an unsigned binary number (no sign bit) is multiplied or divided by 2 if we first clear the carry flag and then rotate, respectively, left or right. Such an operation is called an unsigned shift. After multiplication by 2, a 1 in the carry flag indicates overflow.

A signed 1s-complement number is multiplied or divided by 2 if we first make the carry bit equal to the sign bit and then rotate. Such an operation is a signed shift.

A signed 2s-complement number is multiplied by 2 through an unsigned left shift but divided by 2 through a signed right shift.

Overflow of any such multiplication by 2 is indicated if the sign and carry bits differ. Each division by 2 will "chop" rather than round the result to the given number of bits; i.e., the result is always less than or equal to the correct result.

Some minicomputers have explicit SIGNED SHIFT and UNSIGNED SHIFT instructions. Some machines can also rotate a register *without* the carry flag (see also Secs. 2-11, 2-12, and 2-14).

(c) Operations on Flag Bits. A number of one-cycle instructions permit the program to clear, set, and complement specified processor flip-flops such as carry and overflow flags, e.g.,

CLEAR CARRY FLAG

This may be done in preparation for conditional branching (Sec. 2-11*c*), to store 1-bit decisions for later use, or in connection with arithmetic shifts (Sec. 2-10*b*). Bit operations are often combined with rotation and/or conditional skips (Sec. 2-11*d*). ADD CARRY and SUBTRACT CARRY (into accumulator; and clear carry flag) are useful for double-precision operations (see also Sec. 2-14).

2-11. Instructions Controlling Program Execution and Branching. (a) NO OPERATION and HALT. The one-cycle instruction NO OPERATION does nothing except advance the program counter to the following instruction and serves as a time delay or as a "spacer" for later insertion of another instruction or data word. HALT advances the program counter and stops processor operation to give the operator a chance to examine or change registers, switch settings, and/or peripheral-device operation.

(b) *Unconditional Branching.* The one-cycle instructions SKIP and

JUMP TO (effective address)

are employed for program branching (Secs. 4-8 and 4-9) and also to "jump around" memory locations used to store data between instructions (Secs. 4-5 and 4-14). SKIP simply increments the program counter twice to jump over one memory location. JUMP resets the program counter to [illegible] value; the old program-counter setting is *lost*. By

contrast, the two-cycle instruction

JUMP AND SAVE (effective address)

resets the program counter (and thus causes a jump) to the effective address plus 1 or 2 and saves the (incremented) old program-counter setting at the effective-address location. JUMP AND SAVE permits one to return to the original program after a subroutine (Sec. 4-14) and is, therefore, often referred to as JUMP TO SUBROUTINE.

In some machines, JUMP AND SAVE automatically saves not only the return address but also the page register and/or carry, overflow, etc., flags (combined at the effective address plus 1). More sophisticated minicomputers also automatically increment a *stack pointer* to keep track of reentrant-subroutine nesting (Secs. 4-16 and 6-10). More advanced computers can have a JUMP AND SAVE IN INDEX instruction, which stores the return address in an *index register* rather than in memory. This speeds up subroutine processing by avoiding memory references (Secs. 4-14 and 6-10).

The one-cycle instruction

EXECUTE (effective address)

causes execution of an instruction stored at the effectively addressed memory location and then continues with the program; this amounts to the execution of a one-instruction subroutine.

(c) Conditional Branching. Each one-cycle instruction

SKIP ON CONDITION

causes a skip subject to a condition or conditions specified by instruction-code bits, e.g.,

ACCUMULATOR = 0	CARRY FLAG = 0
ACCUMULATOR NO. 2 < 0	CARRY FLAG = 1
INDEX REGISTER > 0	OVERFLOW FLAG = 1

Different instruction-bit combinations can produce logical ORing or ANDing of such conditions, e.g., ACCUMULATOR ≥ 0 (see also Sec. 2-11*d*).

There are also two types of *two-cycle* conditional-skip instructions which reference memory. INCREMENT (or DECREMENT), SKIP IF ZERO was introduced in Sec. 2-9. The second type is exemplified by

SKIP IF ACCUMULATOR DIFFERS FROM (effective address)
SKIP IF ACCUMULATOR NO. 2 EQUALS (effective address)

SKIP ON CONDITION instructions are the (only) way most minicomputers implement conditional branching, e.g.,

```
SKIP ON CONDITION                          /Condition true?
JUMP TO        (effective address)         /No, go to branch 2
(next instruction)                         /Yes, continue on branch 1
   . . .
```

(Sec. 4-8). Only a few minicomputers have "direct" conditional-branching instructions, i.e., JUMP ON CONDITION, JUMP AND SAVE ON CONDITION, and EXECUTE ON CONDITION.

(d) **Combined Register/Flag Operations, Rotations, and Tests.** Most computers can implement certain *combinations* of register/flag clearing, setting, or complementing, a rotation, and/or a skip test through single one-cycle instructions. Appropriate instruction-code bits will call for the individual operations, and the *programmer must be sure to understand their relative order of execution*. For instance, to multiply an unsigned number in a register by 2, one must *first* clear the carry flag, *then* rotate left, and *then* test for a carry indicating overflow (Sec. 2-10*b*). **Check the reference manual and also the assembler manual for your specific computer.**

An especially useful one-cycle combination instruction is INCREMENT (or DECREMENT) INDEX REGISTER, SKIP IF ZERO, which is used to implement and terminate program loops (Sec. 4-9).

2-12. Input/Output-related Instructions. Each minicomputer instruction set must reserve a respectably large number of different instruction-code-bit combinations for input/output instructions intended to select and operate external devices (Secs. 5-2 to 5-8, Table 5-1). For example, the Hewlett-Packard 2115A, which is a typical "basic" 16-bit minicomputer, admits $2^{12} = 4{,}096$ different one-word input/output instructions, and the 12-bit PDP-8 series admit $2^9 = 512$ such instructions. In addition, each minicomputer has some instructions for controlling its interrupt system, such as INTERRUPT ON and INTERRUPT OFF (Secs. 5-9 to 5-15).

SPECIAL FEATURES, INSTRUCTIONS, AND OPTIONS

2-13. Byte Manipulation. 8-bit minicomputers naturally handle 8-bit bytes holding an ASCII character plus parity or two BCD digits. A 16-bit word holds two such bytes. Most 16-bit minicomputers have at least one or two one-cycle **byte-manipulation instructions**

CLEAR LEFT (or RIGHT) ACCUMULATOR BYTE
INTERCHANGE ACCUMULATOR BYTES
INTERCHANGE AND CLEAR LEFT (or RIGHT) ACCUMULATOR BYTE

Such instructions replace multiple ROTATEs and ANDing with mask words and can save much time and memory in character-handling programs (e.g., text editing, communications).

Some 16-bit machines permit byte addressing of LOAD and STORE ACCUMULATOR instructions. Byte addressing is specified by an opcode bit or by a status register set through a special instruction (BYTE mode)

We load or store accumulator bits 8 through 15, while bits 0 through 7 remain unaffected. The effective address refers to individual bytes in memory, so an extra address bit will be needed to specify even or odd bytes. Another type of byte-addressed instruction is

SKIP IF ACCUMULATOR BYTE DIFFERS (effective address)

which is useful for detecting special characters in text strings.

2-14. Arithmetic Options. (a) Double Store, Load, Add/Subtract, and Rotate/Shift Operations. To simplify double-precision operations, some minicomputers can store the contents of two accumulators in successive memory locations through a single (usually three-cycle) instruction

DOUBLE STORE (effective address)

DOUBLE LOAD similarly loads two accumulators from successive memory locations. More extensive facilities for double-precision operations usually come only as part of extra-cost hardware multiply/divide options. The Honeywell 316/516 high-speed arithmetic option, for instance, has DOUBLE ADD and DOUBLE SUBTRACT, with an automatic carry from the low-order accumulator to the high-order accumulator. To accommodate so many extra memory-reference instructions, the 316/516 must first set a status register to DOUBLE PRECISION through a separate instruction. The *sign bit* of the double-precision number is usually bit 0 of the high-order accumulator (Fig. 1-17).

Two accumulators can be similarly concatenated (together with the carry flag) for double-precision LONG ROTATE, LONG UNSIGNED SHIFT, and LONG SIGNED SHIFT operations (see also Sec. 2-10*b*). Most extended-arithmetic options have instructions for multiple shifts; the number of bits shifted is determined by an extra processor register, the shift counter.

With a binary fraction in the double accumulator, the instruction NORMALIZE will shift the double fraction left until its most significant bit differs from the sign bit (see also Sec. 1-10). Some computers use ordinary long shifts and test the result with a special instruction SKIP IF ACCUMULATOR IS NORMALIZED

(b) Hardware Multiply/Divide Options (see also Sec. 1-9). Multiply/divide hardware always requires two arithmetic registers to hold a double-precision product or dividend. It is best if these registers are general-purpose accumulators accessible through DOUBLE STORE and DOUBLE LOAD instructions (Sec. 2-14*a*; e.g., Hewlett-Packard and Honeywell minicomputers). A less desirable arrangement adds a special multiplier/quotient (MQ) register, which is harder to access (PDP-8 series, PDP-9/15).

The better hardware multiply/divide options place no restrictions on

operand signs, employ 2s-complement arithmetic, and have simple instructions

MULTIPLY	(effective address)
DIVIDE	(effective address)

It is most convenient to interpret operands and the result either as *signed binary integers* or as *signed binary fractions* (Tables 1-1 and 1-2 and Sec. 1-9).

NOTE Many popular minicomputers (PDP-8 series, NOVA/SUPERNOVA) implement *unsigned* multiplication/division (unsigned nonnegative operands and result). This produces some extra precision since no bits are needed as sign bits, but signed multiplication/division then requires cumbersome multiple-instruction sequences, which waste time and memory. PDP-9/15 has basically unsigned multiplication/division with some special extra instructions attempting to simplify signed operations which are, however, still inconvenient.

The multiplication $A \times B = C$ starts with A (single-precision) in an accumulator and B (also single-precision) in the effectively addressed memory location. The double-precision result C appears in a pair of accumulators (or, less desirably, in an accumulator plus index register or MQ register).

The division $C \div B = A$ starts with the double-precision dividend C in the two registers and B (single-precision) in the effectively addressed memory location. The quotient A will appear in the high-order register (accumulator), while the remainder will be left in the low-order register. Unlike multiplication, division can cause overflow, which should be detectable by a flag test; consult your minicomputer manual.

In some minicomputer multiply/divide units (PDP-9/15, PDP-8 series except for PDP-8e), the operand B cannot be taken from an arbitrary memory location but must be placed into the location following the MULTIPLY or DIVIDE instruction.

Typical hardware multiply/divide times are between 5 and 35 machine cycles. This compares with between 70 and 300 cycles required for non-hardware multiply/divide subroutines.

2-15. Miscellaneous Options. The following useful options are offered by many minicomputer manufacturers:

1. **Extra memory** may simply require extra plug-in modules, or one may have to add a page register or extended memory address register to the processor. **Read-only memory** (ROM, Sec. 1-14), often interchangeable with ordinary memory-bank modules, stores important programs or routines "firmly" ("firmware"). Some minicomputers yield faster cycle times for instructions read from ROM (Sec. 6-5).
2. **Parity-check interrupt** (Sec. 1-4c) on all word or byte transfers to and from memory requires an extra bit per memory word, plus parity logic. This option may be useful, e.g., in critical process-control applications. It is not really needed in most end-user installations.

3. **Memory protection,** which usually also requires an extra bit per memory word, protects preselected areas of memory from unauthorized users. This is done to protect system programs from overwriting and to protect time-sharing users from each other. Either all instructions referencing unauthorized memory locations, or STORE instructions only, cause interrupts which usually return control to an executive program (Sec. 3-11). Memory-protection hardware is operated by a set of special instructions, which permit the system programmer to "tag" selected memory areas for protection. The computer user is not directly concerned with these instructions.
4. **Power-failure protection/restart:** Low power-supply voltage causes an interrupt, and a service routine stores all processor registers safely in core memory before the power-supply capacitors can discharge. A restart routine makes it easy to restore the registers. With semiconductor memories, a trickle-charged battery keeps the computer working while memory as well as register contents are saved on a disk or on tape.
5. **Extra interrupts** and/or **more sophisticated interrupt logic** (Secs. 5-9 to 5-16).
6. **Hardware floating-point arithmetic** is often a small accessory processor; it is still fairly expensive but is potentially very useful (Secs. 6-12 and 6-13).
7. **Automatic bootstrap loader** is a hard-wired program to load system programs from paper tape or magnetic tape (Sec. 3-4*b*).
8. **Indicator-light test switch:** This small feature avoids surprises due to panel-light failures.

Other options will deal with improved or additional input/output circuits (Chap. 5), peripheral equipment, and software.

REFERENCES AND BIBLIOGRAPHY

Consult the minicomputer reference manuals of various manufacturers for specific detailed instruction lists, execution times, and other hardware features. See also the bibliographies of Chaps. 1 and 6.

CHAPTER 3

MINICOMPUTER OPERATION AND SOME PROGRAMMING, WITHOUT ASSEMBLY LANGUAGE

INTRODUCTION AND SURVEY

In this chapter, we describe the *front-panel operation of small digital computers*, including the most common procedures for *loading, translating, and executing computer programs* (Secs 3-1 to 3-6). To squeeze the last bit of efficiency from the minicomputer hardware, we will have to learn some assembly language (Chap. 4), but many small computers do remarkably well with FORTRAN and BASIC, which are more convenient for general problem solving (Secs. 3-7 and 3-8). To make a general-purpose minicomputer center truly powerful and convenient, we must get away from paper-tape operation. The remainder of this chapter deals with the hardware (*small disks, tape units, cathode-ray-tube/keyboard terminals*) and software (*executive systems or monitors*) which make comfortable minicomputation possible; a discussion of *on-line editing* is included.

CONTROL PANEL AND PAPER-TAPE OPERATION

3-1. The Operator's Control Panel. A typical minicomputer control panel (Fig. 3-1) will have the following controls and indicators:

1. A key-operated main power switch with three positions: ON, OFF, and LOCK PANEL. In the latter position, power is ON, but all front-

panel controls are ineffective -this keeps visitors from ruining computations by playing with the controls.

2. **Indicator-light fields**, which display the contents of the principal processor registers for examination. Smaller machines may have only one indicator field, which can display different processor registers selected by a REGISTER SELECTOR switch.

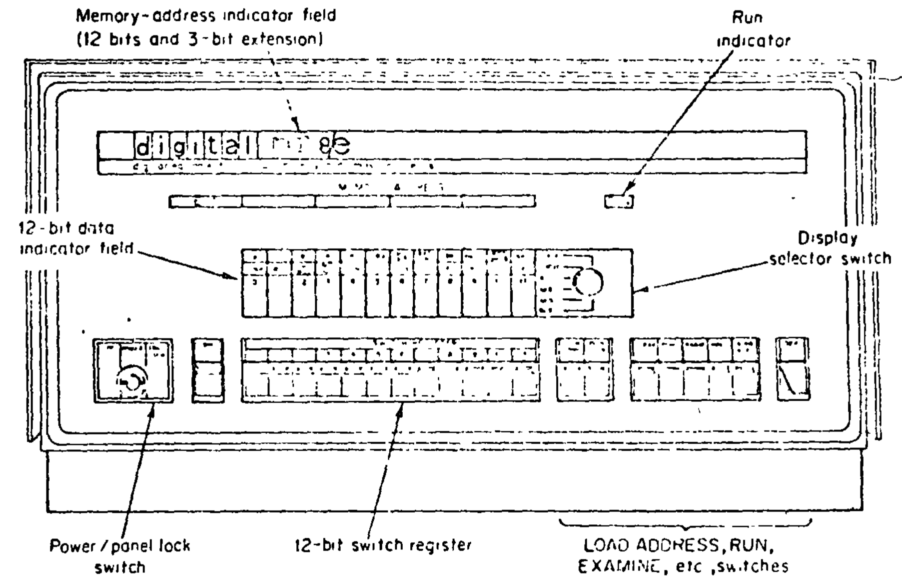


Fig. 3-1. A typical minicomputer control panel (Digital Equipment Corporation PDP-8 e a 12-bit machine) Individual indicator fields display memory address and data. a selector switch connects various processor registers, or a set of status indicators for display in the data field. Indicator fields and switch register are arranged in 3-bit groups to simplify octal-number interpretation

3. **A switch register or registers** for entering binary numbers bit by bit into a processor register selected by a REGISTER SELECTOR switch or by the LOAD ADDRESS and DEPOSIT switches.
4. Various switches.
   (*a*) A REGISTER SELECTOR switch selects the processor register connected to indicator and/or switch registers.
   (*b*) A LOAD ADDRESS switch loads the memory address register (in some machines also the program counter) with the number set into the switch register.
   (*c*) A DEPOSIT switch loads the currently addressed memory location with switch-register contents.

5. An EXAMINE (FETCH) switch fetches the contents of the currently addressed memory location into the memory data register for front-panel display.
6. Controls for starting, stopping, and stepping processor operation.
   (*a*) The START (RUN)/STOP switch starts the program with the current register contents.
   (*b*) SINGLE INSTRUCTION and SINGLE CYCLE switches for "stepping" the program one instruction or one processor cycle at a time; they are used for troubleshooting hardware or programs.

Additional controls and indicators may be provided. Some minicomputers have a READ IN switch for starting a paper-tape reader or even for automatic loading (Sec. 3-4). Sense switches on the front panel may permit the operator to modify a program while it is running (Sec 5-8; in other machines, sense lines are available only in peripheral devices). As further aids in troubleshooting, there may be indicators for the current processor status, e.g., INSTRUCTION FETCH, EXECUTE, INPUT/OUTPUT, INTERRUPT, etc.

Some minicomputers have a front-panel CLEAR switch, which clears a selected processor register or registers and which may also send a clear pulse to the computer peripheral devices for clearing appropriate flags and/or registers.
Some machines (PDP-15) have an I/O instruction to read their front-panel switch register during computation.

The operator's control panel is used mainly for starting programs and for troubleshooting through examination of register contents and stepwise program execution. Original-equipment manufacturers (OEMs) using minicomputers with little reprogramming may wish to purchase machines without elaborate control panels; service technicians can then carry *plug-in control panels* for start-up and diagnostic work.

3-2. Typical Control-panel Operations. Please be sure to note that specific front-panel controls and their operation will vary somewhat for different computers—you must consult the operator's manual for your own machine. The following operations are typical:

1. With the computer halted by the START/STOP switch, we can examine and change the contents of registers and memory locations. To examine a memory location, we set its address into the switch register and press the LOAD ADDRESS switch. The EXAMINE switch will now bring the contents of the addressed location into the memory data register for display.
2. To load a memory location manually, we set its address into the switch register and press LOAD ADDRESS. Then we set the desired binary number into the switch register and press DEPOSIT.
3. It will be useful to examine or load *successive* memory locations. For this purpose, we must increment the memory address register between successive EXAMINE or DEPOSIT operations. Different minicomputers do this in different ways, e.g.,
   (*a*) In the PDP-8I, LOAD ADDRESS sets the address into the program counter as well as in the memory address register. Program counter and memory address are incremented after *every* EXAMINE or DEPOSIT operation.

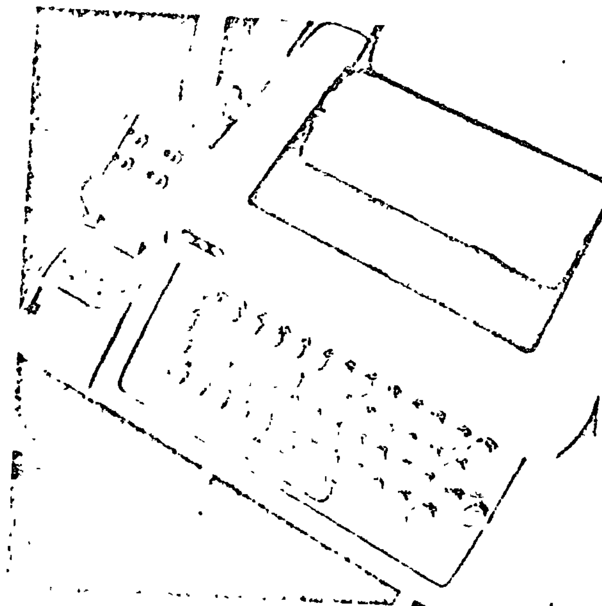


Fig. 3-2*a*. ASR-33 teletypewriter console. The paper-tape reader/punch is on the left.

   (*b*) The more elaborate PDP-9 has *special switch positions* (EXAMINE NEXT, DEPOSIT NEXT) which step the memory address before fetching or depositing.
   (*c*) In the PDP-11, *repeated* operation of the EXAMINE or DEPOSIT switch steps the memory address register.

3-3. The Console Typewriter. Most minicomputers are furnished with an ASR-33, ASR-35, KSR-33, or KSR-35 printer/keyboard (teletypewriter) manufactured by the Teletype® Corporation (Fig. 3-2).[1] With the OFF/LINE/LOCAL switch in the LOCAL position, the teletypewriter is disconnected from the computer and acts like a typewriter with the special character set shown in Fig. 3-2*b*. In the LINE position, the keyboard

[1] ASR stands for Automatic Send Receive, while KSR stands for Keyboard Send Receive. ASR-37 has both uppercase and lowercase characters and permits 15 character/sec operations, but is substantially more expensive.

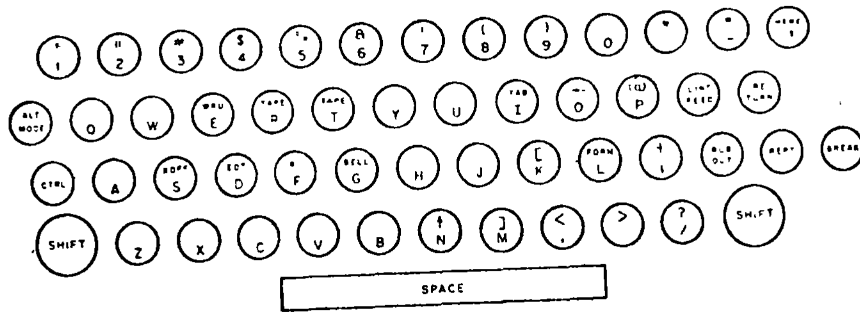


Fig. 3-2*b*. ASR-33 teletypewriter keyboard. Note the following:

RETURN returns printer to start of current line.
LINE FEED advances printer one line (without return unless RETURN is also depressed).
FORM FEED advances printer to the top of a new page (without return).

The nomenclature on the extra keys is intended for communications applications, not for computing, but the extra keys are useful.

transmits 8-bit ASCII character sequences (Table A-9) to the computer, and the printer can accept and output ASCII characters. These machines can print up to 10 characters/sec. They produce only capital letters but do have two shift keys (SHIFT and CTRL), which produce special characters or control functions when depressed simultaneously with other keys (see also Fig. 3-2*b*). Some of these special functions will depend on specific computer programs; conventional interpretations are listed in Table A-7.

The ASR models have a slow (10 character/sec) **paper-tape punch and reader;** in the LOCAL mode, we can punch the tape from the keyboard or get printed output through the paper-tape reader. In the LINE mode, we can read paper tape into the computer or let the computer punch paper tapes. Program preparation with the console typewriter will be further discussed in Sec. 3-16.

ASR-33 and KSR-33 are designed "for intermittent light duty," and this means exactly what it says. Teletypewriters will not last long if you use them as line printers for long listings. Even the "continuous-duty" ASR-35 and the KSR-35 teletypewriters are really designed for use in communications offices, where they are rebuilt on a regular schedule. Altogether, teletypewriters are the most frequent source of minicomputer troubles, and repairs are not cheap. To save your teleprinter, we suggest substitution of a cathode-ray-tube keyboard terminal for conversational input/output: this is also much faster and more pleasant to operate. Use the printer only when you really need hard copy, and keep the printer motor turned off as much as possible. If you require much hard-copy output, get a small line printer. The Digital Equipment Corporation DECwriter (Fig. 3-3), which [illegible]s about as much as a KSR-35 and is faster and also mechanically

simpler, is another useful alternative. An IBM Selectric typewriter with an adapter base plate for computer control is another possibility.

3-4. Loading and Running Simple Programs with Paper Tape. (a) **Manual Loading.** An executable program (which may or may not have some data attached to it) is, as we have seen, a sequence of multibit computer words. We might have such a program in binary form (or in the more convenient octal form, Sec. 1-4*b*) on a sheet of paper; **we must enter the program words into appropriate (usually consecutive) memory locations in the computer.** A simple-minded way to *load the program* is to use the front-panel controls:

1. Select a memory location for the first program word (which could be an instruction or a data word) via the switch register and the LOAD ADDRESS switch.
2. Load successive program words into consecutive memory locations with the aid of the switch register and the DEPOSIT switch, as shown in Sec. 3-2.
3. Set the actual starting address (address of the first instruction) into the program counter via switch register and selector or DEPOSIT switch.

The program is now ready to run if we press the RUN switch. As the program runs, it will *output data* via the teletypewriter, paper-tape punch, or other peripherals. The program may also *read input data* (or additional input data) from the teletypewriter, paper-tape reader, measuring instruments, etc.

(b) Paper-tape Systems and Bootstrap Loaders. Practical programs can have hundreds or thousands of words. Manual loading is clearly impractical, and programs are prepared (and stored for repeated use) on a **computer-readable storage medium,** usually punched cards, punched paper tape, or magnetic tape. These media are compared in Table 3-3. Most minicomputers are available with paper tape because this requires minimal

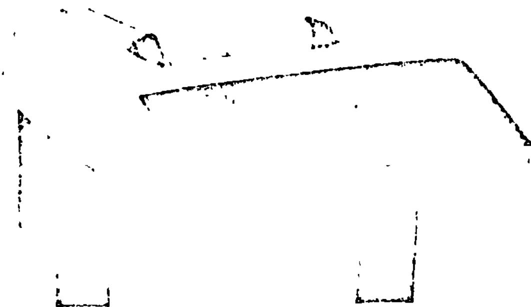


Fig. 3-3. The $2,500 type LA30 DECwriter employs seven solenoid-driven printing wires to form different character sets from a five by seven dot matrix. There are relatively few moving parts, and printing speed is 30 characters/sec. *(Digital Equipment Corporat[illegible])*

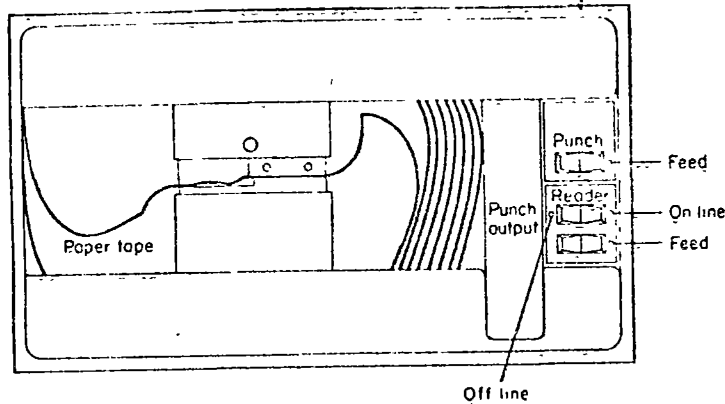


Fig. 3-4. Medium-speed reader punch for fanfold paper tape. Read at 300 characters/sec, punch at 50 characters/sec. (*Digital Equipment Corporation*)

peripheral equipment. The ASR-33 and ASR-35 teletypewriters, for instance, have built-in 10 character/sec tape punches and readers. These will do for loading (binary) program tapes and for infrequent problem preparation in applications requiring few such operations (e.g., special-purpose-system start-up, interpreter systems). For faster work, *one usually buys a 300 character/sec reader and a 50 character/sec punch* (Fig. 3-4), *both for fanfold paper tape, which does not require rewinding* (see also Table 3-3 and Sec. 3-9). *Faster reel-type readers* serve in special applications with long program or data tapes.

The operations needed to load words from paper tape into the computer memory will themselves constitute a computer program (**paper-tape loader**).

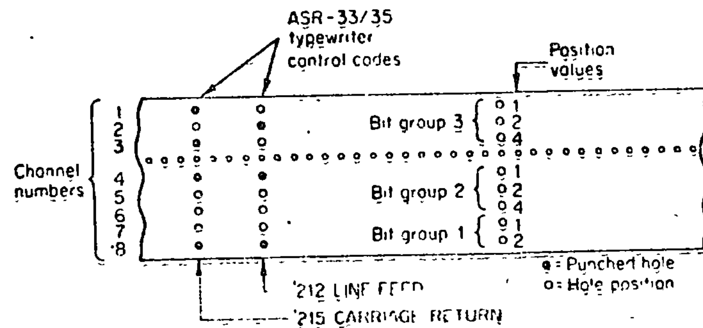


Fig. 3-5a. Paper tape with eight-channel ASCII code. This format is commonly employed for *source-program* tapes. *Binary object programs* are punched in different formats depending on the minicomputer word length. 12-bit words, for instance, can be coded into two paper-tape frames with channel 8 blank, while channel 7 indicates whether the word is meant to be a memory word or its *address*. The most economical object-tape codes contain only the starting address for recorded blocks of consecutive memory words, while others alternate words and their storage addresses. (*Honeywell Computer Control Division.*)

This is usually supplied on a short paper tape: The loader will read and load its own tape as soon as the first few instructions are in memory and is therefore called a **bootstrap loader**. The initial loading instructions can be loaded manually; some computers protect them from overwriting with a special front-panel switch. We set the program counter to the first loading-instruction address (usually printed on the loader tape) and press READ IN (or RUN, depending on the computer) to load the loader. As a desirable option, some minicomputers have the entire paper-tape-loader program permanently in read-only memory.

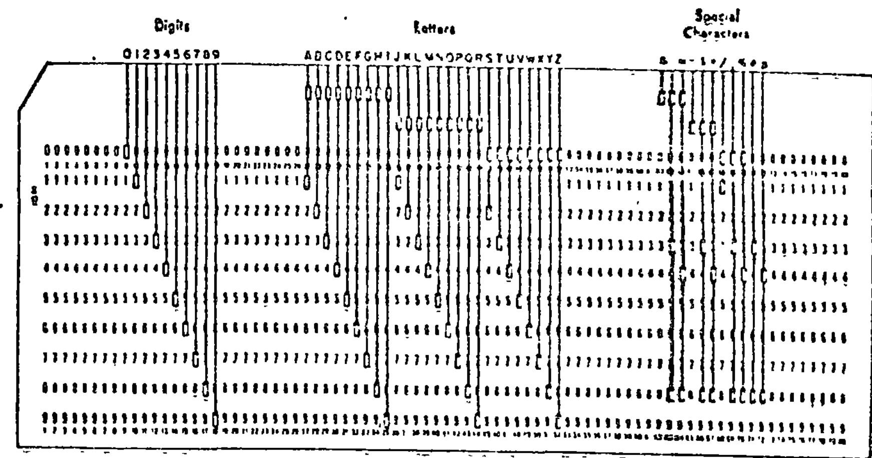


Fig. 3-5b. International Business Machines Corporation 80-column punched-card code.

Once the loader is in memory, we can load any program or data tape directly behind the loader program:

1. Place the tape in the reader (consult your manual).
2. Set the program counter to the first loading-instruction address.
3. Press READ IN, or RUN (consult your manual).

An executable program thus loaded can now be started as soon as we set the program counter to the appropriate starting address. Some programs include a jump to the starting address as the last instruction loaded, so we can simply press RUN and go.

MINICOMPUTATION: SOURCE-PROGRAM TRANSLATION

3-5. Programming and Program Translation. In Sec. 3-4, we did *not* discuss preparation of new programs but only the loading and running of

executable machine-language programs. Indeed, *with many special-purpose computer systems we may never have to prepare a program*, for the system may come with "canned" programs on tapes, kindly furnished by the computer manufacturer or by a software house, for a variety of jobs. All we do is supply the data inputs, load, and run

Less specialized applications require us to create our own programs. It would be a cruel job to write programs in binary or even octal machine language, so we type and/or punch a **source program** in a **programming** language admitting a restricted set of stylized English and mathematical statements. A **translator program** then employs the computer itself to read the source-program character code and to translate our source-program statements into machine-language instructions and data words of an executable **object program**.

Translators will be rather formidable system programs supplied (one hopes) by the computer manufacturer. There are three types of translators:

1. An **assembler** translates an **assembly language**, most of whose statements correspond to machine-language instructions on a one-to-one basis (e g , LOAD ACCUMULATOR WITH CONTENTS OF MEMORY LOCATION A, or LAC A)
2. A **compiler** translates a **compiler language**, which is closer to English-cum-mathematics and can include statements (e.g., formulas) which will each be translated into *many* machine-language instructions (e.g., FORTRAN, ALGOL, Sec. 3-7).

Each new computer type needs a new assembly language, and relatively simple mathematical and input/output operations can require substantial numbers of assembly-language instructions. But **assembly-language programming** (Chap 4) **can take the most efficient advantage of minicomputer hardware to save memory and time during execution.** By contrast, some minicomputer compilers generate slow-executing code because both compiler and object programs are compromised by the small amount of memory available

After an assembler or compiler is loaded, we load the source program (Fig. 3-6). The machine then produces either the object program (say on paper tape) in one pass, or the same or an intermediate tape is processed in a second pass (**two-pass assembler or compiler**). A third pass can produce a teletypewriter **listing** of both source program and machine code.

If we have made a mistake in our source-language syntax, exceeded the available memory storage, used too large numbers, etc., the translator program will notify us of this fact by stopping and printing an appropriate error message on the teletypewriter. At this point, we will have the pleasure of doing the job over. If the program works correctly, however, we will have a storable "binary" object tape, which can be used again and again wi :w data, without any need for translation.

Punching intermediate paper tapes consumes time (more time than tape reading with most tape reader punches) For this reason

- All modern two-pass *assemblers* are designed to read the original source tape a second time after the first pass, without any need for an intermediate tape.
- Some minicomputers (e g , Raytheon 700 series) have a "conversational" FORTRAN compiler which can compile (somewhat restricted) FORTRAN directly into the memory, object-tape punching is optional

A number of minicomputer manufacturers also furnish assemblers and or compilers which can be run on a large batch-processing digital computer. The resulting minicomputer object programs will still be on paper tape, or possibly on cards.

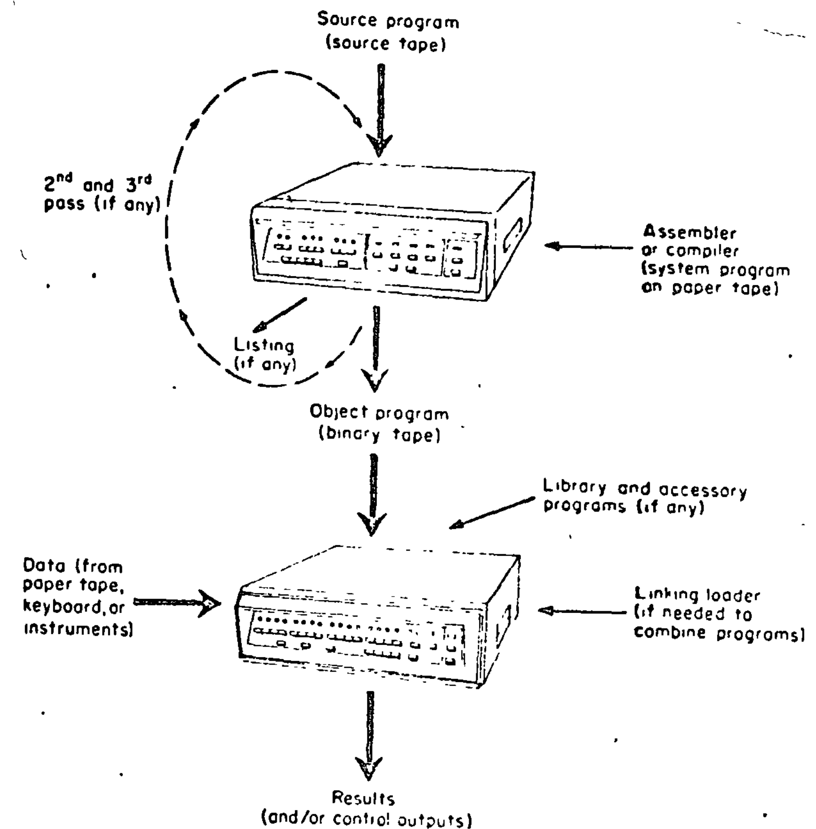


Fig. 3-6. Program translation and loading with a paper tape system

Assemblers and compilers generate complete object programs for execution. Our third translation scheme works differently:

3. An **interpreter** translates one source-language statement at a time, executes the resulting machine instruction or instructions at once, translates the next statement, etc.

Interpreter translation is inefficient for "production programs" which are to be executed many times, since each execution will be slowed by translation. This is not objectionable in on-line conversational computing, where interpreter systems translate compiler-type source languages like BASIC and FOCAL (Secs. 3-8 and 7-3). Interpreters can also implement step-by-step emulation of computer instructions by the instruction set of a different computer.

3-6. Loading and Combining (Linking) Binary Object Programs. An independent binary object program loaded into core with our paper-tape loader should be ready to execute. *Data* for such a program may have been loaded together with the program, or the program contains instructions to get its data from peripheral devices, e.g., from typed input, from another paper tape placed in the paper-tape reader, or from instruments such as analog-to-digital converters. Program output will be obtained on the teleprinter, display, tape punch, etc., as specified by the program itself. Very often, though, *we should like to combine a binary object program with other such programs.* These may be user programs, perhaps modules of a larger program, or library subroutines supplied by the computer manufacturer (e.g., floating-point arithmetic routines, sine/cosine generators, input/output routines). With a paper-tape operating system, all these programs and subroutine libraries will be on various pieces of paper tape; we would like to load them for combined execution. This will practically always impose two requirements:

1. We must relocate binary programs so that they can be loaded into successive core-memory areas. This will mean *changing both instruction addresses and memory-reference addresses.*
2. Since the combined programs will refer to one another (by supplying data and/or through jump instructions), we must find all such external references and provide them with the correct memory addresses.

To satisfy the first requirement, the object programs to be relocated must have been prepared by an assembler or compiler which generates relocatable code. That is, all memory references to addresses needing relocation are either specially marked -- say with an extra word -- or they are relative to the current program-counter reading; see also Sec. 4-18. To satisfy the second requirement, each program segment must list all its external references according to a specified convention.

To combine program segments satisfying these requirements, we first load a new system program called a linking loader and then the various object tapes. The linking loader will note the final address of each program segment, relocate the succeeding program, and supply the necessary linkage references. The combined program will be left in core ready for execution. Options,

usually selected by front-panel register or sense switches, will run the combined program as soon as it is loaded ("load and go") and/or punch a binary tape for the combined object program. The linking loader will also type out *error messages* if it is prevented from doing its job by user errors such as faulty or missing external references; consult your minicomputer manual.

3-7. FORTRAN Computations and Related Topics. Many engineers and scientists will be familiar with FORTRAN programming (or, especially if they live in Europe, with ALGOL; minicomputer operation will be quite similar). The principal features of the FORTRAN language are outlined in Table 3-1. Note that minicomputer compilers may implement only a subset or restricted version of the FORTRAN language available with larger machines; **consult your minicomputer manual.** But many minicomputers have remarkably comprehensive versions of FORTRAN. Minicomputer FORTRAN compilers are usually designed to minimize the core storage required for the compiler and for the compiled program at some expense in execution speed.

Normally, you will need a linking loader (Sec. 3-6) to load a compiled FORTRAN program or program segments with the computer manufacturer's "math library" of floating-point-arithmetic and function-generator routines; the linking loader will load only those routines actually called by your program. I/O routines are usually supplied by the compiler or by the compiler together with an executive program (Sec. 3-13).

Other compiler languages available with minicomputers (ALGOL, subsets of COBOL) are dealt with quite similarly.

3-8. Conversational Computing with Interpreter Programs. **Conversational interpreter systems (Sec. 3-5) are especially easy to use with paper-tape-loaded minicomputers: Only the interpreter tape must be loaded, for the program itself will be supplied by the user on the teletypewriter or on a cathode-ray-tube/keyboard terminal.**

The most popular conversational-interpreter language is BASIC, which is widely used in time-sharing systems. BASIC interpreters are available for many minicomputers. The Digital Equipment Corporation has developed another rather similar system called FOCAL (but DEC machines all come with BASIC interpreters as well). Both BASIC and FOCAL can be learned rapidly; both allow you to employ a minicomputer as a very versatile desk calculator and for real stored-program computation with loops, subroutines, etc. Such interpreters permit floating-point computation but usually only with six-decimal-digit precision (this is less than that available from most desk calculators). BASIC and FOCAL interpreters supply their own utility routines (e.g., for trigonometric functions). BASIC, unlike FORTRAN, permits operations with matrices. Both BASIC and

TABLE 3-1 Minicomputer FORTRAN Check List.

In FORTRAN.

1. Specification statements *define the properties of, and allocate storage to*, named variables, functions, and arrays.
2. Arithmetic statements define computing operations which assign the value of an expression to a variable, e g ,

```
VAR - B + 01
```

3. Control statements determine the sequence of operations in a program,* e g ,

```
GO TO 17
```

4. Input/output statements specify input/output operations.

FORTRAN is "portable", i e the language is to a large extent independent of the processor
and compiler used Most minicomputer FORTRAN systems are subsets of USASI FORTRAN
IV. But different minicomputers (and even the same minicomputers with different amounts of
core) implement more or less complete FORTRAN systems. It will be necessary to check
precisely on your particular minicomputer.

1. Are logical and/or complex type quantities admissible?
2. Representation of Real Constants: How many digits are accommodated? Are all possible formats, for example,

```
5E-02    0 05
0 5E 01  0 5E-1
5 0E 02
```

admissible? The same questions should be answered for double-precision quantities
3 Logical constants are .TRUE and .FALSE If no logical variables and operations are
available, you can still employ the arithmetic IF statement.
4 Check on specification statements for declaring variables as real, logical, etc In general,
integer names begin with I J K, L, M, or N
5. Check on the extent to which expressions can be used as subscripts for subscripted variables.
6. Relational Operators: .LT., .LE., .EQ., .NE., .GT., .GE.. Are all admissible? Are logical expressions admitted?
7. Check on the availability of each of the following control statements:
   (*a*) Assigned GO TO `ASSIGN 18 TO K` `GO TO K, (3 4, 18 21)`

   (*b*) Computed GO TO `I - 2` `GO TO (3 18 21) I`

The examples in (*a*) and (*b*) are equivalent to the *unconditional* GO TO 18

(*c*) Arithmetic IF IF (arithmetic expression) n_1, n_2, n_3
Program goes to statement number n_1, n_2, or n_3 if the specified expression is, respectively,
less than, equal to, or greater than zero

(*d*) Logical IF IF (logical expression) (statement)
The specified statement which must be executable and neither a DO nor a logical IF
statement, is executed if the logical expression is true; otherwise, control transfers to the
following statement The logical expression might be a hardware sense line or switch
output.

```
IF (SENSE SWITCH 3)     (statement)
```

(*e*) DO n INDEX = m_1, m_2, m_3
n is a statement number, m_1 and m_2 are the *initial* and *final values* of the integer INDEX,
and m_3 is the *increment* of INDEX If m_3 = 1, one may write

```
DO n INDEX = m1, m2
```

* [illegible] Digital Equipment Corporation, Maynard, Mass., 1968

TABLE 3-1. Minicomputer FORTRAN Check List (*Continued*).

(*f*) CONTINUE STOP PAUSE n
PAUSE END STOP n

8. Check on the interpretation of READ and WRITE statements and device numbers, minicomputer peripheral devices may differ from those used with batch-processed FORTRAN systems on large digital computers
9 Check on the library subroutines and special functions available with your FORTRAN system
10 Can FORTRAN programs be linked to assembly-language programs (Secs. 4-20 and 5-28)?
11. Can FORTRAN be used for interrupt servicing (Secs 4-16 and 5-16)?

FOCAL permit graphic output from cathode-ray-tube displays and digital plotters. Table 3-2 outlines the main features of the BASIC language; many good texts on BASIC programming are available (Refs. 8 to 11).

Some of the most useful minicomputer applications employ BASIC or FOCAL interpreters extended to incorporate input/output commands for operating measuring instruments, test-voltage sources, and process-control equipment (Secs. 7-3 and 7-5).

TABLE 3-2. A Quick Reference Guide to BASIC.

This table was prepared by the software staff of the Hewlett-Packard Corporation for their 2000B (time-shared) version of BASIC (Refs 9 and 10) The complete BASIC system illustrated here is more than a simple algebraic-interpreter language, it permits array and matrix manipulation, limited string manipulation, some editing, file manipulation, and chaining (overlays) of program segments. Hewlett-Packard BASIC has also been extended to operate with *displays* and *instruments* (Sec. 7-3*b*). Nevertheless, completely untrained operators can use BASIC as a very simple "conversational calculator" by typing statements like

```
LET V1 - 7 5
LET B  - V1 + 2 1
```

as commands, i.e , *without statement numbers*, and to obtain answers by typing, say

```
PRINT B, V1
```

OPERATORS

Operators are used in the statements of a program

Sample Statement	*Purpose/Meaning/Type*
100 A = B = C = 0	Assignment operator; assigns a value to a variable.
110 LET A = 0	May also be used without LET.
120 Z = X↑2	Exponentiate (as in X^2).
130 LET C5 = (A*B)*N2	Multiply.
140 IF T5/4 = 3 THEN 200	Divide.
150 LET P = R1 + 10	Add
160 X3 = R3 -- P	Subtract
NOTE: The numeric values used in logical evaluation are: "true" = any nonzero number; "false" = 0	
170 IF D = E THEN 600	*Expression* "equals" *expression*.

TABLE 3-2. A Quick Reference Guide to BASIC (*Continued*).

Example	*Purpose*
180 IF (D + E) ≠ (2•D) THEN 710	*Expression* "does not equal" *expression*
180 IF (D + E) < > (2•D) THEN 700	*Expression* "does not equal" *expression*
190 IF X > 10 THEN 620	*Expression* "is greater than" *expression*
200 IF R8 < P7 THEN 640	*Expression* "is less than" *expression*
210 IF R8 > = P7 THEN 810	*Expression* "is greater than or equal to" *expression*.
220 IF X2 < = 10 THEN 650	*Expression* "is less than or equal to" *expression*
230 IF G2 AND H5 THEN 900	*Expression* 1 AND *expression* 2 must both be "true" for composite to be "true."
240 IF G2 OR H5 THEN 910	If either *expression* 1 OR *expression* 2 is "true," composite is "true."
250 IF NOT G5 THEN 950	Total expression NOT G5 is "true" when *expression* G5 is "false."
260 LET B = A2 MAX C3	Evaluates for the larger of the two expressions
270 LET B1 = A7 MIN A9	Evaluates for the smaller of the two expressions.

STATEMENTS

Programs consist of numbered statements. The statements are ordered by number.

Example	*Purpose*
300 CHAIN PROG 310 CHAIN SLIBR	GETs and RUNs the program specified. The current program is destroyed, except for COMmon variables.
320 COM A,B1,C(20),C$(72)	Declares variables to be in COMmon, they can then be accessed by other programs. Must be lowest numbered statements.
360 DATA 99, 106.7, "HI!"	Specifies data, read from left to right
310 DIM A(72)	Defines maximum size of a string or matrix
400 END	Terminates the program; must be last statement in a program
375 ENTER #T 380 ENTER A,B,C 390 ENTER T,A,B,C$	Fills the first variable #T with the user terminal number and/or allows the user a specified number of seconds to reply (A), returns the actual response time B, and returns the value entered C,C$. On time out, the response time is set to −256. On illegal input type, the response time is negated.
400 FOR J = 1 TO N STEP 3 500 NEXT J	Executes the statements between FOR and NEXT a specified number of times, incrementing the variable by a STEP number (or by 1 if STEP is not given)
330 GO TO 900	Transfers control (jumps) to specified statement number.
412 GO TO *N* OF 100,10,20	Transfers control to the *N*th statement of the statements listed after "OF."
420 GOSUB 800	Begins executing the subroutine at specified statement (See RETURN.)
415 GOSUB *N* OF 100,10,20	Begins executing the subroutine *N* of the subroutines listed after "OF." (See RETURN.)
340 IF A # 10 THEN 350	Logical test; transfers control to statement number if "true."
390 INPUT X$,Y2,B4	Allows data to be entered from terminal while a program is running.
300 LET A = B = C = 0 310 A1 = 6.35	Assigns a value to a variable; LET is optional.
360 READ A,B,C	Reads information from DATA statement
350 READ #3,A	See "Files."

TABLE 3-2. A Quick Reference Guide to BASIC (*Continued*).

Example	*Purpose*
320 REM--ANY TEXT**!!	Inserts nonexecutable remarks in a program
356 PRINT A,B,C$	Prints the specified values, 5 fields per line when commas are used as separators, 12 when semicolons are used
358 PRINT	Causes the teleprinter to advance one line
395 PRINT #3,A	See "Files."
38Ø RESTORE	Permits rereading data without rerunning the program
385 RESTORE *N*	Permits data to be reread, beginning in statement *N*.
85Ø RETURN	Subroutine exit, transfers control to the statement following the matching GOSUB.
41Ø STOP	Terminates the program, may be used anywhere in program

STRINGS

1. A string is 1 to 72 teleprinter characters enclosed in quotes; it may be assigned to a string variable (an A to Z letter followed by a $).
2. Each string variable used in a program must be dimensioned (with a DIM or COM statement) if it has a length of more than one character. The DIM sets the physical or maximum length of a string.
3. Substrings are described by subscripted string variables. For example, if A$ = "ABCDEF," then A$(2,2) = B, and A$(1,4) = "ABCD."
4. The LEN function returns the current string length, for example: 100 PRINT LEN (A$). This length is the logical length.

Example	*Purpose*
1Ø DIM A$(27)	Declares the maximum string length in characters
2Ø LET A$ = "**TEXT I"	Assigns the character string in quotes to a string variable.
3Ø PRINT LEN (B$)	Gives the current length of the specified string
1Ø5 IF A$ = C$ THEN 6ØØ 11Ø IF B$ # X$ THEN 65Ø 115 IF N$(2,2) > B$(3,3) THEN 1Ø 12Ø IF N$ < B$ THEN 999 125 IF P$(5,8) > = Y$(4,7) THEN 1Ø 13Ø IF X$ < = Z$ THEN 999	String operators. They allow comparison of strings, and substrings, and transfer to a specified statement. Comparison is made in ASCII codes, character by character, left to right until a difference is found. If the strings are of unequal length, the shorter string is considered smaller if it is identical to the initial substring of the longer
2Ø5 INPUT N$	Accepts as many characters as the string can hold (followed by a *return*). The characters need not be in quotation marks if only one string is input
21Ø INPUT N$,X$,Y$	Inputs the specified strings, input must be in quotes, separated by commas
215 READ P$	Reads a string from a DATA statement, string must be enclosed in quotes
22Ø READ #5; A$,B$	Reads strings from the specified file.
31Ø PRINT #2; A$,C$	Prints strings on a file.

FUNCTIONS

Functions return a numeric result, they may be used as expressions or parts of expressions. PRINT is used for examples only, other statement types may be used

TABLE 3-2. A Quick Reference Guide to BASIC (*Continued*).

Example	*Purpose*
3ØØ DEF FNA (X)=(M*X)+B	Allows the programmer to define functions, the function label A must be a letter from A to Z
31Ø PRINT ABS (X)	Gives the absolute value of the expression X
32Ø PRINT EXP (X)	Gives the constant e raised to the power of the expression value X, in this example, e ↑ X
33Ø PRINT INT (X)	Gives the largest integer ≤ the expression X
34Ø PRINT LOG (X)	Gives the natural logarithm of an expression; expression must have a positive value.
35Ø PRINT RND (X)	Generates a random number greater than or equal to Ø and less than 1, the argument X may have any value.
36Ø PRINT SQR (X)	Gives the square root of the expression X, expression must have a positive value
37Ø PRINT SIN (X)	Gives the sine of the expression X, X is real and in radians
38Ø PRINT COS (X)	Gives the cosine of the expression X, X is real and in radians
39Ø PRINT TAN (X)	Gives the tangent of the expression X; X is real and in radians
4ØØ PRINT ATN (X)	Gives the arctangent of the expression X; X is real, result is in radians
41Ø PRINT LEN (A$)	Gives the current length of a string A$, i.e., number of characters.
42Ø PRINT SGN (X)	Gives: 1 if X>Ø, Ø if X=Ø, −1 if X<Ø
43Ø PRINT TAB (X),A	Tabs to the specified position X, then prints the specified value A. Used for plotting
44Ø PRINT TIM(X)	Gives current minute (X=Ø), hour (X=1), day (X=2), or year of century (X=3)
45Ø PRINT TYP (X).	If argument X is negative, gives the type of data in a file as: 1=number, 2=string, 3="end of file," 4="end of record", or if argument X is positive, gives the type of data in a file as: 1 = number, 2 = string, 3 = "end of file." (For sequential access to files—skips over "end of records.") If argument X = Ø, gives the type of data in a DATA statement as 1 = number, 2 = string, 3 = "out of data."

MATRICES

Absolute maximum matrix size is 2,500 elements. Matrix variables must be a single letter from A to Z.

Sample Statement	*Purpose*
1Ø DIM A (1Ø, 2Ø)	Allocates space for a matrix of the specified dimensions.
15 MAT X = IDN (M,M)	Establishes an identity matrix (with all 1s down the diagonal). A new working size (M,M) can be specified.
2Ø MAT B = ZER	Sets all elements of the specified matrix equal to Ø
25 MAT D = ZER (M,N)	A new working size (M,N) may be specified after ZER
3Ø MAT C = CON	Sets all elements of the specified matrix equal to 1
35 MAT E = CON (M,N)	A new working size (M,N) may be specified after CON
4Ø INPUT A(5,5)	Allows input from the teleprinter of a single specified matrix element
45 MAT INPUT A(4,7)	Allows input of an entire matrix from the teleprinter. a new working size can be specified.
5Ø MAT PRINT A;	Prints the specified matrix on the teleprinter.

TABLE 3-2. A Quick Reference Guide to BASIC (*Continued*).

Sample Statement	*Purpose*
55 PRINT A(X,Y)	Prints the specified element of a matrix on the teleprinter, element specifications X and Y can be any expression
6Ø PRINT #2, A(1,5)	Prints matrix element on the specified file
65 MAT PRINT #2,3,A	Prints matrix on a specified file and record.
7Ø MAT READ A	Reads matrix from DATA statements.
75 MAT READ A(5,5)	Reads matrix of specified size from DATA statements
8Ø READ A(X,Y)	Reads the specified matrix element from a DATA statement
85 MAT READ #3; A(I,J)	Reads matrix from the specified file; new working size can be specified.
9Ø MAT READ #3,5; A	Reads matrix from the specified record of a file.
1ØØ MAT C = A + B	Matrix addition, A and B must be the same size
11Ø MAT C = A − B	Matrix subtraction, A, B, and C must be the same size
12Ø MAT C = A * B	Matrix multiplication, number of columns in A must equal number of rows in B
13Ø MAT A = B	Establishes equality of two matrices, assigns values of B to A
14Ø MAT B = TRN (A)	Transposes an *m* by *n* matrix to an *n* by *m* matrix.
15Ø MAT C = INV (B)	Inverts a square matrix into a square matrix of the same size, matrix can be inverted into itself

FILES

A FILE = a named storage area of from 1 to 128 records. Maximum size varies with systems
A RECORD = 64 words of memory
A NUMBER = a data item using 2 words of memory
A STRING = a data item using about ½ word of memory per character.

Example	*Purpose*
OPE-MYFILE,8Ø	Opens a file with a specified name and size.
KIL-MYFILE	Removes the specified file
1Ø FILES BUG,GANG	Declares which files will be used in a program. Up to 4 FILES statements with a total of 16 files per program. Files must be OPENed first
2Ø PRINT#N A,B	Prints the specified values A,B on a specified file at the current position. Files are numbered from 1 as they appear in the FILES statements
3Ø PRINT #X,Y A,B,C$	Prints the specified values on a specified record Y of a file X
4Ø PRINT 3,5	Erases the specified record of a file
7Ø READ#1 A,B2	Reads the next values of a specified file into the specified variables
8Ø READ#2,3 A,B	Reads values from the beginning of a specified record of a file into specified variables.
185 READ#3,5	Resets the pointer for a file to a specified record
19Ø IF END#N THEN 8ØØ	Transfers control to a specified statement if an end-of-file occurs on a specified file.

COMMANDS

Commands are executed immediately; they do not have statement numbers

Example	*Purpose*
APP-PROG1	Appends the named program to the current program.
BYE	Logs the user off his terminal.

TABLE 3-2. A Quick Reference Guide to BASIC (*Continued*).

Example	*Purpose*
CAT	Lists the names and lengths of user library programs
CSA	Saves the current program in semicompiled form
DEL-100	Deletes all statements after and including the specified ones.
DEL-100,200	Deletes all statements between and including the specified ones.
ECH-OFF	Permits use of half-duplex coupler
ECH-ON	Returns user to full-duplex mode
GET-SAMPLE	Retrieves the program from the user's library and makes it the current program
GET-SPROG	Retrieves the program from the system library
HEL-D007,B*G*	Logs the user onto his terminal. User must give I.D. code and password
KEY	Returns terminal to keyboard entry after TAPE command
KIL-SAMPLE	Deletes the specified program from the user's library (does not modify the current program)
LEN	Lists the current program length in words
LIB	Lists the names and lengths of system library programs
LIS LIS-150 LIS-100,200	Lists the current program, optionally starting at a specified statement number and stopping at a specified statement.
NAM-SAMPLE	Assigns the name to the current program, name may consist of one to six printing characters
PUN PUN-50 PUN-100,200	Punches the current program to paper tape, optionally starting at a specified statement number and optionally stopping at a specified statement
REN REN-50 REN-50,20	Renumbers the current program from 10 (optionally from a specified statement number) in multiples of 10 (optionally in multiples of a specified number)
RUN RUN-50	Starts executing the current program, optionally starting at a specified statement number
SAV	Saves the current program in the user's library
SCR	Erases the current program (but not the program name).
TAP	Informs system that input will now be from paper tape
TIM	Lists terminal and account time

EXAMPLE A Complete BASIC Program

Program
```
10 LET X=1
20 FOR Y=1.1 TO 3.1 STEP .5
30 LET H=SQR(X↑2 + Y↑2)
40 PRINT "WHEN (X,Y) =" X,Y, "THE HYPOTENUSE IS" H
50 NEXT Y
60 END
RUN
```
Results
```
WHEN (X,Y) =   1   1.1   THE HYPOTENUSE IS 1.48661
WHEN (X,Y) =   1   1.6   THE HYPOTENUSE IS 1.8868
WHEN (X,Y) =   1   2.1   THE HYPOTENUSE IS 2.32594
WHEN (X,Y) =   1   2.6   THE HYPOTENUSE IS 2.78568
WHEN (X,Y) =   1   3.1   THE HYPOTENUSE IS 3.2573
DONE
```


CONVENIENT VERSUS INCONVENIENT OPERATING SYSTEMS

3-9. Introduction. Paper-tape operation, as described in Secs. 3-4 to 3-6, is a reasonable way to operate minicomputer systems dedicated to a single task or to only a few different tasks. But for general-purpose computation requiring frequent creation, translation, loading, correction, and modification of different programs, paper-tape operation presents an untenable situation, even with high-speed paper-tape readers and punches. Loading and compilation with an ASR teletypewriter *alone* can take hours even for relatively small FORTRAN programs. We will need a system which can quickly read, store, and retrieve system programs (loaders, assembler, compiler), source programs, and object programs without so many repeated manual loading operations. Above all, we should like to load, combine, and execute programs automatically or on typed commands. This is made possible by an executive program (sometimes called a monitor system) in conjunction with magnetic disk, drum, or tape storage of system and library programs, user files, and intermediate translator outputs. In addition, it will be a good idea to supplement the failure-prone, slow, and noisy teletypewriter with an inexpensive cathode-ray-tube/keyboard terminal.

3-10. Magnetic Disk, Drum, and Tape Storage. (a) **Storage Requirements and Operations.** Minicomputer system programs, such as assemblers and compilers, typically require several thousand words each. Additional thousands of words will be required for library programs (frequently used arithmetic, function-generator, and input/output routines), stored user programs, and user programs in the intermediate stages of a translation process. For general-purpose computation on minicomputers with 8K to 32K of core memory we will, moreover, "chain" successive segments of longer user programs stored on a disk or tape: We load and execute the first program segment, keep some intermediate results in core, load and execute a second program segment, etc.; such program segments are known as successive *core overlays*. Many applications also involve creation of permanent or intermediate text or data files with many thousands of words. Altogether, general-purpose minicomputation will require between 30K and several million words of mass storage, which should be accessible without manual loading operations. Since mass *core* storage is too expensive for minicomputers (of the order of $0.30 per 16-bit word), disk, drum, and/or magnetic-tape storage is used.

Magnetic mass-storage systems are compared in Table 3-3. Fixed-head rotating disks and drums have the highest *data-transfer rates* and, since they can access any storage location within one revolution, also the shortest *access times*. Disks are, therefore, best for storing system programs and intermediate output. Small disks are, moreover, inexpensive (about $6,000

TABLE 3-3. Comparison of Minicomputer Input/Output Storage Media and Peripherals.

	PAPER TAPE		MAGNETIC TAPE Synchronous (continuous) operation			FIXED-HEAD DISKS AND DRUMS
	ASR-33 teletypewriter	Medium-speed reader/punch	Cassette/Cartridge	DECtape	Standard (IBM-compatible) tape	
8-bit characters/sec	10	300 READ 50 PUNCH	500–5,000	10,000	6,000–60,000	40,000–360,000
16-bit words/sec	5	150 READ 25 PUNCH	250–2,500	7,500 (12-bit words) 5,000 (18-bit words)	3,000–30,000	20,000–180,000
Time to read 1,000 16-bit words (object programs)	200 sec	7 sec	0.4–4 sec	133 msec (12-bit words) 200 msec (18-bit words)	30–300 msec	6–50 msec
Time to read 1,000 typical 20-character lines (source programs for assembly or compilation)	2,000 sec (over 30 min)	70 sec	4–40 sec	2 sec	0.3–3 sec	60–500 msec
Access time	—	—	12–150 sec	up to 100 sec	up to 240 sec	8–17 msec
Total storage (16-bit words in one unit)	—	—	50K–250K	100K 18-bit words or 150K 12-bit words	200K–2M	30K–600K
Typical price (combined input/output unit and interface)	$300 more than KSR-33	$3,000	$300–$3,000	$3,400–$9,700*	$5,000–$13,000	$6,000–$30,000

* One $7,400 interface can serve up to eight transports

for 30K words), but magnetic tape on removable reels is cheaper for larger amounts of storage. Small and larger disks are often combined with magnetic tape so that different programs or data can be loaded from removable tape reels. There are also disk systems with removable disks or disk cartridges and combinations of fixed and removable disks.

Disk, drum, and magnetic-tape interface hardware and operations are described in Table 5-2. Blocks of computer words are almost always transferred directly from or to the computer core (or semiconductor) memory. Input/output programs are fairly involved, but they are usually supplied by the computer manufacturers. The user/programmer simply sees buffer areas in the computer memory, i.e., blocks of memory locations whose contents will be transferred to or from mass storage. Each buffer is identified by its *starting address* and *size (word count)*; *header words* in each storage block may identify the block by a name and specify word count and addresses of succeeding or preceding blocks. To safeguard the large

amounts of information handled, mass-storage systems usually employ parity checks (Sec. 1-4c) on each partial word (byte) transferred *and* an additional parity check for each multiword block.

(b) Disk and Drum Systems (see Fig. 3-7). Disk and drum systems record data words in *serial* form; i.e., successive bits are recorded or read as

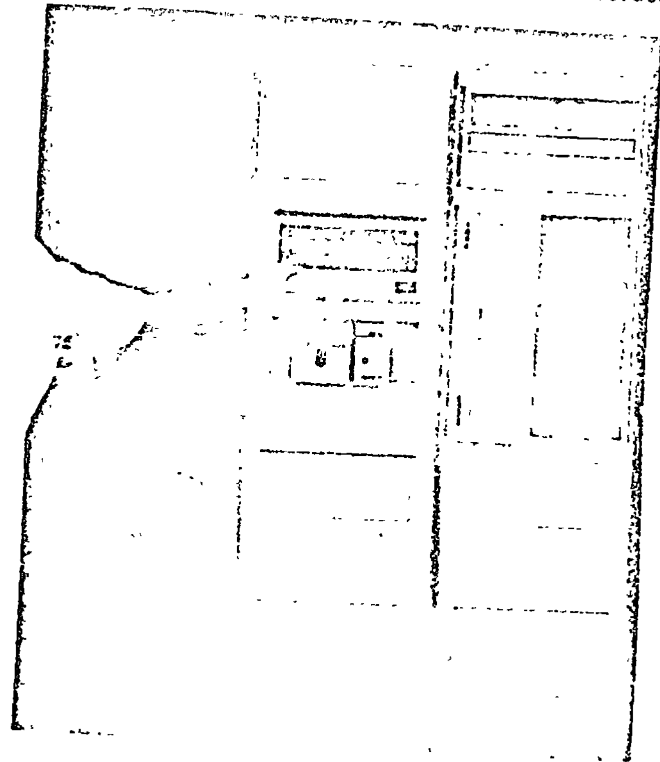


Fig. 3-7. A complete minicomputer system with a 500 character sec paper-tape reader, disk memory (top right), and a 16 700 character sec magnetic-tape transport. A card reader is seen on the table at left (*Hewlett-Packard* 2100A computer.)

the magnetic film passes under a record read head (see also Sec. 1-3) Magnetic drums, and most small disks, have *fixed heads* for individual data tracks. *Moving-head disks* require fewer heads, but they must be positioned quickly and accurately by expensive mechanisms. Storage addresses on disks and drums refer to the track and to timing marks on special timing tracks. The program usually establishes a *directory table* which produces the specific timing-track readings corresponding to the starting words of named blocks.

(c) Magnetic-tape Systems. Unlike disks and drums, magnetic-tape systems store several bits of a partial word (byte) *in parallel* across the tape. Parity can be checked across the tape (*transverse parity*) and for blocks of

data along the tape (*longitudinal parity*). Formatted tape employs a prerecorded timing track or tracks to find blocks of data by reference to a directory table, just like a disk. Unformatted tape has no timing track, and the header word of a desired block must be found by scanning the tape. Incremental-tape systems start and stop the tape for individual words, but most magnetic-tape systems stop only at *record gaps* between blocks of words. Start and stop times are between 1 and 20 msec. The better tape transports can read or write backward as well as forward. The beginning and end of each tape are usually marked by reflective markers sensed by tape-transport hardware.

Blocks of data on tape can be of fixed or variable length. To update a block of data on tape will require *two* tape transports unless one is sure that the new data will fit the old block.

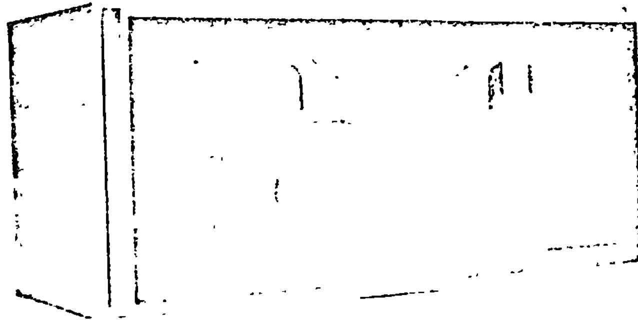


Fig. 3-8. 4-in reels of this type CO-600A LINC tape system hold about 100,000 16- or 18-bit words. Prerecorded timing tracks address blocks of data. Transfer rate is 4,200 words/sec at 60 in/sec with 30 sec maximum access time. A thin Mylar layer over the tape oxide protects both oxide and heads. Phase recording on nonadjacent duplicate data and timing tracks and capstan-less simplicity make such systems very reliable. (*Computer Operations, Inc., Beltsville, Maryland.* Digital Equipment Corporation DEC tape is similar.)

Small capstan-less formatted-tape units like that shown in Fig. 3-8 (see also Table 3-3) employ duplicate data tracks for redundancy checks, have very handy small reels, and are reliable and inexpensive, an excellent choice for minicomputers. Standard unformatted-tape (IBM-compatible) systems are somewhat faster and can store more data, but they are also more complicated and expensive; we would use them with a minicomputer only *if tapes must be transferred from or to a larger digital computer* (Fig. 3-7).

Unformatted-tape systems record either 9 or 7 bits across the tape (1 bit will be a parity bit). Figure 3-9 shows a typical arrangement of data blocks, record gaps, and longitudinal check characters.

(d) Tape Cassette/Cartridge Systems. Tape cassette/cartridge units (Fig. 3-10) are slower than other tape systems but are so convenient to mount and change that they should be an excellent replacement for punched paper tape

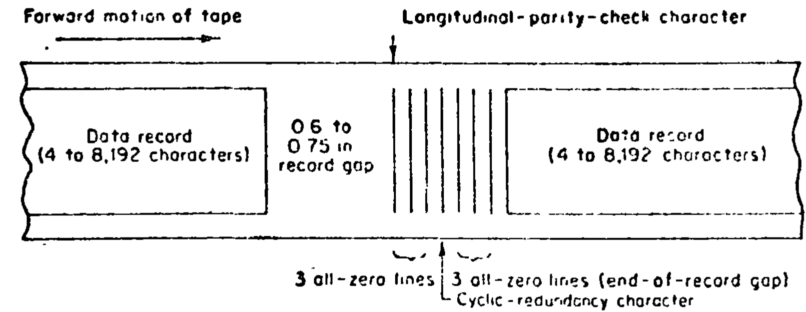


Fig. 3-9. Arrangement of data on nine-track tape. Tape *records*, corresponding to core buffers accommodating some reasonable amount of information (teletypewriter line), have between 4 and 8,192 eight-bit *characters*, each character comes with a ninth bit (transverse-parity bit). Each record is terminated by three all-zero lines (*end-of-record* gap), a cyclic redundancy character, three more zero lines, and a longitudinal-parity-check character. This is followed by a *record* gap at least 0.6 in long. A *file* is a group of records terminated by a 3-in gap followed by a *file mark* comprising an *end-of-file character* and a longitudinal-parity-check character.

in many computer systems. Since the small reels have limited storage, and also to speed access, one usually employs multiple cassette drives. Both formatted and unformatted tape are used, and a wide variety of systems exists (Table 3-3). Access times are a little slow for serious operating systems. Cassettes might be replaced by simple "flexible" disks.

3-11. Keyboard Operating Systems with Mass Storage. With a mass-storage system (we hope it is a disk), we are ready for respectable general-purpose computation. We initially use paper tape or magnetic tape to load

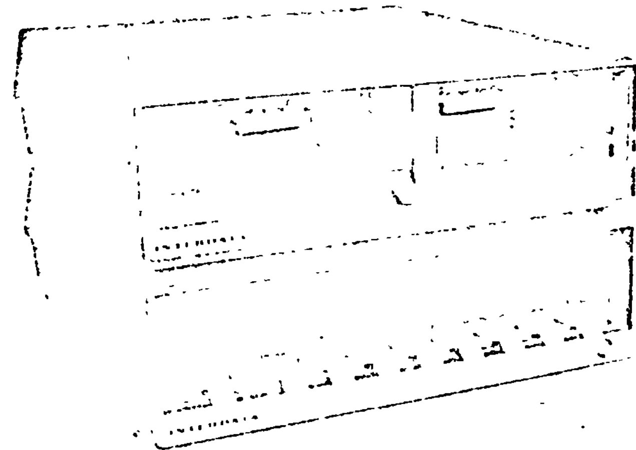


Fig. 3-10. Minicomputer with dual tape-cassette units. Each unit stores 250,000 eight-bit bytes on 300 ft of formatted tape at an 800 bit/in density. Transfer rate is 300 bytes/sec at 3 in/sec tape speed; fast-rewind speed is 90 in/sec. (*Interdata, Inc.*, Model 1 computer.)

an executive program (sometimes called a **monitor system**; different manufacturers employ different terms). This is a system program which announces its presence by typing EXEC or MONITOR on a teletypewriter or display and then waits for *keyboard commands*. The executive program responds to an interrupt (Sec. 5-9) when a key is struck, reads keyboard commands, and branches to an appropriate subroutine for loading and/or executing programs stored in core or in mass storage. To save core storage, only the portion of the executive needed to recognize keyboard commands is permanently stored in core (**resident executive, resident monitor**) and serves to call the various loading and service subroutines of the executive program from the system disk or tape. The minicomputer user does not need to study the detailed operation of the executive program; he need only consult the computer manufacturer's manual for the available list of keyboard commands and options

Suppose that we have an edited FORTRAN source program stored on the system disk under the file name MYFILE. **We want to compile this program, combine it with a binary object program available on paper tape, load, and execute.** We will assume that the necessary system programs (compiler and linking loader) are available on our system disk; with small disk systems, assemblers might have to be loaded from paper tape or magnetic tape as needed. **The following conversational-programming sequence will exhibit typical features of minicomputer keyboard-executive operation.** Please note that specific features, codes, and rules will differ from system to system; consult your minicomputer manual.

1. When the executive is loaded and ready, it types out

   EXEC

   We now type

   FORTRAN

   to call the *FORTRAN compiler* from the system disk.
2. The executive loads the compiler and responds with

   FORTRAN LOADED OUTPUT?

   We want to give the compiler output the file name CPUT for reference and store it on the system disk for loading and execution; we type

   DISK 1/CPUT

We could also have saved CPUT on a library tape (Sec. 3-12) by typing, say, TAPE 3 CPUT. Paper-tape output could also be specified, but is usually a compiler option (see below)

3. The system next asks

   INPUT?

   We specify the compiler input by typing

   DISK 1/MYFILE

4. The system now asks us to specify *compiler options:*

   FORTRAN-OPTIONS?

   We type an option code (if any), and then a carriage return to start compilation.

Compiler options could include *preparation of a binary object-program paper tape*, *printing a symbol table*, and/or *printing a listing*. The compiler will use the system disk for quick storage and retrieval of intermediate output.

5. The system completes compilation and saves the output (CPUT) or, if compilation is unsuccessful, prints *error diagnostics*. Return to executive control is announced with

   EXEC

   We call the *linking loader* by typing

   LOADER

6. The system loads the linking loader and asks for input:

   LOADER-INPUT?

   We specify

   DISK 1/CPUT; READER

   and place our second object program (which was on paper tape) in the paper-tape reader, pushing an appropriate button to clear the reader interface (Table 5-2).
7. The system asks

   LOADER-OPTIONS?

   We type an option code asking for a *combined object tape*, a *memory-map printout*, and/or LOAD AND GO to execute the combined program (we could also start execution with the computer front-panel switch or by typing RUN).
8. After execution, the system returns to monitor control and types

   EXEC

We can, if we wish, *save the combined program* by giving it a *file name*, say NUFILE, and typing

SAVE TAPE 1/NUFILE

If we no longer need our source program MYFILE we can *delete* it by typing

DELETE DISK 1/MYFILE

To *save* MYFILE *on a library tape*, however, most operating systems require us to call another system program (copying program, peripheral-interchange program) specifically designed *to copy files from one peripheral device onto another*. Such a copying program must be given *format information* (binary, ASCII on paper tape, etc.). Other service programs callable from the system disk or from a library tape include *listing programs, editors, debugging programs, extra assemblers and compilers*, etc. (see also Secs. 3-12, 3-16, and 3-17). *Hardware-diagnostics programs* are usually loaded separately from paper tape, not through the executive.

The better executive programs can also be called from user programs, which may request loading and saving of specified files. This permits, in particular, successive core overlays of chained-program segments. Each program segment can call other overlays by simple external references (such as CALL SEGMENT 3 in FORTRAN), whereupon the resident executive causes loading of the desired segment. Such systems will, in general, include a special linking loader ("chain loader"), which loads all program segments (and all required library routines) of a chained program together as one big file onto a disk or tape prior to execution.

3-12. More Operating-system Features. (a) Input/Output Control System and Device Assignment (see also Secs. 5-27 to 5-32). Since the executive program will, in any case, comprise many input/output routines, most software systems incorporate their entire library of standard I/O routines (device drivers or device handlers for the most frequently used peripheral devices) with the executive program. All user-program requests for these device drivers take the form of system macros (like FORTRAN READ and WRITE statements) or subroutine calls linked through a portion of the executive program usually called the **input/output control system (IOCS)**. *The user need not write or know any details of I/O programs but only the simple calls on IOCS* (Secs. 4-20 and 5-31)

In our description of keyboard-executive operation (Sec. 3-11), we referred to I/O devices for the executive, compiler, and loader by actual *device names*, such as DISK 1, TAPE 3, etc. In a more elaborate system, it is preferable for system and user programs to employ **logical device numbers**. Thus, in the FORTRAN statement

READ (2, 17)

the device number 2 could be made to refer to the paper-tape reader or to a selected magnetic-tape transport by a device-assignment command in the executive program *without any change in the user program* (**device-independent I/O programming**). Similarly, any tape transport could be substituted for the system disk, magnetic tapes could be loaded on any one of several tape transports, etc. Device assignments can be changed by new entries in a device-assignment table in the executive program; there may be different device assignments for different system and user programs.

Users can ascertain the current device assignments for, say, the FORTRAN compiler by typing a request like

REQUEST FORTRAN ASSIGNMENTS

(this might be contracted into an abbreviated code). The system will answer by typing out device assignments, say

SYSTEM	DISK 1 (intermediate-pass storage)
INPUT	TAPE 3
OUTPUT	PAPER-TAPE PUNCH
2	PAPER-TAPE READER

The user may then *change device assignments* by a typed command like

ASSIGN TAPE 4 TO INPUT

for use in his compilation, but the standard device assignments will be restored the next time the compiler is loaded (see also Sec. 5-31*d*)

(b) System Generation. A minicomputer system with an executive program will need a special system program called a **system generator**, which tailors the executive program to a specific minicomputer configuration at the time the system is installed or modified. The system generator (supplied by the manufacturer on paper tape or magnetic tape) loads the skeleton executive program and completes it through a conversational sequence in which the user is asked to type in his memory size, his interrupt-system options, and a list of his peripheral devices. This procedure generates a *system tape* (paper tape or magnetic tape), which is saved and serves to refresh the system disk (if any).

(c) File Manipulation. Our example of executive-program operation included several instances where a program was saved on (or retrieved from) a "file-oriented" mass-storage device (disk or magnetic tape). A file is a block of instructions and/or data on a disk or tape usually ending with an end-of-file code and starting with a file header, which is a set of words comprising the file name and various information about the file. Each tape or disk will have a directory table listing all files stored (and, on disks and formatted tapes, also their addresses). One file can contain several programs or sets of data, but these will not be listed separately in the directory table; they must be found by scanning the file for record-header words

The executive program has **file-manipulation commands**, such as

FIND	DELETE
SAVE	RENAME
LOAD	READ HEADER

each followed by a device number (or name) and a file name. Note that SAVE, DELETE, and RENAME involve operations on the directory table as well as on the file. A keyboard command such as

DIRECTORY (device number or name)

will cause typing or CRT display of the directory table for the named device. *Copying or updating a file* involves *two* files and is usually accomplished by a copying program (peripheral interchange program) called by the executive.

Other file-manipulation operations include *saving and retrieving specified core areas* and *protection of specified files against deletion or overwriting*. Assembly-language operations or named files (or named blocks in a file) are done with the aid of READ and WRITE IOCS subroutines or macros (Secs. 4-20 and 5-31). The first READ or WRITE must be preceded by an OPEN FILE(device, file name) subroutine or macro, which reserves a core buffer for communication with the file and may initialize some I/O operations. After the user's program is finished with the file, a CLOSE FILE(device, file name) subroutine or macro dismisses the buffer and enters the file name, if it is new, into the device directory.

3-13. Real-time Executives and Batch-processing Monitors. The executive program described in Secs. 3-11 and 3-12 was specifically designed for *keyboard-controlled general-purpose computation*. In review, we see that the executive's main task is to call specified stored programs and data in response to interrupt-service requests from the keyboard. With the addition of IOCS to the executive, it also handles *user-program requests for routine I/O service*.

In many important applications, minicomputer systems must execute program sequences for reading instruments, processing data, or implementing control actions in response to interrupt requests from real-time clocks, sensors, or control logic as well as in response to keyboard commands. An executive program extended to handle such real-time service requests is known as a real-time executive (or real-time monitor). The real-time executive will again comprise IOCS, plus *skeleton interrupt-service subroutines* with entry points for user-written service programs. The user will write his service programs and label them for reference by the executive program, *which adds the tedious routines of the skeleton interrupt-service programs*, such as saving and restoring registers, I/O drivers, formatting, etc. (Secs. 5-27 to 5-32). The executive program will assign initial

interrupt-service priorities, which can be changed by certain user programs (Sec. 5-14).

An example would be executive-program control of a minicomputer system which must (1) perform routine data logging at clock-determined times, (2) compute some statistics from the data, (3) control valves or power-supply voltages supervised by temperature or voltage sensors, and (4) respond to overload alarms.

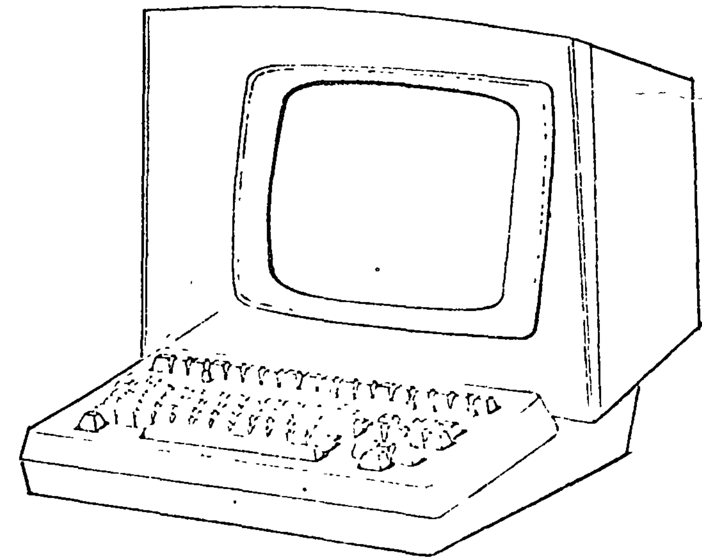


Fig. 3-11. A cathode-ray-tube/keyboard terminal for alphanumeric input/output. (*Delta Data Systems, Inc.*)

Another type of executive program (needed less frequently with minicomputers than with large digital computers) is a **batch-processing monitor.** Batch processing involves execution of multiple programs transcribed onto magnetic tape or a disk from punched cards or paper tape. Each program will have a *priority code* determined by the arrival time, urgency, and program length. The batch-processing monitor must load and execute the various programs, together with any needed library routines and data files, according to some priority-sequence strategy; the monitor must also make a record of the time and resources spent on each program.

3-14. Cathode-ray-tube/Keyboard Terminals and Other Peripheral Devices. The convenience of a keyboard operating system is greatly enhanced if we use the slow, noisy, and trouble-prone teletypewriter only when hard copy is really wanted and employ a **cathode-ray-tube/keyboard terminal** (Fig. 3-11) for communicating with the computer (and for program preparation and debugging, Secs. 3-16 and 3-17).

Typical CRT/keyboard terminals display 40 to 48 forty-character lines on a standard television monitor. This is inexpensive and quite satisfactory for most assembly-language and FORTRAN programming. If much text or tabular material is to be displayed or if the system handles programs prepared by or for a larger digital computer, then 80-character lines are preferable. An 80-character line can display a complete 80-column punched-card image or a complete 72-character teletypewriter line.

The pattern for each ASCII character is generated by table lookup in a MOSFET read-only memory (Sec. 1-14). The display character sequence is periodically refreshed by a serial (MOSFET-shift-register) memory. Most such terminals are *teletypewriter-compatible;* i.e., they connect to the serial teletypewriter interface on a minicomputer or communication system. It is usually possible to speed up the shift-pulse rate of such an interface when the much faster CRT/keyboard unit is substituted for a teletypewriter.

CRT/keyboard control keys are similar to those on teletypewriters but can have more pleasing and convenient arrangements. A blinking cursor, which can be moved up, down, left, or right by control keys, indicates the current point for character entry.

Other minicomputer peripherals include small *line printers, card readers,* and *pencil-mark readers. Graphic CRT displays* and *graphic plotters* will be discussed in Secs. 7-8 to 7-12.

3-15. A First Look at Minicomputer Time Sharing. Hands-on on-line computer operation, especially with CRT/keyboard terminals, is a tremendously effective way of working for *people.* It is also a very inefficient operation for the *computer,* which tends to be mostly idle while the operator thinks about the next step, scratches himself, or interprets results. It is surely one of the finer features of the inexpensive minicomputer that it still permits this type of operation to be cost effective (conversational computation on a larger machine *must* be time-shared). Nevertheless, especially in a research organization, a good on-line computing facility creates its own scarcity. Given a chance for creative "play" with the on-line computer, research workers may feel constrained not so much about cost but by the nagging feeling "I am depriving Joseph Blow of computer time." For this reason, some sort of time sharing is psychologically as well as economically indicated. Time sharing may also be attractive where a minicomputer supervises a relatively slow-moving experiment or process and is largely idle between operations.

Executive programs like those described in Secs. 3-11 and 3-13 can be readily extended to provide foreground/background programming. This is the simplest way to time-share the computer between two programs. The lower-priority *background program* is interrupted by keyboard, clock, or other device requests for the *foreground program,* which is entirely interrupt-driven. To prevent either program from overwriting the other, it is best to establish memory boundaries with memory-protection hardware (Sec. 2-15), which interrupts the guilty program before any harm is done. In any case, serious program changes must usually wait until both programs are finished. Some types of programs can coexist nicely (Ref. 11). More ambitious multiuser time sharing requires *program swapping from a disk,* which is usually controlled by a second minicomputer (Sec. 7-1[illegible]).

PROGRAM PREPARATION, EDITING, AND DEBUGGING

3-16. Program Preparation and Editing. (a) Off-line Paper-tape and Card Punching. The most primitive way to prepare a FORTRAN or assembly-language program for a minicomputer is with the built-in punch of an

ASR-type teletypewriter. An advantage of this procedure is that it can be done *off-line* if a second teletypewriter is available for the computer itself. Limited editing can be done with the aid of the teletypewriter RUBOUT key. The result is, generally speaking, a mess, although the computer can retype a clean copy of the program later; it is very hard to keep track of successive program corrections. Patching paper tapes for corrections is, essentially, impractical.

Conventional punched-card program preparation is preferable to such teletypewriter operation, assuming that a card punch and reader are available. The advantage of punched-card operation is that *individual cards can be corrected.* Punched-card operation is also employed where assembly or compilation of minicomputer programs is done with a larger digital computer.

(b) Editor Programs. Most minicomputer program preparation is done with the aid of editor programs loaded from paper tape, from magnetic tape, or from the system disk. The text to be edited (program or data, usually in ASCII-character format) comes from a teletypewriter, CRT keyboard, or paper-tape reader, or from an *unedited file* on a magnetic tape or disk. The editor program moves this text to an *edited file* or output device by way of a working area (variable-length text buffer) in core. The text in the working area is printed out by a teletypewriter or (preferably) displayed by a CRT/keyboard terminal and can be modified, deleted, or added to by means of the teletypewriter or CRT keyboard. This is very convenient but, unfortunately, ties up the computer itself for on-line editing.

In the input mode (text mode) of a typical editor program (Ref. 12), the user can type on the current text line and delete individual characters, or the entire line, with control keys. The working area to be edited may be the last line typed or a block of lines. Text may be output from the working area line by line (after each line-feed/return), or an entire block may be output on command.

A control key switches the editor program between input mode and command mode. In *command mode,* the user may type *editor commands,* such as:

TOP Moves the current line to the first line of the input file (if any).

NEXT Moves the current line to the start of the next block or display page.

BOTTOM Moves the current line to the last line of the input file.

GET N LINES FROM DEVICE Adds N lines from a subsidiary input device after the current line.

LOCATE (character string) Moves the current line to the first occurrence of the quoted string. It is used, for example, to change a variable name each time it occurs.

CLOSE (file name) Outputs the remainder of the input file, edited or not, and closes the output file at the end of an editing job.

When a teletypewriter is used, most actual editing (text modification) is done in *command mode:* the user types commands underneath the current line, e.g.,

INSERT (string) The string just typed is inserted after the current line.
REPLACE (old string/new string) The old string in the current line is replaced with the new string (which can be longer).

The editor can also be commanded to *retype* some or all the text or to prepare *paper-tape output.*

Editing with a CRT/keyboard terminal is by far more convenient. The user can *type over the displayed text, insert characters or lines with the aid of control keys,* and *delete characters or lines.* All these changes are automatically applied to the text buffer.

NOTE Editing programs can be very useful for editing *general text* (e g , reports) as well as computer programs and data

3-17. On-line Debugging (Ref 13). A good **program-debugging system** works with both FORTRAN and assembly-language programs. The debugger is loaded with the linking loader, which loads the user's programs; as a rule, the debugger disables all interrupts, so interrupt-service routines must be tested separately. The debugger can:

1. Insert breakpoints at specified memory locations. When started, the program will run and stop at the next breakpoint to permit examination of register and memory contents.
2. Remove one breakpoint or all breakpoints
3. Start or restart the program at a specified location, e.g . after a breakpoint.
4. Display the contents of a symbolically addressed memory location on the teletypewriter or CRT terminal, step the displayed-location address up or down, or display the contents of the memory location indirectly addressed by a desired symbol.
5. Modify the contents of memory locations addressed as above.
6. Search a specified area in memory for the location addressed by a specified symbolic expression or for specified symbolic contents.
7. Output modified or corrected program sections onto a named file; the new program sections can contain newly defined symbols.

Less elaborate debugging programs permit only octal (rather than symbolic) address references (octal debuggers). Reference 12 contains a short example of a debugging session; see also Ref. 13.

REFERENCES AND BIBLIOGRAPHY

(Refer also to manufacturers' manuals for computers and peripheral devices.)

Minicomputer Peripheral Devices

1. Rexroad, W T *Teletypewriter Fundamentals Handbook*, Computer Design Publishing Corporation, West Concord, Mass , 1970
2. Murphy, W. J Cassette-Cartridge Transports, *Mod Data*, August 1970
3. Kashman, M. J Computer Peripherals Buyer's Guide, *Control Eng* , February 1970
4. French, M · Digital Cassette and Cartridge Recorders, *EEE*, May 1970
5. Sykes, J. R · Design Approach for a Digital Cassette Recording System, *Comput Des* , October 1970.
6. Brick, D. B , and E. N Chase: Interactive CRT Terminals, *Mod. Data*, May–July 1970
7. Murphy, W. J : Medium and Small-scale Disk and Drum Drives, *Mod Data*, March 1971.

BASIC Programming

Many minicomputer manufacturers supply BASIC manuals. Among the best ones are:

8. *Programming Languages* (PDP-8 Handbook Series), Digital Equipment Corporation, Maynard, Mass. (current year)
   This reference also describes FOCAL.
9. 2000B *A Guide to Time-shared BASIC*, Hewlett-Packard Corporation, Cupertino, Calif (current year).

There are many books on BASIC. The classic text is that by BASIC's originators:

10. Kemeny, J. G , and T F Kurtz *BASIC Programming*, Wiley, New York, 1967.

Miscellaneous

11. Puls, J H.: A Simple Method of Multiprogramming the PDP-9 Computer, *CSRL Rept* 207, Electrical Engineering Department, University of Arizona, 1970, see also *Proc Fall DECUS Symp* , Digital Equipment Corporation, Maynard, Mass., 1970
12. *Introduction to Programming*, PDP-8 Handbook Series, Digital Equipment Corporation, Maynard, Mass. (current issue)
13. Evans, T. G., and D. L Durley· On-line Debugging Techniques· A Survey, *Proc FJCC*, 1966 (contains further bibliography on debugging).

CHAPTER 4

MINICOMPUTER PROGRAMMING WITH ASSEMBLERS AND MACROASSEMBLERS

INTRODUCTION AND SURVEY

Assembly-language programming will enable us to obtain the greatest possible effort from a digital computer, i.e., to optimize computing speed and/or memory requirements. This is because assembly-language instructions correspond, more or less, to the actual hardware operations possible with a specific machine and permit us to exploit its features cleverly. This advantage of assembly-language programming is especially pronounced for small digital computers, whose algebraic compilers (which must fit into 4K to 8K words of memory) may not produce very efficient code.

Modern symbolic assemblers not only *translate instruction mnemonics into machine code* but also permit *symbolic memory references* by assigning binary location numbers to symbols (Sec. 4-2). The better symbolic assemblers can also *compute addresses by evaluating symbolic expressions* (Sec. 4-3), can *reserve blocks of storage locations* (as well as single storage locations) for data or instructions, and can arrange for storage and formatting of decimal, double-precision, and floating-point data (Sec. 4-5). Good general-purpose assemblers further free the programmer from assigning program pages and work with a companion linking-loader program to facilitate *relocation and linkage of multiple program segments* (Sec 4-17, see also Sec. 3-6). Finally, *macroassemblers* can generate useful multi-instruction sequences from one-line commands (Sec. 4-21) and, together with *conditional assembly* (Sec. 4-23), can combine some of the programming simplicity of a compiler language with assembly-language efficiency.

With a suitable operating system, assembly-language program segments can be neatly combined with FORTRAN programs (Sec. 4-20) so that even a little knowledge of assembly language can be used to improve important or frequently used routines.

ASSEMBLY LANGUAGES, ASSEMBLERS, AND SOME OF THEIR FEATURES

4-1. Machine Language and Primitive Assembly Language. A typical program sequence for a 12-bit minicomputer, say

2	5	
3	. . .	
4	LOAD INTO ACCUMULATOR (the contents of)	2
5	INVERT ACCUMULATOR	
6	STORE ACCUMULATOR IN	3

specifies the contents of successive memory locations 2, 3, 4, 5, *and* 6. Location 2 contains a *data word* (5) given by our program, but location 3 is only *reserved* for an as yet unspecified data word to be stored there by the program. The program proper (i.e., the first instruction) starts at location 4. *The program counter will be initially set to* 4 and will step to 5, 6, and on to 7 as each instruction is executed.

Such a program is actually entered into the computer in binary machine language, viz.,

000	000	000	010	000	000	000	101
000	000	000	011	. . .	. . .	. . .	. . .
000	000	000	100	001	000	000	010
000	000	000	101	111	000	100	001
000	000	000	110	011	000	000	011

perhaps from a binary paper tape or from front-panel toggle switches. The first 12-bit word on each line is the memory address of the second word. The first line again locates the data word (5). The second line reserves location 3 for a data word which is not supplied by the program, but will be stored there at run time by our last instruction; some assemblers would deposit 0 in such a location for the time being.

The first word of the third line is, again, the address of the second word. This time, this stored-program word represents an *instruction code* and, since this is a memory-reference instruction, some address bits needed to determine an effective memory address. In our simple example, the five leading instruction-code bits 001 00 signify LOAD INTO ACCUMULATOR with the "page 0" direct-addressing mode (Sec. 2-7). In this case, the remaining seven address bits 0 000 010 directly represent the binary address. The remaining two instructions are similarly translated.

that we should at least mention it here. A typical and especially important application is *minicomputer automation of supermarket check-out stands*, as developed by Honeywell Information Systems. As individual sales items are checked out, the operator either manually keys a code number printed

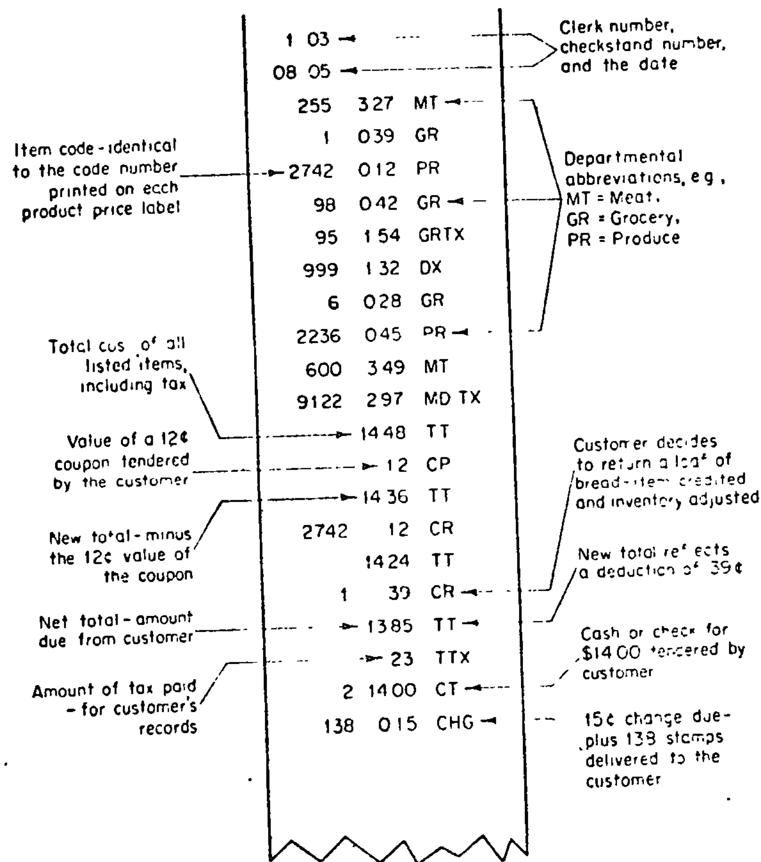


Fig. 7-9. Minicomputer-prepared supermarket sales receipt indicating the amount of information handled by the computer system (*Honeywell Information Systems*)

on the individual can or cereal package into a local minicomputer, or the code is read automatically with a mark-reading sensing probe.

The check-out terminal displays the item's price (and, if relevant, its weight) to the customer and computes the total purchase price and sales tax, and prints a sales tape (Fig. 7-9). The checker need not read, compute, or enter prices or sales totals. A computer-compatible scale next to the terminal is used to weigh produce.

The store manager has a counterpart of the check-out terminal ("manager's interrogation device," MID). Since the minicomputer keeps track of the sale of each individual item, the manager can use his terminal to ascertain his inventory of any product, to determine sales up to the moment in any department or at any check-out stand, to know total sales, to change prices, to see whether any purchases have been paid for with coupons or food stamps, and to determine what taxes have been paid. In addition, all bookkeeping concerned with sales and inventories is taken care of by the small computer; books are balanced more accurately because automatic computation has replaced human arithmetic errors. The minicomputer will prepare daily sales reports of the entire store, showing the number of customers handled by each checker at each check-out stand, to permit management to schedule and rate store personnel and thus to reduce costs. A daily sales report by stores and with total transactions permits management to evaluate store operation, the effect of special promotions, and to handle its own inventory problem. These reports can be printed out and sent through the mail, or the local minicomputer may be connected to a supervisory computer at a central location through a communication link.

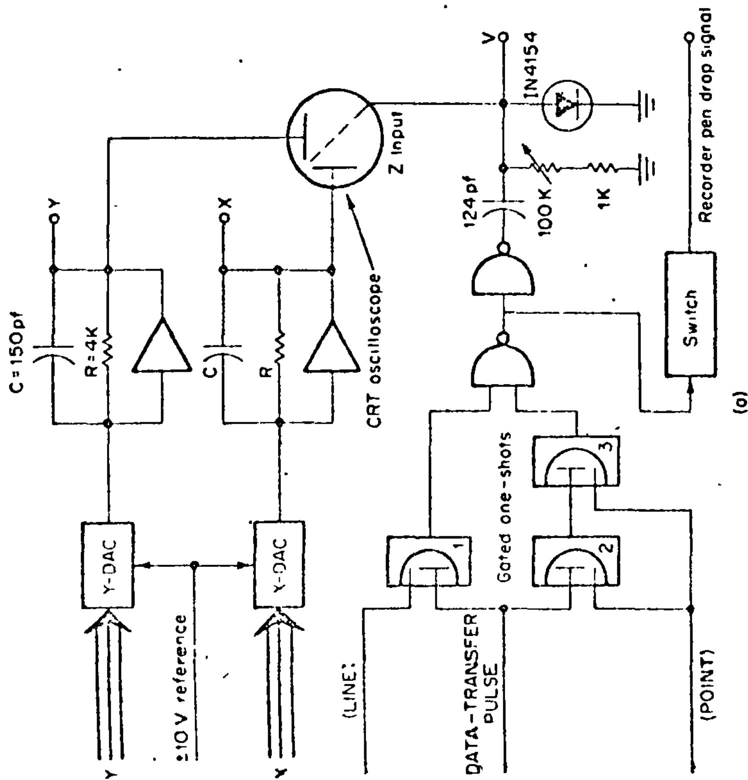
CATHODE-RAY-TUBE GRAPHIC DISPLAYS AND SERVO PLOTTERS

7-8. Cathode-ray-tube Displays. A cathode-ray-tube graphic display positions and brightens a CRT beam to plot a sequence of points (and/or brightens the beam *between* points to draw line segments or "vectors") Small displays (up to 11-in diameter) employ *electrostatic deflection* in the X and Y directions and can plot up to 10^6 distinct points/sec. Larger displays use *electromagnetic deflection* for better focusing and resolution, but such displays are slower (up to 100,000 points/sec). High-quality electromagnetic-deflection displays may add fast electrostatic deflection for small beam displacements (e.g., to display characters labeling a picture).

Figure 7-10 shows how the X and Y deflection amplifiers of a CRT display are driven by X and Y digital-to-analog converters (DACs). A digitally controlled brightening voltage (Z-axis voltage) is also indicated. 9-bit X and Y resolution is quite satisfactory for most CRT displays, but many displays have 10-bit DACs.

Very elaborate displays can use full 16-bit resolution to specify points in a picture much larger than actually displayed on the CRT screen. 10-bit portions of the 16-bit X- and Y-coordinate words are then shifted into position to display small or large portions of the overall picture at different scales (*scissoring*).

Displayed pictures range from simple 256-point graphs with coordinate axes to elaborate design drawings with several thousands of points, plus alphanumeric characters. A *storage-tube CRT display* (Fig. 7-11) permits



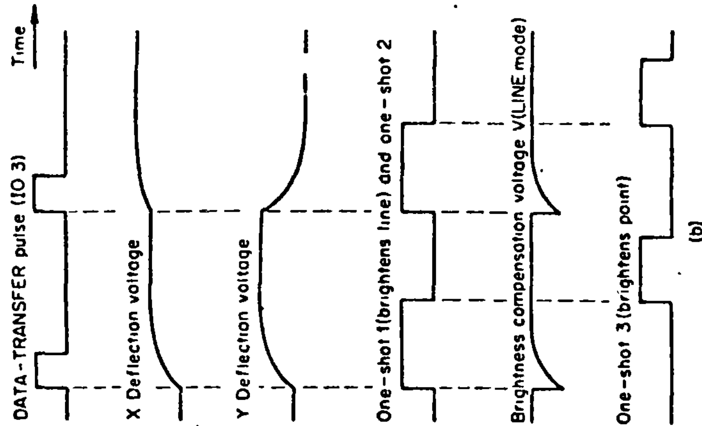


Fig. 7-10. A simple digital CRT display suitable for Dertouzos-type line-segment display as well as for point display (*a*) and waveforms (*b*). The low-pass-filter capacitors *C* establish equal *X* and *Y* time constants *RC* and also reduce transients ("glitches") due to switching spikes and different DAC bit-switching times. The line-brightness-compensation-voltage waveform also has the time constant *RC* to brighten the beam most when it moves most quickly. But perfect compensation is not possible (Fig. 7-11*a*). The *X* and *Y* display outputs can operate at a slow rate with an *xy* servo recorder, in which case the brightening logic drops the recorder pen on the paper. (*University of Arizona*, see also Ref. 39.)

you to view the displayed points after they have been written only once—the display remains visible until a manually controlled or computer-controlled voltage pulse is applied to an erasing electrode in the storage tube. Storage CRTs have excellent resolution and greatly simplify display operation. But they cannot display *moving* pictures and require complete erasure and rewriting for *display editing*. Most digital displays, therefore, use short-persistence cathode-ray tubes (P7 phosphor) and must rewrite (refresh) the entire display periodically 30 to 60 times/sec. This requires not

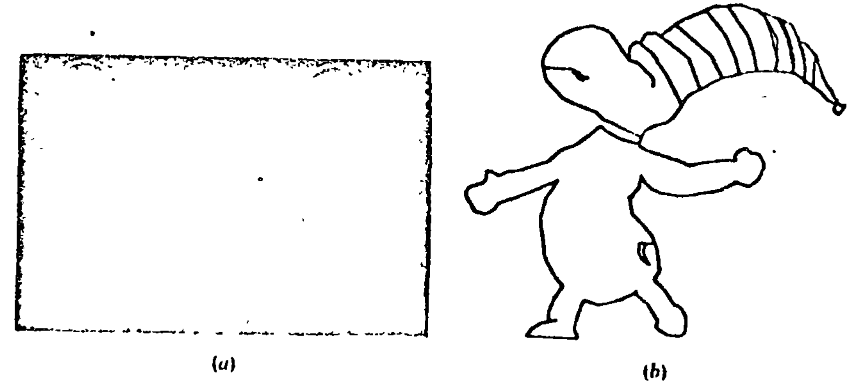


Fig. 7-11. CRT displays (*a*) and servo plot (*b*) produced by the simple display/plotter circuit of Fig. 7-10. Both POINT and LINE modes are used in Fig. 7-11*a*; note the effect of adjusting the line-brightness-compensation time constant. Perfect compensation for the exponential change in the writing rate was not possible because the compensation voltage tends to defocus the beam. (*University of Arizona.*)

only many fast writing operations but a display-refreshing memory capable of storing coordinate and brightness information for 1,000 to 6,000 points and/or vectors. *Alphanumeric characters* are generated and refreshed as sets of points or vectors (strokes) usually stored in special read-only memories (character generators) and called out by special character-code display-instruction words.

Each display point will require 18 to 20 bits of refresher storage for X and Y plus, possibly, some extra bits to specify brightness or special display operations. Some CRT displays, especially the more elaborate displays used with larger digital computers, have their own 16- to 24-bit refresher memories, perhaps 4K to 16K words. A minicomputer display can conveniently share the minicomputer memory. This simplifies computer operations on display words and makes the extra memory available to the computer when the display is not used; although the time needed for display-refreshing operations will necessarily slow concurrent computations.

7-9. Display Operations and Interfaces. (a) Simple Point Display. To display a point, we transfer its X- and Y-coordinate words from a processor register or from memory into the X and Y DAC registers (Fig. 7-12) and then brighten the beam. This can be done through programmed I/O instructions with different control bits and IO pulses (Sec. 5-2), but **it is much more**

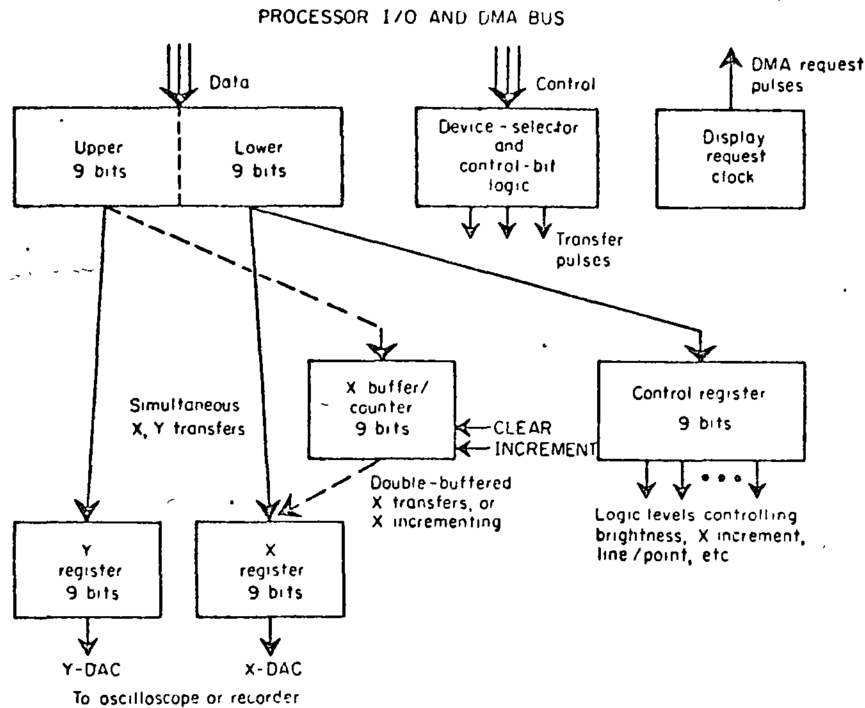


Fig. 7-12. Design of a graphic-display interface for an 18-bit minicomputer. Programmed or DMA data transfers can transmit packed X-Y words or load a buffer with X and then transfer X from the buffer and Y from the data bus. It is also possible to simply *increment* the X buffer for graph plotting while transferring only Y-coordinate words from the bus. The 9-bit control register is loaded with the last 9 bits of any DMA data word starting with 100 000 000. (*University of Arizona*; see also Ref. 39 and Sec. 7-10.)

efficient to employ direct-memory-access block transfers (Sec 5-19). A DMA display interface can readily request and transfer alternate X and Y words (from one array or two arrays in memory), but an especially neat scheme is to use an 18-bit minicomputer with 9-bit X and Y bytes packed into a single word; this *halves* the refresh memory needed *and* the computer time needed to refresh the display, and simplifies the interface. Figure 4-8*d* shows a suitable word-packing program. In Fig. 7-12, a brightness control bit gates the transfer pulse loading the X and/or Y DAC into a pair of monostable multivibrators to brighten the beam. One usually controls brightness by changing the *duration* of the brightening waveform (e.g., by

ORing outputs of different logic-controlled monostable multivibrators). Beam *current* changes will also control brightness, but may defocus the beam.

(b) Simple Line-segment Generation. Figure 7-10 also illustrates the Dertouzos technique of displaying *line segments* between display points. *The X and Y DACs shown drive operational-amplifier low-pass filters with equal time constants RC so that the beam will move from point to point along a straight line after the X and Y DACs have been loaded simultaneously* (Fig. 7-10*b*). This line is brightened if a control bit gates the DAC transfer pulse into monostable multivibrator 1. Unfortunately, the beam speed varies exponentially along each line segment, so the beam becomes progressively brighter. This is partially compensated in Fig. 7-11 by a differentiating network in the brightness control circuit, but beam defocusing makes perfect compensation impossible (Fig. 7-11*a*). *The simple Dertouzos line-segment generation technique is, however, excellent for producing hard copy with a simple servo plotter.* Figure 7-11*b* shows a drawing produced by feeding the X and Y inputs of a servo recorder with the display circuit of Fig. 7-11*a*; the brightness voltage lowers the pen to plot line segments. The transfer rate was about 10 points/sec, and the monostable-multivibrator time constant was appropriately longer (Refs. 39 and 40).

(c) Improved Line-segment Generation and Incremental Display Techniques. More elaborate line-segment generators employ operational-amplifier integrators for *straight-line* interpolation between successive coordinate voltages so that line brightness will remain constant between successive display points. If the time interval between successive display points remains constant, though, short line segments will necessarily be brighter than long ones. For this reason, elaborate displays employ *digital interpolation* (hardware or software similar to numerical-control methods, Sec. 7-2) to place extra display points between widely separated points; analog interpolation may still be used. Electromagnetically deflected CRT beams can, in general, follow short displacements more quickly than long ones.

In many of the better graphic displays, DAC registers (or DAC buffers) are implemented as *reversible binary counters*, which can be incremented or decremented by IO pulses to produce small beam displacements. The incrementing pulse may be gated to higher-order or lower-order bits to produce increments of a few different sizes, but many displays only permit

$$\Delta X = -2^{-10}, 0, \text{ or } 2^{-10} \qquad \Delta Y = -2^{-10}, 0, \text{ or } 2^{-10}$$

so increment/decrement operations can move a display point only in one of eight directions separated by 45° angles. *Incrementing-mode* display programs can generate any reasonable curve from such displacements.

In display pictures containing continuous curves or small detail, incrementing-mode instructions can save refresh memory and memory accesses

at the expense of extra display-logic hardware. For example, the 10-bit X and Y coordinates of a single point require *two* 16-bit words. But a *single* 16-bit word could specify up to 2^{16} different combinations of X and Y *increments* (usually the hardware will not permit all possible combinations). With still more elaborate hardware, a display can be instructed, say, to *repeat* the same beam displacement n times to generate a straight line from n line segments.

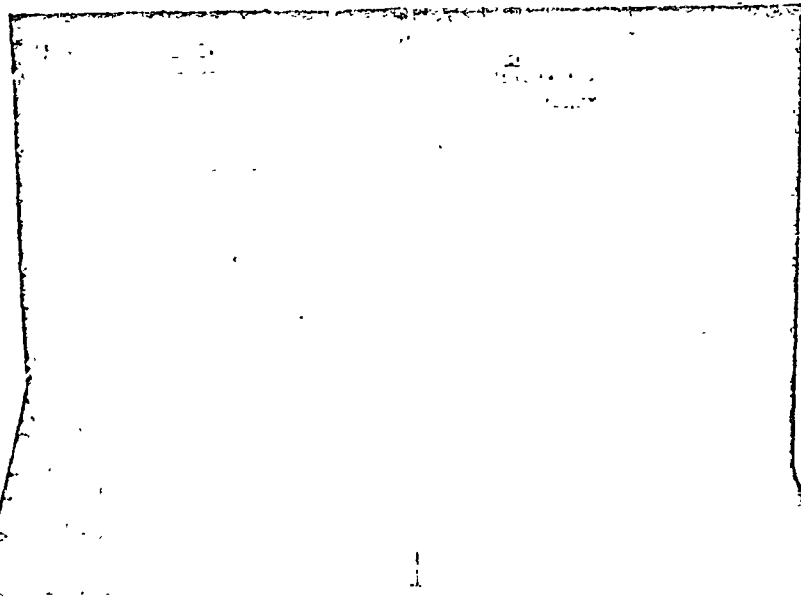


Fig. 7-13. A simple homemade graphic display is combined with a TV-scan alphanumeric display in this minicomputer system for on-line solution of ordinary differential equations. Reading coordinate labels and scales off the alphanumeric display is much like reading a figure legend and imposes no hardship. (*University of Arizona*.)

Even simple CRT displays may permit incrementing the X coordinate to permit graph plotting (Y versus X in equal increments) without any need to fetch X-coordinate words (Fig. 7-12).

(d) A Suggestion for Do-it-yourself Displays. Addition of alphanumerics to graphs and pictures (e.g., labels on coordinate axes) complicates display hardware and software, because:

1. The display of characters adds much fine detail and many display points
2. Characters are generated from points (five by seven dot matrix) or strokes by reference to a fair-size table, which must be stored either in the computer memory or in display-controller hardware (MOSFET read-only memory)

3. The need for character spacing and line feed requires still more software and/or hardware

An excellent way to simplify this situation is to provide two displays side by side, viz., a simple graphic display (refreshed or storage-tube display) *and* an inexpensive TV-scan alphanumeric display with MOSFET character generation and refresher shift registers (Fig. 7-13), which permits convenient text editing and can work with only slight modification of teletypewriter software (Sec. 3-14).

7-10. From Display Control Registers to Display Processors. The simplest graphic displays have only a simple point mode with a single brightness level. Such a display can be operated with only *two* I/O instructions, viz., TRANSFER X and TRANSFER Y AND BRIGHTEN. If we use packed 18-bit X, Y words, a *single* I/O instruction will do (TRANSFER AND BRIGHTEN).

As we noted, though, it is *by far* more efficient to fetch alternate X- and Y-coordinate words, or packed X, Y words, *by direct memory access*. For example, the points of a simple picture may be represented as packed 18-bit X, Y words stored in an N-word block starting at the memory location PICT.

To display the picture (i.e., its N successive points), the program first places the addresses of PICT and N into two pointer locations in memory. The program then enables interrupts from a real-time clock in the display or processor to refresh the display 30 to 60 times/sec through the following interrupt-service routine:

1. Programmed I/O instructions preset a *current-address counter* and a *word counter* (Sec. 5-19) in the display (or in the computer memory, Sec. 5-23) to PICT and to N, respectively.
2. Another programmed instruction enables a DMA request-pulse oscillator in the display to produce successive cycle-stealing coordinate-data transfers and to display successive points.
3. The word counter counts down from N with each DMA transfer and *stops the request oscillator* when the count reaches 0, presumably before the next clock interrupt repeats the process.

More complicated display programs will display multiple blocks of points, corresponding to different portions of an overall picture (Sec. 7-11).

Display options associated with individual display points, such as different brightnesses, line brightening, and X incrementing for plotting graphs, are controlled with logic levels from a display control register (Sec. 5-4), which may have between 1 and 18 flip-flops. Programmed display instructions can include a few control bits, and there may be special instructions to load the control register. To obtain the control-register information through

direct memory access:

1. Control bits may be packed into data words (e.g., a 16-bit word could contain a 10-bit Y coordinate and 6 control bits).
2. The display may always request X words, Y words, and control-register words in succession. This can waste a good deal of time.
3. The display may recognize certain codes as data words and some as control words.

As an example of the last possibility, packed 18-bit X, Y words need not represent $X = -1$ and $Y = -1$, since $X = 1$ and $Y = 1$ are not available in 2s-complement code either. Thus, data words *beginning* or *ending* with 100 000 000 can be used to load two 9-bit control registers; such control words can be freely inserted into the display file, as needed. Figure 7-13 illustrates the design of a simple graphic-display interface with a control register (Ref. 40).

Our direct-memory-access display interface can be regarded as an accessory processor (display processor) which shares the computer memory, accepts programmed instructions from the central processor, and can respond with interrupts (see also Sec. 6-12*a*). The accessory processor has:

A *program counter* (the DMA word counter)
A *memory address register* (the DMA current-address counter)
A *memory data register* (DAC register or buffer)
An *instruction register* (the display control register)

The simple "instructions" executed by the display processor are DISPLAY A POINT (using X- and Y-coordinate information), CHANGE BRIGHTNESS, etc. With more elaborate display operations, the display processor looks more and more like a small stored-program computer; it might implement:

Coordinate-incrementing instructions (Sec. 7-9*b*)
Display subroutine jumps and returns, using a display linkage register to store return addresses
Hard-wired subroutines (ROM-implemented character generation called by suitable control words)
Output to multiple CRT consoles

Display operations involving actual *arithmetic*, e.g.,

$$X = aX' + b \qquad Y = aY' + c \qquad (\textit{TRANSLATION AND SCALING OF PICTURE OR SUBPICTURE})$$

$$X = X' \cos\vartheta + Y' \sin\vartheta \qquad Y = -X' \sin\vartheta + Y' \cos\vartheta \qquad (\textit{ROTATION})$$

can be implemented either in the main processor or in the display processor; a few display processors incorporate fast multiplying digital-to-analog converters for rotation operations. The display processor can be a complete minicomputer.

7-11. The Display File and Display Software. The display file for a simple picture is a block of words containing display-point-coordinate and display-control information. There must also be "header" words specifying the block starting address and the block size (PICT and N in Sec. 7-10). It will be expedient to structure display files for more complicated pictures as *linked lists* (Sec. 4-10*c*): each item in the list is a subpicture file ending with a pointer to the start of the next subpicture file and its block size. Display requests can then fetch each subpicture in turn, and it will still be possible to perform operations such as erasure, scaling, or rotation only on selected subpictures.

Suitable header or label words can further structure subpictures into hierarchies of sub-subpictures; so operations can be performed on sets of sub-subpictures which in some sense "belong together." Display-structuring and display-modifying operations can be called as assembly-language subroutines or macros and as FORTRAN subroutines, with symbolic names for display files and subpictures.

7-12. Operator/Display Interaction. Joy sticks, various tablet-stylus combinations, and the "mouse" rolling on a table surface all contain dual analog-to-digital conversion devices which enter X and Y coordinates into a computer display file so that the operator can "draw" points and lines on the CRT screen.

A light pen contains a photocell, which is held against a CRT display screen and which responds to the flash of a display point with an interrupt or sense-line response. The computer can then mark the X and Y coordinates of the point in question to *erase* or further brighten the point. The computer can generate a dimly lighted raster or random-scan pattern and brighten points touched by the light pen (which contains a button switch to disable this action, if desired), so the operator can draw pictures on the CRT screen. The computer can also generate a *tracking pattern* with a program designed to move this pattern in order to center it on the light pen; this can also be used for drawing on the screen and for moving subpictures (e.g., circuit or block-diagram symbols) into desired screen positions with the light pen.

Finally, the computer may display a "menu" of possible decisions or commands on the CRT screen, each with a "light-button" pattern which is touched by the light pen to implement the command.

FRONT-ENDING, DATA-COMMUNICATIONS, AND MULTIPROCESSOR TIME-SHARING SYSTEMS

7-13. Minicomputers as Input/Output Processors. An ever-increasing number of minicomputers are employed as front ends intended to relieve a

larger digital computer of input/output operations and its memory of multiple input/output routines. We discussed in Sec. 7-10 how a data channel designed to implement block transfers of input and output data (and to perform a few extra device-control chores) acquires many of the features of a small digital processor. A minicomputer transferring data blocks by direct memory access and communicating with a larger digital computer through programmed instructions and interrupts (see also Sec. 6-12) can perform data-transfer and control operations equivalent to those of several data channels. like a data channel, it accepts programmed instructions to preset address counters, word counters, and control registers and, in turn, speaks to the larger digital computer through processor interrupts.

But a minicomputer can do much more than transfer data blocks and control device functions The minicomputer memory *buffers* external devices, which can thus operate at their optimal speed without waiting for the main digital-computer program, and vice versa. If there is the time, moreover, the minicomputer peripheral processor can perform formatting, scaling, and code-changing operations, sense and announce error conditions, and perform parity and syntax checks. What is more, many input/output programs, interrupt-service routines, etc , can be stored in the minicomputer memory at substantially lower cost than is possible in the more expensive large-computer memory with its greater word length. Tens of thousands of bytes of main-computer core storage may be saved in this manner. Minicomputers have been used as peripheral processors for practically all types of peripherals, such as multiple teletypewriters, CRT displays, line printers, disks, multiple tape units, and communication interfaces (Sec. 7-14). Such applications favor minicomputer instruction sets which permit 8-bit byte handling and operations on multibyte data words. In particular, microprogrammed minicomputers may be furnished with instruction sets especially adapted to those of an associated large digital computer. As an example, the microprogrammed Interdata Models 70 and 80 have instruction sets conveniently related to those of IBM System/360 and 370 machines.

7-14. Minicomputers and Data Communications (see Refs. 48 to 58). For data communication over distances greater than a few hundred feet, digital words are transmitted serially (bit by bit, Sec. 1-3) and *modulate a carrier* on a communication line or wireless data link. Amplitude, phase, frequency modulation, or combination amplitude and phase modulation is used. Communication links specially designed for digital data transmission may employ radio-frequency carriers, but audio frequencies are used on telephone lines (which are not primarily designed for data transmission). At each end of any carrier link, one requires a modulator/demodulator (modem).

Simplex transmission is one direction only; half-duplex permits communication in both directions, but only one at a time; full-duplex permits simultaneous transmission and reception (e.g., on two two-wire lines).

Conversion between parallel computer input/output lines and serial bit streams is usually achieved with *shift registers* having parallel input/serial output and/or serial input/parallel output (Table 1-5*b*; see also Sec. 5-1). Most digital data transmission is in terms of 8-bit ASCII-character bytes, 1 bit being a parity bit (Sec. 1-4). It is necessary to mark the start and/or

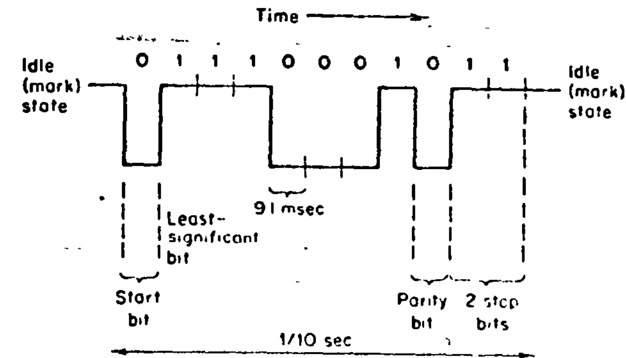


Fig. 7-14. Serial representation for an ASCII 7-bit-and-parity character Start and stop bits "delimit" the serial character representation for asynchronous transmission Timing is for an ASR-33 teletypewriter operating at 110 baud (bits/sec) or 10 characters/sec.

end of each byte or word unequivocally. In asynchronous data transmission, this is done through start/stop bits inserted between data words by the transmitting shift-register interface (Fig. 7-14). Since 3 start/stop bits transmitted with each 8-bit byte waste a good deal of time, asynchronous transmission is used only with low-cost, slow data-transmission systems (up to 1,400 bits/sec). Faster data-communication systems justify the cost of synchronous transmission, which transmits a continuous stream of true data bits and marks the start of a *message* (not the start of a word or byte) with a synchronizing signal (or the *end* of a LINE IDLE signal) transmitted *over an extra link or carrier frequency*. In any case, the interface between a parallel computer I/O bus and the serial input/output of a modem requires, besides a shift register, a bit-rate-determining clock, a bit counter, and some logic which generates or recognizes start/stop bits or synchronizing signals This interface, which is usually designed for a specific computer, is called a **data-set coupler (data-set controller)** and connects to a general-purpose data set combining the functions of a modem and various options, such as automatic telephone dialing.

Extra hardware registers are generally desired to help increase the execution speed of a machine, but they may actually slow down the interrupt response time if they have to be saved. The problem of protecting one user from another is one of the keys to making a general purpose RTE system work smoothly. There can be no reliance on the good will of other programs here, as there could be in a dedicated system. This is especially true where compiling and debugging of programs is to be done in the background while the system is running.

When thinking about the I/O system, one must consider that most or all of the I/O will be done by the operating system. Will it handle the variety of devices needed, and how difficult will it be to write and add special I/O routines to the system? Does the system overhead impair the transfer rate of the hardware? Generally, the user is not allowed to get at the I/O system in an RTE since I/O is a critical resource that is shared among all users--again the problem of protection arises.

Similarly, memory protection among users must be considered. Is it adequate? Does it add time to the basic memory cycle? Is it a hardware or a software function? Associated with this problem is that of virtual memory systems. Does the operating system provide each user with a memory address space that looks as though he is the only program on the machine? If so, how much overhead is added to the system to provide this desirable feature? Memory size then becomes of concern only in that it is big enough to hold the working set of programs.[3] The multiprogramming operating system then handles memory allocation and the swapping of programs, and data are demanded by either an interrupt or the software scheduler.

If the user programs in a high level language, the word size and instruction set are pretty much invisible to him. They may show up indirectly in running time if, for example, too many multiple precision fetches are being made or too many subroutines are called to perform operations that are not in the hardware. The instruction set should contain features designed to help the operating system in order to decrease the overhead. For instance, relative addressing is a must on multiprogramming systems if a large amount of time is not to be spent on every memory swap to relocate the addresses to keep from having to load a program into the same exact location every time. Is the memory protection an integral part of the instruction set, or is it an add-on after thought that may cause additional system overhead? The computer manufacturer's software department has the difficult job of using the instruction set to write the operating system and the compilers. It would seem reasonable that instructions would be included in the initial design to help high level languages have increased code compression and shorter execution times.

The prospective user would be well advised to carefully examine the software provided with an RTE system to see how efficient and easy it is to use. What high level languages interface with the RTE? Can batch jobs be run in the background and tasks added to the foreground while the system is operating? Is there a reasonable library with routines applicable to the job at hand? Can these routines be shared at run time, allowing reduced core requirements? All these questions and many more should be asked, since it is the software specifications that make up a major part of any real time executive system specification.

In summary, one can say that the architecture of a real time system is made up of many facets, including the hardware, the software and all the devices connected to the system in the particular application. The form is not complete and cannot be specified without all of these elements. Each individual specification cannot be set up and examined alone, as though it were the deciding factor in designing or purchasing a real time system, but must be taken as part of the whole. In conclusion, one hopes that both designer and users will go beyond the traditional and parochial evaluation of a potential system, joining me in a plea for sanity in those who conceive the specs, those who write the specs, those who read the specs, and those who analyze the specs in terms of the requirements of the overall system.

References

1. Minutes of Third Workshop on Standardization of Industrial Languages, Part 1, Purdue University, March 2-6, 1970
2. Basic Control System Reference Manual, Hewlett-Packard, February 1968
3. Denning, P. J., "The Working Set Model for Program Behavior," Communications ACM, Vol. 11, No. 5, p. 323-333, May 1968.

A review of hardware and software...

Comparative Criteria for Minicomputers

J. L. BUTLER, Fisher Controls Co.

Minicomputers come in many varieties. The author tabulates data for 45 models available from 27 manufacturers. The roundup is not complete, but the minicomputer data presented here is comprehensive and forms the basis for an analysis of "price to performance ratio." The author proposes three equations that permit calculation of a minicomputer's hardware price/performance (P_h), software price/performance (P_s) and overall price/performance (P).

LOW COST IS THE FACTOR that makes "minicomputers," the small computers, so attractive to industry. For less than $10,000 a central processor and 4,096 words (4K) of core memory can be purchased—hardware that approximates the computing power of the top of the line computers available 15 years ago. This economy has been accomplished with medium scale integration (MSI). The large scale integration (LSI) technology which is potentially more economical has yet to be applied in industrial control.

The flood of minicomputers into the marketplace continues because the number of potential applications is almost limitless. In the process industries, these small computers are used in increasing numbers in the laboratory and for process control as well as in data communications and data acquisition systems.

The dynamic nature of minicomputer technology produces the biggest problem encountered in a minicomputer survey. Computer manufacturers offer and revise new and old models continuously, so keeping up with the minicomputers could be a full time job.

This survey and analysis of minicomputer hardware and software covers 45 models that are made by the 27 manufacturers listed in Table I. Discussion is limited to the manufacturers of computer mainframes; OEMs aren't included. Computers such as the IBM 1130, CDC 1700 and the GE/PAC 4020 were not reviewed because of their price.

Each of the 45 minicomputer models in Table II costs about $25,000 or less; in all cases, the price includes a "standard" processing unit, at least 4,096 words of core memory and sometimes a teletypewriter. Tables III and IV contain data on the IBM 1800, but only so the calculated price/performance ratios for minicomputers could be compared with the ratios for a "big" computer.

The central processor's job

This section presents and analyzes common CPU features and points out some major design problems encountered by computer manufacturers. Because design trade-offs must be made, users will be better equipped to analyze products if they understand the manufacturers' problems. An "ideal architecture" is not specified in this article—architecture varies with the application. A typical system configuration is shown in Figure 1.

A central processing unit (CPU) contains the circuits to interpret and execute instructions. The CPU performs the following functions:

- Keeps track of the instruction being executed in a "program counter"
- Fetches instructions from memory and interprets or decodes these instructions
- Executes the instructions with the corresponding arithmetic, logic, test, and shift hardware units
- Directs the I/O hardware when I/O instructions are encountered in the program

The CPU may contain registers for computation, address modification, or other purposes, plus error-detecting circuits to catch certain hardware failures. Obviously, the CPU has a direct effect on the speed with which any program can be executed and determines how easily a computer can be programmed.

Reprinted with permission from *Instrum. Technol.* vol 17, pp 67-82, Oct 1970

Table I: Minicomputer makers

Company	Code
Business Information Technology Inc 5 Strathmore Road Natick Mass 01760	BIT
Compiler Systems Inc P O Box 366 Ridgefield Conn 06877	CSI
Computer Automation Inc. 895 W 16th Street Newport Beach, Calif 92660	CAI
Data General Corporation Route 9 Southboro Mass 01772	DG
Data Technology Inc 1050 E Meadow Circle Palo Alto Calif 94303	DT
Datamate Computer Systems Box 310 Big Spring Texas 79720	DCS
Digital Equipment Corporation 146 Main Street Maynard Mass 01754	DEC
Electronic Associates Inc 187 Monmouth Park Hwy West Long Branch N J 07764	EAI
General Automation Inc 706 W Katella Orange Calif 92668	GAI
GRI Computer Corporation 76 Howe Street Newton Mass 02166	GRI
Hewlett Packard Company 1501 Page Mill Road Palo Alto Calif 94304	HP
Honeywell, Inc Computer Control Division Old Connecticut Path Framingham Mass 01701	HON
Information Technology Inc 164 Wolfe Road Sunnyvale Calif 94086	ITI
Interdata Inc 2 Crescent Place Oceanport N J 07757	INT
IRA Systems Inc 332 Second Ave Waltham Mass 02154	IRA
Lockheed Electronics, Inc Data Products Division 6201 E Randolph Street Los Angeles Calif 90022	LEI
Micro Systems, Inc 644 E Young Street Santa Ana Calif 92705	MSI
Motorola Inc Box 5409 Phoenix Arizona 85010	MOT
Philco Ford Inc 3939 Fabian Way Palo Alto Calif 94303	PFI
Raytheon Computer 2700 S Fairview Street Santa Ana Calif 92704	RAY
Redcor Corporation 7800 Deering Ave P O Box 1031 Canoga Park, Calif 91304	RC
Scientific Control Corporation Box 96 Carrollton, Texas 75006	SCC
System Engineering Laboratories, Inc Box 9148 Fort Lauderdale, Florida 33310	SEL
Tempo Computer Inc 340 W Collins Ave Orange Calif 92667	TEM
Varian Data Machines, Inc 2722 Michelson Drive Irvine Calif 92664	VDM
Westinghouse Electric Corporation Computer and Instrumentation Div 1200 West Colonial Drive Orlando, Fla 32804	WES
Xerox Data Systems 701 S Aviation Blvd El Segundo Calif 90245	XDS

Almost all minicomputers now employ a parallel internal processor structure and some form of modified single address instruction word format, however, there is extreme variation in the number of different address modifications and in the relative "power" of individual hardware instructions for given machines.

A computer's word—From an economic standpoint, the computer manufacturer decides on a standard word length for his computer consequently, almost all small computers have a fixed word length and this is true of most computers in this article

The computer word will represent both instructions and data. The determining word-length

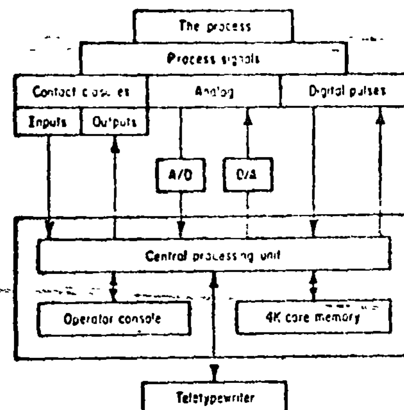


Figure 1 A minicomputer may look small and simple, but its architecture can vary considerably The configuration of the CPU and the size of the memory or memories, is determined by the application.

factor for data representation is usually the arithmetic precision needed by the application, but other factors determine the word length needed for the instructions

The instruction is usually divided into fields, Figure 2, whose size is determined by these factors

- Address field—size based on the maximum address to be referenced directly in an instruction
- Address-mode field—size based on the number of different address modification and special addressing techniques (such as indexing)
- Operation-code field—size based on the number of different basic hardware instructions

Most minicomputers have a 16-bit word length available to provide the capability to indirectly address a maximum 65K of core memory. The largest address that fits into the computer word generally determines the largest core configuration offered for a particular minicomputer. Therefore, word length is directly proportional to the "expandability" of the computer. Experience shows that the 16-bit word is usually of sufficient length to handle the basic single-word fields mentioned previously. Computers with a shorter word length (8 or 12 bits) pay a premium in that many double-word instructions may be needed

For most process applications, the 16-bit word (15 data bits + sign) provides a precision that is acceptable The acceptability of shorter lengths is questionable, because it would be necessary in many cases to resort to double precision arithmetic on the 8- and 12-bit computers. On the other hand, a word length greater than 16 bits is a waste of money for most process applications

Some computers have an 18 bit word length, but usually these computers are 16 bit machines—with 2 bits for extras like "parity checking" and/or "memory protect" The next word length size is generally 24 bits, a size that usually prices the small computer out of the "minicomputer market."

Kinds of addressing—As mentioned above, the kind of address modification employed determines the size of the address mode field in the instruction word. The address modifications usually employed in minicomputers are:

- Relative addressing—The address field of the instruction is added to either the program-location register (floating page concept) or a page register (fixed page concept) to arrive at the effective address
- Indirect addressing—The address field of the instruction holds a "pointer" to the location that contains the address of the operand, this is known as "single-level" indirect addressing. If the indicated location doesn't contain the operand address but does contain another pointer then "multilevel" indirect addressing is in use. Indirect addressing utilizes one extra memory cycle per level of indirectness
- Indexed addressing—The contents of an index register (implemented as either a flip flop register or as a word of core memory) are added to the address field to arrive at the effective address If a core memory location is used for the index register,

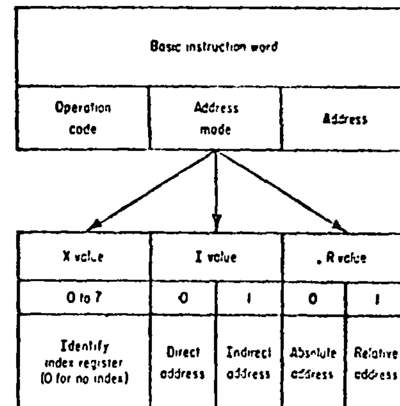


Figure 2 Usually, a basic instruction has three "fields." For a 16-bit word, the address mode's "subfield" can have 3 to 5 bits, then to 11 to 13 bits are available for the other two "fields."

Table IIA: Hardware data for minicomputers

Manufacturer	BIT	CSI	CAI	CAI	DG	DG
Model Number	BIT 483	CSI-16	PDC 208	PDC 216	NOVA	SUPER NOVA
CPU features						
Instruction word length(s)	Variable	16	8/16	16	16	16
Accumulators	1	3	1	2	4	4
Hardware registers	18	11	8	6	5	5
Index registers	0	1 Hardware	0	1 Hardware	2 Hardware 16 Memory	2 Hardware 16 Memory
Bits for operation code	Variable	4	N/A	N/A	N/A	N/A
Bits for address modes	Variable	2	N/A	N/A	N/A	N/A
Address modes	2	3	4	8	4	4
Bits for address	9	10	9	10	10	10
Words directly addressable	512	1K	512	1K	1K	1K
Words indirectly addressable	512	32K	16K	32K	32K	32K
Indirect addressing (levels)	Single	Single	Multiple	Multiple	Multiple	Multiple
Instruction sets						
Store time (full word) μsec	4	2	8	5 33	5 5	1 6
Add time (full word) μsec	Variable	2	8	5 33	5 9	0 8
Fixed point hardware X ÷	No	Standard	No	No	No	Optional
Floating point hardware X/÷	No	Standard	No	No	No	No
Fixed point hardware X time μsec		8 8		-	-	3 8
Fixed point hardware ÷ time μsec	-	9 0			-	6 7
Fixed point software X time μsec	N/A	-	370	55	340	68
Fixed point software ÷ time μsec	N/A		630	95	480	96
Memory						
Memory cycle time μsec	1 0	1 0	2 67	2 67	2 6	0 8
Memory word length bits	8 (multiples)	16	8	16	16	16
Minimum memory size words	1K	1K	4K	4K	4K	4K
Memory increment size words	4K	4K	4K	4K	4K	4K
Maximum memory size words	65K	32K	16K	32K	32K	32K
Parity checking	No	No	No	No	No	No
Memory protect	No	Optional	Optional	Optional	No	Optional
Miscellaneous						
Power failure protect	Standard	Standard	Optional	Optional	Standard	Standard
Automatic restart	Optional	No	Optional	Optional	Standard	Standard
Real time clock	Optional	Optional	Optional	Optional	Standard	Standard
System cost (CPU + 4K memory)	$7,660	$10 750[1]	$5,990	$7 990	$7 950	$11 700

Manufacturer	DT	DCS	DEC	DEC	DEC
Model Number	DT 1600	Datamate 16	PDP 8/I	PDP 8/L	PDP 9/L
CPU features					
Instruction word length(s)	8/16	16	12/24	12/24	18
Accumulators	1	1	1	1	1 Std, 1 Opt
Hardware registers	8	6	N/A	N/A	N/A
Index registers	0	1 Hardware	8 Memory	8 Memory	7 Memory
Bits for operation code	4	5	3	3	4
Bits for address modes	2	3	1	1	1
Address modes	2	8	2	2	2
Bits for address	8	8	8/15	8, 13	13
Words directly addressable	512	256	256	256	4K
Words indirectly addressable	16K	32K	32K	8K	16K
Indirect addressing (levels)	Multiple	Multiple	Single	Single	Single
Instruction sets					
Store time (full word) μsec	24	2	3	3 2	3
Add time (full word) μsec	24	2	3	3 2	3
Fixed point hardware X ÷	No	Standard	Optional	No	Optional
Floating point hardware X/÷	No	No	No	No	No
Fixed point hardware X time μsec	-	7	N/A		16 5
Fixed point hardware ÷ time μsec	-	9	N/A		18
Fixed point software X time μsec	1 200	-	360	360	421
Fixed point software ÷ time μsec	1,500	-	460	460	528
Memory					
Memory cycle time μsec	8 0	1 0	1 5	1 6	1 5
Memory word length bits	8	16	12	12	18
Minimum memory size words	4K	4K	4K	4K	4K
Memory increment size words	4K	4K	4K	4K	4K
Maximum memory size words	16K	32K	32K	8K	16K
Parity checking	No	Optional	Optional	Optional	Optional
Memory protect	Optional	Standard	Standard	Standard	Optional
Miscellaneous					
Power failure protect	Optional	Standard	Optional	Optional	Optional
Automatic restart	Optional	Standard	Optional	Optional	Optional
Real time clock	Optional	Optional	Optional	Optional	Optional
System cost (CPU + 4K memory)	$6 600	$14 900	$12,800	$8 500	$19 900

Notes and symbols

X/÷ multiply/divide
X multiply
÷ divide
N/A not available
K 1 024 words (4K = 4 096 words)

1 Software supplied at extra cost
2 Includes ASR 33 teletypewriter
3 Implied by address
4 8K core minimum
5 Includes 256 words of ROM
6 Includes 512 words of ROM
7 [illegible] 8417
8 Minimum 8K core includes ASR 33
9 Data for P 2000 [illegible] Modified P 2000 [illegible] made in Orlando. Refer to Table I

an additional memory cycle is needed for each indexed instruction. But if a separate flip flop register is used there will be no noticable slowdown in instruction execution time.

If all of the above forms of address modification are available in a minicomputer, the order of execution to calculate the "effective address" is generally

1 Derelativize the address field
2 Fetch the indirect address (repeat for multilevel)
3 Perform the specified indexing

A less frequently applied alternative to the above "post-indexing" scheme is one of "pre indexing." In this latter method the specified indexing is performed prior to fetching of the indirect address.

Computers which do not have the relative addressing capability are fairly common and this is not an important limitation if the application can be handled with an "all core" computer. However, lack of relative addressing rules out the possibility of implementing a core-bulk multiprogramming system with dynamic core allocation, programs for such a machine are not relocatable in core.

Figure 2 illustrates the number of bits needed in each instruction word for the above address modification schemes. Subfields in the address mode field are interpreted as shown in Table II. The relative, indirect, and indexed address modifications can be implemented in 3 to 5 bits (depending on the number of index registers). If only 1 bit is used for the X field shown in Figure 2 the computer can have only one index register. With 3 bits then seven index registers can be identified, which for a small computer in a process application will usually provide sufficient flexibility.

From 11 to 13 bits of a 16 bit word will be available to accommodate both the operation-code field and the address field. This may look like far too few bits to handle both of these fields. However, computer designers often show much ingenuity. One trick reduces the size of the operation-code field by grouping the instructions into two classes those that reference memory and those that do not. Instead of assigning a unique number in the operation-code field for each instruction, only those that reference the memory are given a unique code.

Instructions that do not reference memory have no need for a memory address and can be assigned one operation code number. The address field then codes the specific instructions. Because most instructions are nonmemory reference types, the savings in the size of the operation-code field is dramatic the field can often be reduced to 4 bits.

If only 4 bits are assigned to the operation-code field, 7 to 9 bits remain for the address field. Assume that only three index registers are specified—limiting the size of the X field to 2 bits, therefore, the address field will be set at 8 bits which can address only 256 words directly. The address modification supplied will usually have enough flexibility to overcome this limitation.

Program execution—The CPU has a significant effect on the speed of program execution. Minicomputers with shorter word lengths (8 or 12 bits) must rely extensively on double-word instructions, an extra word of memory and one extra memory read-write cycle for each double-word instruction that is executed in a program. Even though the minicomputer that uses a shorter word length might have a memory read write cycle time which is faster than the time for a corresponding 16 bit machine, program execution will probably be faster in the 16-bit machine because it will execute fewer memory read write cycles for a given program. The CPU in the computer with the shorter word length must work harder to get the same job done.

A trend toward the utilization of multiple, general purpose registers is apparent. These registers in the CPU can be used as accumulators, arithmetic extensions (to multiply divide, etc) and index registers to greatly decrease the execution time required for a given program by reducing the number of references made to memory (such as load and store) this innovation is a definite plus in the evaluation of a new computer's features.

Another feature cuts costs and increases flexibility, many new minicomputers have a "microprogramming" capability. A microprogram is "the sequence of elementary steps which permits the computer hardware to carry out a computer instruction." These elementary steps are called microinstructions.

Some minicomputers provide a minimum set of microinstructions and allow the actual machine architecture to be developed from this basic set. This approach could be of great benefit for an OEM in that it results in a very inexpensive mainframe, but it's probably of little value to a company that must apply minicomputers to industrial control problems. The drawbacks for one of a-kind applications are

- Extra programming effort to define and implement the macroinstructions from sequences of microinstructions
- Inherent loss of computer speed (usually 2 to

Table IIB: Hardware data for minicomputers

Manufacturer	DEC	DEC	EAI	GAI	GAI	[illegible]	GRI	HP	HP	HP	HON	HON	ITI	INT
Model number	PDP-11 20	PDP 15 10	EAI 640	SPC-12	SPC 16	System 8 33	909	2114A	2115A	2116B	H 316	DDP 516	4900	Model 3
CPU features														
Instruction word length(s)	16	18	16/32	8/16	16/32	16 32	16	16	16	16	16/32	16/32	16/32	16/32
Accumulators	6	1 Std, 1 Opt	2	4	16	2	Variable	2	2	2	2	2	8	16
Hardware registers	8	N A	9	8	19	13	9	7	7	7	5	5	16	18
Index registers	6	1 Hardware 8 Memory	1 Hardware	3 Hardware	6	3	32K Memory	0	0	0	1 Hardware	1 Hardware	6 Hardware	15 Hardware
Bits for operation code	4	4	4	8	4	5	0[3]	4	4	4	5	5	8	8
Bits for address modes	4	2	3	3	4	5	2	2	2	2	2	2	2	2
Address modes	12	4	8	6	11	12	4	4	4	4	4	4	4	3
Bits for address	16	12	9/15	12	8/16	8 16	12	10	10	10	9/14	9/14	6/16	6 16
Words directly addressable	32K	4K	512	4K	32K	32K	32K	2K	2K	2K	1K	1K	32K	32K
Words indirectly addressable	-	32K	32K	4K	32K	32K	32K	8K	8K	32K	32K	32K	32K	
Indirect addressing levels	None	Single	Multiple	Single	Single	Single	Single	Multiple	Multiple	Multiple	Multiple	Multiple	Multiple	None
Instruction sets														
Store time (full word), µsec	2 3	1 6	3 3	4 2	2	2 4	3 52	4	4	3 2	3 2	1 92	1 95-3 5	6
Add time (full word), µsec	2 3	1 6	3 3	4 2	3	2 4	1 76 7 03	4	4	3 2	3 2	1 92	1 95 3 5	3 2
Fixed-point hardware multiply/divide	No	Optional	Standard	No	No	Standard	Optional	No	Optional	Optional	Optional	Optional	Optional	Optional
Floating-point hardware multiply/divide	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Fixed-point hardware multiply time, µsec	-	7	18 15	-	-	12	10	-	24	19 2	8 8	5 28	10	23
Fixed-point hardware divide time, µsec	-	7 25	18 98	-	-	13 2	14	-	26	20 8	17 6	10	25	38
Fixed-point software multiply time, µsec	N/A	200	-	N/A	N/A	-	320	187	187	150	258 2	154 6	50	900
Fixed-point software divide time, µsec	N/A	250	-	N/A	N/A	-	600	387	387	310	368 7	220 8	100	1 020
Memory														
Memory cycle time, µsec	1 2	0 8	1 65	2 0	0 96	0 96	1 76	2 0	2 0	1 6	1 6	0 96	0 96 1 76	1 5
Memory word length, bits	16	18	16	8	16	16 + 2	16 + 2	16	16	16	16	16	16	16
Minimum memory size, words	4K	4K	4K	4K	4K	4K	4K	4K	4K	8K	4K	4K	4K	2K
Memory increment size, words	4K	4K	4K	4K	4K	4K	4K	4K	4K	8K	4K	4K	4K	2K
Maximum memory size, words	32K	131K	32K	16K	32K	32K	32K	8K	8K	32K	16K	32K	32K	32K
Parity checking	No	Optional	No	Optional	No	Standard	Standard	Optional	Optional	Optional	No	Optional	Optional	Optional
Memory protect	No	Optional	Standard	No	No	Standard	Standard	No	Optional	Optional	No	Optional	Optional	Optional
Miscellaneous														
Power fail protect	Standard	Optional	Standard	Optional	Standard	Standard	Standard	Optional	Optional	Standard	Standard	Standard	Optional	Optional
Automatic restart	Standard	Optional	No	Optional	Standard	Standard	Standard	Optional	Optional	Optional	Optional	Optional	Optional	Optional
Real-time clock	Optional	Optional	Optional	Standard	Standard	Standard	Optional	Optional	Optional	Optional	Optional	Optional	Optional	Optional
System cost (CPU + 4K memory)	$10 800[2]	$15,600	$26,500	$5,000	$10 000	$ 8 [illegible]	$9.740	$9 950	$14 500	$24,000[4]	$9,700	$23 800	$9,950	$10 800

3 times slower than a computer with a fixed instruction repertoire and a comparable memory read-write cycle time)

Whether the initial savings in purchase price for this kind of machine can offset the added programming expense is debatable

The problem of evaluating the instruction sets of various computers is very complex The great majority of new minicomputers are being programmed in assembly language, so the prospective user should take a close look at the instruction sets of machines that are of special interest. The instruction sets in Table II run the full range from very simple microinstructions sets to large, powerful instruction sets

Because of cost most small computers are now programmed in assembly languages The money saved by buying a minicomputer is easily lost if all the extra peripheral hardware needed to compile programs efficiently is added to the system An economical system that is programmable in a higher level language has yet to be introduced As a reasonable alternative to programming in assembly language, programs can be written in a compiler level language (the compilations are run on a large machine) This approach shows promise and should become more popular in the future Most manufacturers do not offer this alternative now, therefore, the evaluation of hardware instruction sets is still important

Particular attributes needed in the instruction set will vary with the application If many arithmetic calculations are necessary, a hardware multiply/divide package is desirable If the application is primarily logic oriented then an ideal computer would have strong capabilities in this area (such as bit manipulation and bit testing)

Miscellaneous features—It is important to note which features are standard and which are optional The manufacturer that offers the least expensive mainframe usually does not offer the least expensive system In real-time applications for example, power failure protection and a real-time clock are highly desirable and are specified for most systems The manufacturer that offers these features as standard equipment is usually ahead of the game, his price will probably be lower

Evaluating memories

The magnetic core memory, introduced in the early 1950's, is still today's most common, high speed random access, read-write memory device All minicomputers in this survey rely on this kind of memory for primary storage

Core memories range in size from a few hundred to several million bits with cycle times from about 0 5 to 10 µsec The attributes that made the magnetic core memory so popular are reasonably large capacity, high speed and acceptable cost.

Table II provides comparative specifications on the memory modules with reference to the definitions that follow The "cycle time" referred to above is the memory read-write cycle time in other words, the time required to transfer one word of binary data from core memory to the CPU, and to replace the same word in core "Memory word length" is the number of bits accessed in one memory cycle this term is usually synonymous with the "fixed word length" described in detail earlier The "memory size" is given as number of words consequently, "memory size" must be multiplied by the "memory word length" to obtain the total number of bits contained in the memory. This latter figure is important because it truly represents the storage capacity of a given machine

Magnetic cores ranging from 100 to 10 mils (thousandths of an inch) in diameter have been made and used The most common sizes are 50, 30 and 20 mils—in spite of their size, two to four wires are threaded through each core, usually by hand. Wiring of the cores is probably the greatest obstacle to cheaper faster, and larger memories

Individual core size is the factor that has the biggest effect on memory cycle time For example, computers with 50-mil core memories generally have cycle times ranging from 6 to 10 µsec. Similarly the 30-mil core usually results in cycle times of 2 to 5 µsec and the 20 mil core performance ranges from 500 nsec to 2 µsec

An important point to remember the sense threshold of the signals (to read from and write into the core memory) will vary in a magnitude that is proportional to the size of the individual cores. if a signal level of 1 unit is necessary to read a 50 mil core, then a 20 mil core will require only about 0 5 units of signal strength It's obvious that the smaller the core (the faster the memory) the more susceptible the machine will be to noise-induced computer failures such as dropped bits

Because of the noise sensitivity problem the industrial user should be wary of a computer that is too fast for the application Reliability may be lower and cost will probably be higher

Table IIC: Hardware data for minicomputers

Manufacturer	INT	IRA	LEI	MSI	MSI	MOT
Model number	Model 4	SPIRAS 65	MAC 16	800	910	MDP 1000
CPU features						
Instruction word length(s)	16/32	16/32	16	16	16 24	12
Accumulators	16	2	1	15	2	5
Hardware registers	33	3	6	23	5	9
Index registers	15 Hardware	1 Hardware + Accumulators	4 Memory	0	1 Hardware	3 Hardware
Bits for operation code	8	6	4	4	5	8
Bits for address modes	2	2/3	3	0	3	3
Address modes	3	4	8	1	8	6
Bits for address	6/16	10/16	9	8	8 16	12
Words directly addressable	32K	65K	512	256	32K	4K
Words indirectly addressable		16K	65K	-	255	4K
Indirect addressing (levels)	None	Multiple	Multiple	None	Single	Single
Instruction sets						
Store time (full word), µsec	6	3.6	2	1.1 (8 bit)	4.62 (16 bit)	6.48
Add time (full word), µsec	3.2	3.6	2	1.22	5.05	4.32
Fixed point hardware ×, ÷	Optional	Standard	Optional	No	Standard	No
Floating point hardware ×, ÷	No	No	No	No	No	No
Fixed point hardware × time, µsec	23	17	9		63.35	
Fixed point hardware ÷ time, µsec	38	30	12		90.86	
Fixed point software × time, µsec	900	-	150	N/A		N/A
Fixed point software ÷ time, µsec	1 020	-	300	N/A	-	N/A
Memory						
Memory cycle time, µsec	1.0	1.8	1.0	1.1	1.1	2.0
Memory word length, bits	16	16	16	8 9 or 10	8 or 10	8
Minimum memory size, words	2K	4K	4K	256 (ROM)	4K - 512 (ROM)	4K
Memory increment size, words	2K	4K	4K	4K	4K	4K
Maximum memory size, words	32K	65K	65K	32K	32K	16K
Parity check	Optional	No	Optional	Optional	Optional	No
Memory protect	Optional	Optional	Optional	Optional	Optional	No
Miscellaneous						
Power failure protect	Optional	Optional	Standard	Optional	Optional	Optional
Automatic restart	Optional	Optional	Standard	Optional	Optional	Optional
Real time clock	Optional	Optional	Standard	Optional	Optional	Optional
System cost (CPU + 4K memory)	$13 800	$12,750	$11 950[2]	$5,700[5]	$6 900[6]	$8,500

Manufacturer	PFI	RAY	RAY	RC	SCC	SEL	SEL	TEM
Model number	1216 F	703	706	RC-70	4700	810A	6108	TEMPO 1
CPU features								
Instruction word length(s)	16	16	16	16/32	16/32	16	16	16/32
Accumulators	1	1	1	1	3	2	2	2
Hardware registers	7	6	6	5	10	2	2	7
Index registers	1 Hardware	1 Hardware	1 Hardware	1 Hardware	1 Hardware	1 Hardware	2 Hardware	1 Hardware
Bits for operation code	3	4	4	6	4/9	4	4	4
Bits for address modes	2	1	1	3	2	2	2	3
Address modes	4	2	2	5	5	4	4	8
Bits for address	11	11	11	7/14	9/16	10	10	9/16
Words directly addressable	2K	32K	32K	16K	32K	1K	1K	512
Words indirectly addressable	32K			16K	65K	32K	32K	65K
Indirect addressing (levels)	Single	None	None	Single	Single	Multiple	Multiple	Multiple
Instruction sets								
Store time (full word), µsec	4	3.5	1.8	1.9	1.84	3.5	1.5	1.0
Add time (full word), µsec	4	3.5	1.8	1.9	1.84	3.5	1.5	1.8
Fixed point hardware ×, ÷	Optional	Optional	Optional	Standard	Optional	Standard	Standard	Optional
Floating point hardware ×, ÷	No	No	No	No	No	No	No	No
Fixed point hardware × time, µsec	N/A	17.5	9	6.5	6.44	7	4.5	7
Fixed point hardware ÷ time, µsec	N/A	24	9	12.5	6.90	10.5	8.25	9
Fixed point software × time, µsec	N/A	147	75	-	N/A		-	N/A
Fixed point software ÷ time, µsec	N/A	300	154	-	N/A		-	N/A
Memory								
Memory cycle time, µsec	3.5	1.75	0.9	0.86	0.92	1.75	0.75	0.9
Memory word length, bits	16	16	16	16 + 1	16	16	16	16
Minimum memory size, words	4K	4K	4K	4K	4K	4K	8K	4K
Memory increment size, words	4K	4K	4K	4K	4K	4K	8K	4K
Maximum memory size, words	16K	32K	32K	32K	65K	32K	32K	65K
Parity check	Optional	No	Optional	Standard	Optional	Optional	Standard	Optional
Memory protect	Optional	No	Optional	Standard	Optional	Optional	Optional	Optional
Miscellaneous								
Power failure protect	Optional	Optional	Optional	Standard	Standard	Standard	Standard	Standard
Automatic restart	Optional	Standard	Standard	Standard	Optional	Optional	Optional	Optional
Real time clock	Optional	Optional	Optional	Standard	Optional	Optional	Optional	Optional
System cost (CPU + 4K memory)	$11,900	$12 750[2]	$19,000[2]	$13,900[7]	$14,800	$18 000[2]	$33,500[8]	$15 000[2]

As previously mentioned, some computers utilize extra bits in each word of core memory for such things as "parity checking" and "memory protect." This feature is usually optional in all but the more expensive small computers. Because of the cost of this feature and the improved reliability of core memories, most computer manufacturers have decided that the extra checking is not warranted. Some manufacturers who feel strongly about this don't offer parity or memory protect as options.

However, some form of parity check is mandatory for a bulk memory device (drum or disc), and memory protect is needed if any form of on line debugging is anticipated. On-line debugging indicates correction of new programs after the basic system has been installed and debugged. An operational system could be impaired by new programs if it's without a memory protect feature.

If a new minicomputer system is small enough to be implemented on an "all core" machine, parity and memory protect are not warranted; when talking industrial control applications of minicomputers, this may be the usual case.

The popular ROM

Read only memory has become very popular during the past five years. The ROM is a medium for storing data in permanent (nonerasable) form. Usually a high speed static storage device—excluding paper tape or other such memories—the ROM provides rapid access to information of a permanent nature. Common square root and exponential routines can be "wired-in" for fast execution from conventional core memory operations. From a programmer's point of view, the result is indistinguishable.

This concept lends itself in a natural way to microprogramming in which every instruction is a permanent subroutine in a ROM. Microprogramming provides economical realization of complex instruction sets in simple computers, and is the main reason for the rise in popularity of the ROM.

If every instruction is a wired-in subroutine in a ROM, a different set of computer instructions may be obtained by changing the ROM. This notion leads to emulation (executing instructions of one computer on the hardware of another computer), a concept applied extensively in the various IBM 360 and RCA spectra 70 computers.

In the past, ROMs have been wired at the factory to contain programs that were specified previously. After shipment to the field, these ROMs were practically impossible to modify. A new design recently introduced by Digital Scientific Corp. (San Diego, Calif.) changes all this; their new ROM allows field programming by plugging "chips" into a ROM board. This ROM can be reprogrammed at any time by the user and forms the base for Digital Scientific's new computer; their META 4 (not in this survey) is unique in that it does not have a fixed instruction set of its own, but it can emulate almost any other computer. This example is cited to emphasize the utility of the ROM.

The ROM provides a very powerful tool for the computer designer. Some strong arguments generally advanced in favor of the ROM are:

- Economy—costs much less than standard read-write core memory
- Speed—typically 5 to 10 times faster than read-write core
- Reliability—the program is protected from overwriting by an errant program

A number of the small computers employ ROMs in varying degrees; this is an important factor and should be considered for specific applications.

Bulk memories for the mini

The two most popular and economical bulk memory devices of the nonvolatile type are the magnetic disc and the magnetic tape. The magnetic drum is in a class by itself because it's considerably more expensive (per unit of storage) to implement. Small, inexpensive systems are being discussed; therefore, magnetic drums will not be covered.

Magnetic discs—Two general classes are available:

- discs with a single floating read-write head
- and those with a fixed read-write head for each track

The first disc memories had only one read-write head which (in multiple disc installations) had to "seek" the correct disc surface and then the correct track. Although average access time was very slow, this type of bulk storage proved economical and was popular with many early computer users.

The design was improved by adding a separate read-write head for each recording surface. Popular today, this kind of disc pack cut the average memory access time to about 100 milliseconds; furthermore, bit transfer rates greater than 1 MHz are now common. The cost of disc bulk memory is only about 2 cents per byte (8 bits), and this cost decreases as the size of the bulk memory increases.

Almost all of the 45 minicomputers in Table II can be equipped with a magnetic disc bulk memory system. But many of the manufacturers who will supply a disc do not (as yet) offer a

Table IID: Hardware data for minicomputers

Manufacturer	VDM	VDM	WES[9]	XDS	XDS	XDS
Model number	520/i	620 i	Prodac 2000	CE 16	CF 16	SIGMA 3
CPU features						
Instruction word length(s)	8 16	16/32	16	16	16	16/32
Accumulators	7	2	4 Memory	1	1	2
Hardware registers	7	6	5	6	6	9
Index registers	1 Hardware	2 Hardware	2 Memory	1 Hardware	1 Hardware	2 Hardware
Bits for operation code	3	4	5	5	5	4
Bits for address modes	3	3	3	3	3	4
Addressing modes	5	4	8	8	8	12
Bits for address	15	9/11	8/16	8/14	8/14	8
Words directly addressable	4K	2K	256	256	256	1K
Words indirectly addressable	32K	32K	65K	16K	16K	65K
Indirect addressing (levels)	Multiple	Multiple	Single	Multiple	Multiple	Single
Instruction sets						
Store time (full word), µsec	4 5	3 6	7	16	5 34	1 95
Add time (full word), µsec	4 5	3.6	7	16	5 34	1 95
Fixed point hardware X, ÷	No	Optional	Standard	No	No	Optional
Floating point hardware X/÷	No	No	Optional	No	No	No
Fixed point hardware X time, µsec	-	10	24 5	-	-	7 8
Fixed point hardware ÷ time, µsec	-	14	32 5			8 13
Fixed point software X time, µsec	N/A	200	-	126	42	N/A
Fixed point software ÷ time, µsec	N/A	200	-	142	47 3	N/A
Memory						
Minimum cycle time, µsec	1 5	1 8	3 0	8 0	2 67	0 98
Memory word length, bits	8	16	16	16	16	16
Minimum memory size, words	4K	4K	4K	4K	4K	8K
Memory increment size, words	4K	4K	4K	4K	4K	8K
Maximum memory size, words	32K	32K	65K	16K	16K	64K
Parity checking	Optional	Optional	Standard	No	No	Standard
Memory protect	Standard	Optional	No	No	No	Optional
Miscellaneous						
Power fail-safe protect	Optional	Optional	Standard	Optional	Optional	Optional
Automatic restart	Optional	Optional	Optional	Optional	Optional	Optional
Real time clock	Optional	Optional	Standard	Optional	Optional	Optional
System cost (CPU + 4K memory)	$7,500	$9 950	$10 000	$9,980	$7,990	$24 000

disc-operating system with their software package.

The latest innovation in disc packs is the "fixed-head" disc which has one read-write head per track. Using this approach the only factor that determines access time is the speed of disc rotation. Average access time will be the time required for one-half rotation (an average time of about 15 milliseconds is typical). Data transfer rates vary considerably from model to model but all are relatively fast transfer devices. A typical fixed-head disc would transfer at least 150K bytes sec, a bit transfer rate of 1.2 MHz. The XDS model 7211 7212 has an extremely fast bit transfer rate, 24 MHz. The price per byte on these units is typically 3 to 3.5 cents, or about 60 percent above the floating head units.

The system designer faces a choice between the high speed, high cost, fixed-head disc and the slower but more economical floating-head disc. For most process control and data acquisition installations, the added expense for the fixed-head units probably will not be warranted.

Magnetic tape—If high speed data storage or retrieval is not a major factor then magnetic tape should be considered. As a bulk storage device, magnetic tape has several strong points. First, there is no upper limit on storage capacity (extra tape reels can be purchased). The capacity of a common 7-track 2,400-ft tape varies from 5M to 20M bytes, which is greater than the capacity of most disc packs. Second, the cost per byte of storage is much lower than for any other bulk storage medium.

Magnetic tape must be read serially, a shortcoming which means the average access time for a piece of information is measured in seconds rather than milliseconds. Of course, if tape must be changed to access the information, then the access time extends to minutes instead of seconds.

Magnetic tape is ideal for a system that faces extensive program development. When combined with a keyboard data entry unit and a crt display, the magnetic tape is a very powerful system development tool. DEC has a good example of this type of system in their LINC-8 and PDP-12 computers. The magnetic tape provides a convenient means, for storing programs under development (much more so than cards or paper tape) and the keyboard crt combination doesn't generate mountains of paper. The program can be written and debugged with a minimum of "hard copy" output.

Another magnetic tape method has become popular recently, the "key-to-tape" unit allows the magnetic tape to replace the punched card without changing drastically the keypunch girl's duties. She still transfers information from the coding sheet to a keyboard, but the information is stored in a tape reel instead of punched cards. Once on tape, the information is of a more permanent nature, less likely to be destroyed, and more convenient to handle, furthermore, it isn't much more expensive than the old keypunch method.

The "cassette" tape housing, physically similar to units designed for stereo tape players, is becoming more popular. The additional ease of handling and storing cassettes is obvious and the tape sees less exposure to atmospheric impurities and dirt particles. The development of cassette tapes for future computer systems will be something to watch.

Interfaces to the mini

The equipment needed for conversation with the minicomputer should satisfy certain requirements:

- Equipment should not be expensive; nobody wants to pay $35,000 for a line printer that will be connected to a $10 000 computer.
- The equipment should be relatively fast, convenient to use, and reliable.

The standard interface of the minicomputer industry is probably the Teletype ASR-33. This versatile machine provides keyboard input, printed output and paper tape I O—and all at relatively low cost of about $1,500. The disadvantages of this teletypewriter are well known: low speed printed output (about 10 characters/sec), low speed paper tape I/O, and paper tape itself which is unquestionably the least convenient bulk storage medium to work with.

Alternatives to paper tape are punched cards and magnetic tape. Cards are convenient to work with and easier to store than tape but unfortunately the card handling equipment (readers and punchers) is very expensive when compared to paper tape equipment. Historically, magnetic tape has been even more expensive, but it appears that low cost cassette equipment will soon be competitive with paper tape. The added convenience of the magnetic tape more than offsets the added expense.

When magnetic tape is combined with a simple keyboard input device and a crt display, the system is very powerful. We are not far from the day when this kind of system will be much lower in cost, already available is the IRA Systems MASCOPY, a unique keyboard-input crt peripheral that sells for under $6,000.

The only communication device missing from the above system (keyboard, crt and magnetic tape) is some kind of a printer, of course, the old teletype is always available with its low speed handicap. Current printers include several moderate-to-high speed models. Teletype's INKTRONIC terminal is now available as a receive-only model, a moderately high speed printer without moving parts in its print mechanism. Charged particles of ink shoot toward ordinary teletype paper at a rate of 105 characters/sec. (about 10 times faster than the ASR-33). This rate produces about 60 lines/min.

Also available are several high speed, relatively low-cost printers such as the Clevite 4800 which prints 4800 (80 character) lines/min, and the Litton Industries DATALOG MC-8800 which prints 6 000 (85 character) lines/min. These small printers appear to be ideal for the small computer system, in addition, they would eliminate the long time required to get hard copy output from a teletypewriter.

Input/output control

Programmed I/O is a term that can describe the type of I/O operation in which the data (input to or output from the computer) must pass through an internal register of the CPU. Usually the accumulator is the register through which the data passes. This means that the computer must be dedicated entirely to the input or output operation throughout its complete cycle—a big drawback.

This method of I/O control requires the least extra hardware, because it's the least expensive to implement it's in most minicomputers. Programmed I/O is definitely a drawback if the application requires a large amount of I O. But if the application needs infrequent I O and only in short "bursts," then programmed I O is an excellent means of saving money.

Most minicomputers with programmed I O will provide only one interrupt with the basic system. If more than one peripheral device is to be connected to this computer, the devices must be connected in "party-line" fashion first come, first served—there is no priority other than the checks by the interrupt handling software to see which device is making the interrupt request.

Minicomputers which provide only this means of I/O control are

Computer Automation	PDC-205
Hewlett-Packard	2114A
Data Technology	DT-1600
Xerox Data Systems	CE-16, CF-16

These computers offer no option to the programmed I/O method.

Direct memory access—The direct memory access (DMA) channel is a popular alternative to programmed I O, but DMA requires more hardware so it's more expensive. Most minicomputers with this feature offer it as an option (not included in the basic price). The DMA channel enables an I/O data transfer to be initiated under program control and then carries out the transfer independent of the CPU. The data path is usually to from the peripheral device directly from/to the core memory.

Completion of the I/O transfer operation is then signaled by a priority interrupt

Actually, minicomputers don't really carry out the I/O transfers with complete independence from the CPU—it just looks that way. The computers utilize a scheme called "cycle stealing" in which the I/O processor effects the data transfer, usually on a word-by-word basis while the CPU cycle is delayed. The "cycle stealing" method does not impair a computer's ability to execute other programs while the I/O transfer is proceeding. It only means the program will not be executed at full speed, even this is a vast improvement over programmed I/O in which no programs can be executed during the I/O operation.

Of the computers in this survey, the following offer DMA as standard equipment.

Business Information Technology	BIT-483
Compiler Systems	CSI-16
Data General	NOVA
Datamate Computers	Datamate-16
Digital Equipment	PDP 8/L-11, and-15/10
General Automation	SYSTEM 18/30 and SPC-16
Lockheed Electronics	MAC 16
Raytheon Computer	703 706
Xerox Data Systems	SIGMA-3

All other computers may be assumed to have DMA available on an optional basis

Software for the minicomputer

In general the software currently offered is minimal. A few manufacturers have fairly comprehensive software packages but they are definitely in the minority. Perhaps one reason for this situation is that the computer firms simply have not had enough time to develop software packages for their new computers. This condition is certainly plausible because most minicomputers now on the market have been introduced during the past 18 months

Predictably, the computers with the most software are products of the older computer companies, the companies in this category are Digital Equipment Corp, EAI, Hewlett Packard, Honeywell, Raytheon Systems Engineering Labs and Xerox Data Systems. All of these mini manufacturers offer a basic software package that usually consists of a compiler, assembler, utility routines math library and an operating system (including I/O driver routines). In addition, most have libraries of application oriented programs which can generally be adapted to specific applications. DEC appears to have the lead in this field with a library of applications programs that's available through their DECUS organization. Of course, many companies are happy to provide programming assistance to a user, sometimes at not-so reasonable rates.

Other companies intend to offer a comprehensive software package but do not have their package completed yet. Most of these firms have the same package as those previously mentioned—with the exception of an operating system, this second group includes Business Information Technology, Datamate Computer, General Automation, IRA Systems, Lockheed Electronics, Redcor, Tempo Computer, Westinghouse Electric and Data General.

All other minicomputers in this survey are made by a third category of vendors who can be characterized as offering an assembler, a loader, some utility routines, but not much else. These computers are intended for applications that require a small dedicated machine, one in which the programs are short and easily written in assembly language, and one in which the programs will not be changed. Naturally the computers in this third group are usually the least expensive.

Several software packages have been developed for industrial data acquisition and process control. In particular, two packages rival the IBM-1800 PROSPRO system in power; the DEC INDAC-8 and the Foxboro CONPAC BATCH LAM are both written for the DEC PDP-8 family of computers and both require considerably more hardware than that found in a "minimum system." Another software system that is in roughly the same class in the SEL CONTRAS package.

In general, systems that utilize these software packages will be priced well out of the "minicomputer" class. A minimum configuration to implement any of these packages usually starts at about $50,000 with many configurations priced well above $100,000.

One trend in the computer industry—and the minicomputer field is no exception—is the move toward separate pricing of hardware and software. The following companies now charge separately for software

Compiler Systems
General Automation
Xerox Data Systems

Sooner or later most other manufacturers will probably follow this "unbundling" trend.

Reliability and service

Another point to keep in mind when evaluating minicomputers for a specific application in many cases, high reliability is more important than high speed. As previously mentioned usually the faster machines have smaller cores in their memories and are more susceptible to electrical noise than the slower machines with the larger cores. Of utmost importance to the computer user who plans a real time application is the system's reliability.

The computer manufacturers are all very optimistic about product reliability. Many are so optimistic that they informally quote MTBF figures of

Table III: Parameter values for calculation of price/performance ratios

			Hardware parameters										Software parameters							
Mfr	Model	Cost (K$)	A_n (%)	t_h (%)	t_s (%)	M (K words)	E (K words)	W (bits)	R	T (μsec)	N	P_h	D	B	L	A	C	S	P_s	P
BIT	BIT 483	7.66	60	50	50	32	1	8	1	1.0	1	1.49	0	1	1	3	1	25	1.06	1.27
CSI	CSI 16	12.95	100	0	90	64	10	16	3	1.0	1	1.40	1	1	1	1	3	30	1.30	1.16
CAI	PDC 208	5.99	50	50	70	32	1	8	1	2.67	0	1.92	1	1	1	1	0	0	2.39	2.16
CAI	PDC 216	7.99	50	50	70	64	10	16	2	2.67	0	1.21	1	1	1	1	0	0	2.83	1.02
DG	NOVA	7.95	60	65	70	64	10	16	4	2.6	3	1.10	1	1	1	1	0	0	3.10	2.14
DG	SUPER NOVA	11.7	70	65	70	64	10	16	4	0.8	3	1.04	1	1	1	1	0	0	4.68	2.06
DT	DT 1600	6.6	30	60	50	32	0	8	1	6.0	0	3.30	1	1	1	1	0	0	2.64	2.97
DCS	Datamate 16	14.9	90	50	85	64	8	16	2	1.0	3	1.83	1	1	1	1	1	0	3.31	2.42
DEC	PDP 8/I	12.8	60	65	70	48	8	12	1	1.5	1	1.90	1	1	1	4	5	75	0.66	1.28
DEC	PDP 8/L	8.5	60	65	70	48	8	12	1	1.6	1	1.29	1	1	1	4	5	75	0.44	0.87
DEC	PDP 9/L	19.9	55	65	70	72	13	18	1	1.5	0	2.27	1	1	1	4	5	75	1.03	1.08
DEC	PDP 11/20	9.30	60	90	85	64	0	18	6	1.2	2	1.22	1	1	1	1	0	0	3.72	2.47
DEC	PDP 15/10	15.6	65	65	70	72	12	18	1	0.8	0	1.41	1	1	1	4	5	75	0.81	1.11
EAI	EAI 640	26.5	90	85	85	64	9	16	2	1.65	2	3.14	1	1	1	2	1	90	2.66	2.89
GAI	SPC 12	5.0	50	65	70	32	4	8	4	2.0	1	1.10	1	1	1	1	0	80	0.76	0.86
GAI	SPC 16	10.0	60	90	80	64	8	16	16	0.96	3	0.84	1	1	1	2	1	80	1.08	0.89
GAI	System 18/30	18.0	90	65	70	72	8	18	2	0.96	5	1.72	1	1	1	1	1	90	2.0	1.00
GRI	909	9.74	50	60	70	72	12	18	1	1.76	4	1.18	1	1	1	1	0	0	3.09	2.62
HP	2114A	9.95	60	70	80	64	10	16	2	2.0	0	1.34	1	1	1	1	1	90	1.11	1.22
HP	2115A	14.5	65	70	80	64	10	16	2	2.0	0	1.95	1	1	1	1	1	90	1.01	1.76
HP	2116B	24.0	70	70	80	128	10	16	2	1.6	1	1.80	1	1	1	1	1	90	2.67	2.23
HON	H 316	9.7	65	70	80	64	9	16	2	1.6	1	1.23	1	1	1	2	2	90	0.81	1.02
HON	DDP 516	23.8	70	70	80	64	9	16	2	0.96	1	2.43	1	1	1	2	2	90	1.97	2.20
ITI	4900	9.95	75	70	75	64	6	16	8	1.75	0	1.30	1	1	1	1	0	0	3.10	2.27
INT	Model 3	10.8	60	60	75	64	6	16	16	1.5	0	1.38	1	1	1	1	1	50	1.64	1.46
INT	Model 4	11.8	60	60	75	64	6	16	16	1.0	0	1.62	1	1	1	1	1	50	1.97	1.76
IRA	SI RAS 65	12.75	90	75	75	64	10	16	2	1.8	0	1.8	1	1	1	1	1	50	1.62	1.71
LEI	MAC 16	11.95	50	65	85	64	9	16	1	1.0	3	1.31	1	1	1	1	1	50	1.71	1.61
MSI	800	5.7	50	50	50	32	0	8	15	1.1	0	1.12	1	0	1	1	0	0	2.88	1.88
MSI	810	6.9	75	60	60	32	0	8	2	1.1	0	1.32	1	1	1	1	0	30	1.73	1.62
MOT	MDP 1000	8.5	50	65	55	32	4	8	5	2.0	0	1.86	1	1	1	1	0	30	2.12	2.03
PFI	1216 F	11.9	50	100	65	64	11	16	1	3.5	0	1.78	1	1	1	1	0	0	4.78	3.27
RAY	703	11.25	50	70	80	64	11	16	1	1.75	1	1.44	1	1	1	2	3	75	0.86	1.14
RAY	706	17.5	55	70	80	64	11	16	1	0.9	1	1.73	1	1	1	2	4	90	1.09	1.41
RC	RC 70	13.9	90	80	80	68	7	17	1	0.86	5	1.28	1	1	1	1	1	50	1.06	1.61
SCC	4700	14.8	75	65	70	64	9	16	3	0.92	1	1.61	1	1	1	1	3	70	1.23	1.37
SEL	810A	16.0	90	80	85	64	10	16	2	1.75	1	2.16	1	1	1	1	1	50	2.07	2.37
SEL	810B	32.0	90	80	85	128	10	16	2	0.75	2	1.83	1	1	1	1	1	90	3.66	2.69
TEM	TEMPO 1	13.5	50	60	65	64	9	16	2	0.9	1	1.49	1	1	1	1	1	50	1.03	1.71
VDM	520/i	7.6	50	60	60	32	0	8	7	1.5	1	1.74	1	1	1	1	0	0	3.0	2.37
VDM	620/i	9.95	65	60	60	64	9	16	2	1.8	0	1.39	1	1	1	1	1	0	2.21	1.80
WES	Prodac 2000	10.0	90	60	65	64	8	16	4	3.0	3	1.49	1	1	1	1	1	60	1.33	1.41
XDS	CF 16	9.99	50	65	70	64	8	16	1	8.0	0	1.08	1	1	1	1	0	0	3.99	2.03
XDS	C 16	7.99	50	65	75	64	9	16	1	2.67	0	1.27	1	1	1	1	0	0	3.19	2.23
XDS	SIGMA 3	21.0	90	85	90	72	8	16	2	0.98	1	2.0	1	1	1	1	1	95	2.69	2.29
IBM	1800	45.88	90	80	100	72	8	16	6	4.0	5	0.12	1	1	1	2	3	100	3.17	4.64

about 20,000 hours, and most do not offer hardware parity checking because (they say) the memories are so reliable that the extra cost is not warranted. Even if the computer manufacturers are right one should consider hardware parity checking as a very desirable option when an auxiliary rotating memory is to be used with the minicomputer.

When the minicomputer's popularity rose in 1968, the trend was toward selling hardware. The PDP-8/I was introduced with little or no software, the first buyers knew that they would have to develop most of their own software. As the number of minicomputer manufacturers multiplied, the competition grew tougher, and most manufacturers offered more software with their machines. Presentation of other services to minicomputer purchasers followed a similar evolution.

Traditionally, service has been hard to get for small computers. The original approach (of most manufacturers) was to sell the concept of "all you have to do is find the board that is bad and replace it." Only 5 to 10 boards are in most minicomputers, so this doesn't sound unrealistic until one remembers that the minicomputer isn't the only piece of hardware in the system. After the users realized this (which didn't take too long) they demanded and are starting to receive more service.

Several manufacturers (Honeywell, DEC, and Hewlett-Packard) already had extensive service organizations, but most of the others had to develop service groups. Some vendors contracted larger computer companies to service their systems. For example, RCA is servicing General Automation computers, and Control Data furnishes service for Computer Automation.

Most manufacturers have been and still are unwilling to rent or lease their minicomputers, they will consider only an outright purchase or sometimes a third party arrangement. A few exceptions are General Automation, Honeywell, Raytheon, Redcor, and SEL. XDS will rent or lease the sigma 3 but not the CF 16 or CE 16 minicomputers.

Criteria for rating minicomputers

A term that is often mentioned but not well defined is "price to performance ratio." This number (if accurate) could tell a user which computer will furnish the most "performance" for his money. A complete and accurate definition of a price/performance ratio for a computer is impossible, however usable ratios can be calculated. In this section, equations are presented for three criteria:

- P_h—hardware price/performance
- P_s—software price/performance
- P—total price/performance (the average of P_h plus P_s)

Hardware performance—The factors that affect the hardware performance of a computer fall into several categories. First, certain variables are fixed (based on the manufacturer's specifications) for a given computer model. These parameters are constraints

M = Core memory storage capacity of a basic machine, total bits
F = Number of bits in the address field of single word instructions
W = Word length, bits
R = Number of general purpose registers
T = Core memory read-write cycle time
N = Number of "extras" in the basic cost of the machine, including
- Real-time clock
- Power failure protection
- Automatic restart after power failure
- Memory parity checking
- Memory protect

Other parameters also affect a computer's hardware performance but are not as easy to arrive at as the above. Arithmetic, logic and I/O capabilities must reflect some degree of opinion, to minimize opinion, the following guidelines are suggested

A_h = A number proportional to the arithmetic capability of the computer—with a range of 0 to 100.
- 0 No arithmetic capability
- 25 Hardware add and complement
- 50 Hardware add and subtract, software multiply and divide (fixed point, slow)
- 75 Hardware add and subtract, hardware multiply and divide (fixed point, fast)
- 90 Hardware add and subtract, hardware multiple and divide (fixed point), software floating point arithmetic
- 100 Hardware fixed point and floating point arithmetic

L_h = A number proportional to the logic capability of the computer—range of 0 to 100:
- 0 No logic capability
- 25 "And" and "or" hardware
- 50 "And" "or" and "exclusive or"
- 75 All of the above, also word test and conditional branch instructions
- 90 All of the above also bit test and bit manipulation instructions
- 100 All of above also arithmetic rational test instructions

I_h = A number proportional to the I/O capability of the computer—range of 0 to 100.
- 0 No I/O
- 25 Programmed I/O through internal registers only
- 75 Same as above, also DMA standard
- 100 All of above, also multiple I/O processors

The principal factors that affect computer performance can be used in empirical equations for the calculation of "price to performance" ratios. The hardware parameters, M to N and A_h to I_h, are in this equation for calculation of P_h, the hardware-price/performance ratio:

$$P_h = \textit{Basic system cost } (\$) - \left\{ 0.1M\left[1-\left(\frac{W-F}{2W}\right)\right] + \frac{20}{T}(A_h+L_h+I_h) + 100N + 50R \right\} \quad (1)$$

The variables are weighted in proportion to their relative importance to system performance. Equation 1 is structured so that a "good" computer will have a P_h value of about unity, the bigger the number, the less performance per dollar. The equation for P_h does not contain factors for operating systems or compilers, however, the value of P_h does give an indication of how easily the machine can be programmed at the assembler level (numbers assigned to A_h and L_h). Calculated values of P_h are in Table III

The denominator in the above formula is an expression of the basic hardware performance. The first term in the denominator of Equation 1

$$0.1M\left[1-\left(\frac{W-F}{2W}\right)\right]$$

takes into consideration the effective reduction in core size for machines that need many double-word instructions. It is assumed that the larger the address field in single-word instructions, the fewer double-word instructions will be needed. If $F = 0$ (true for many 8 bit computers), the effective core size will be one-half of the actual size.

The second term in the denominator

$$\frac{20}{T}(A_h+L_h+I_h)$$

provides a measure of arithmetic, logic, and I/O "power" of a computer. The core memory read-write cycle time (T) has an effect on these parameters, therefore, terms A_h, L_h, and I_h are divided by T. The factor of 20 raises this term to roughly the same order of magnitude as the first term in the denominator. The basic effective memory capacity and the basic processing power are considered of about equal importance in determining the computer's "hardware performance" factor.

The third term, $100N$, is added to boost machines with hardware features that are standard, rather than optional

The last term, $50R$, recognizes the fact that a machine with a larger number of general purpose registers in the CPU is easier to program. This is true because the number of memory references needed in a given program will be reduced.

Software performance—Obviously the hardware price/performance ratio explained above is only half the story, because most computer manufacturers supply software as a part of the basic system package. The software ranges from a simple assembler to a very complex and comprehensive package such as a real-time executive for a system with a bulk memory device and an on-line compiler. In general, the basic price of a machine tends to increase in proportion to the quantity of software supplied with the basic machine.

It isn't possible to evaluate qualitatively the software package of a given machine without the benefit of "hands-on" experience. For this reason, another quantitative (also empirical) equation is proposed. The following variables will be in Equation 2 for software price/performance ratio:

D = Off-line diagnostic routines supplied:
NO = 0 YES = 1
B = Debugging routines supplied:
NO = 0 YES = 1
L = Loader routines supplied
NO = 0 YES = 1
A = Number of assemblers
C = Number of compilers
S = Power of on-line operating system (range of 0 to 100).

The rating of the on-line operating system (S) only indicates how much the system does—not how

Table IV: Total price/performance of minicomputers

Manufacturer, Model number	P
Digital Equipment Corp., PDP 8/L	0.87
General Automation, Inc., SPC-12	0.98
General Automation, Inc., SPC 16	0.99
Honeywell, Inc., H 316	1.02
Digital Equipment Corp., PDP 15/10	1.11
Raytheon Computer, 703	1.14
Hewlett-Packard Co., 2114A	1.22
Business Information Technology, Inc. BIT-483	1.27
Digital Equipment Corp., PDP-8/I	1.28
Compiler Systems, Inc., CSI 16	1.36
Scientific Control Corp., 4700	1.37
Raytheon Computer, 706	1.41
Westinghouse Electric Corp., Prodac 2000	1.41
Interdata, Inc., Model 3	1.46
Lockheed Electronics, Inc., MAC 16	1.51
Micro Systems, Inc., 810	1.62
Redcor Corp., RC 70	1.61
Digital Equipment Corp., PDP 9/L	1.65
IRA Systems, Inc., SPIRAS 65	1.71
Tempo Computer, Inc., TEMPO-1	1.71
Interdata, Inc. Model 4	1.75
Hewlett Packard Co. 2115A	1.78
Varian Data Machines Inc., 620/i	1.80
General Automation, Inc., System 18/30	1.86
Computer Automation, Inc., PDC-216	1.92
Micro Systems, Inc., 800	1.98
Motorola, Inc., MDP 1000	2.03
Data General Corp., NOVA	2.14
Computer Automation, Inc., PDC-208	2.15
Honeywell, Inc., DDP-516	2.20
Hewlett Packard Co., 2116B	2.23
Xerox Data Systems, CF 16	2.23
Information Technology, Inc., 4900	2.27
Xerox Data Systems, SIGMA 3	2.29
System Engineering Laboratories, Inc., 810A	2.37
Varian Data Machines, Inc., 520/i	2.37
Datamate Computer Systems, Datamate 16	2.42
Digital Equipment Corp., PDP 11/20	2.47
GRI Computer Corp., 909	2.52
System Engineering Laboratories, Inc. 810B	2.69
Data General Corp., SUPER NOVA	2.86
Electronic Associates, Inc., EAI-640	2.89
Xerox Data Systems, CE 16	2.93
Data Technology, Inc. DT-1600	2.97
Philco-Ford, Inc., 1216-F	3.27
International Business Machines Corp., IBM 1800	4.64

Minicomputers-A Profile of Tomorrow's Component

REG A. KAENEL
Bell Telephone Laboratories, Inc.
Murray Hill, N. J. 07974

Abstract

There has been an explosive rate of growth of the types and the applications of minicomputers. This growth is a result of employing small computers in a new set of applications that were previously unexploited by the manufacturers of larger computers. To place the emergence of minicomputers in significant perspective, the following analysis is attempted: 1) typical minicomputer characteristics are outlined, 2) technological reasons for the emergence of minicomputers are suggested, 3) the architecture of contemporary minicomputers is given in terms of hardware structure, instruction format, and input/output configuration, 4) a software system that implements a shared environment on a typical minicomputer system is described, and 5) the advantages of using minicomputers are discussed and representative applications are sketched in the area of testing, control, communication, and laboratory experimentation.

Manuscript received August 13, 1970.

I. Introduction—The Emergence of the Minicomputer

The Minicomputer Growth Rate [1]

Over the past five years, the computer arena has witnessed the emergence of low-cost computer systems (see Fig. 1) which have been named "minicomputers". As a matter of fact, the greatest growth in the computer industry in this period has been in minicomputers.

The first minicomputers were introduced in 1962 for aerospace applications. They included such machines as the Arma/Micro Computer, the Burroughs D210, the Hughes HCH-201, and the Univac Add-1000. Commercial minicomputers by such manufacturers as Honeywell, Scientific Control Corporation, Xerox Data Systems, and Systems Engineering Laboratories appeared on the scene about 1966.

Today, some 10 000 systems have been installed representing over 100 different minicomputer types manufactured by more than 50 companies (not counting an additional 15 varieties by 10 foreign companies).

Minicomputer shipments grew approximately 75 percent between 1968 and 1969, reaching about 6000 units per year. The industrial and data communications sectors paced this growth. Continued expansion in these sectors combined with that in the business and laboratory sectors is sufficient to support the anticipated growth shown by the median of Fig. 1, indicating an increase in total shipments from 6000 units in 1969 to a probable 40 000 in 1975. If the education and transportation markets become significant, production could reach the upper bound of Fig. 1—over 45 000 units in 1975.

Fig. 2 indicates substantial growth for the industrial market—from approximately 2500 units in 1969 to over 15 000 units in 1975. Other areas with strong potential are data communications (650 to about 5000) and business (less than 100 to over 9000). Growth in the laboratory market will continue (2000 to about 5000 units), but this will become a less significant portion of the total market. Only moderate applications are anticipated for typesetting and education.

Market studies indicate a significant downtrend in average costs for the central processor unit (CPU). New technologies should begin to reduce CPU costs significantly in the not too distant future. Thus, although unit shipments are expected to rise perhaps fivefold, their dollar value should increase only about threefold, from about $100 million in 1969 to about $300 million in 1975. The ratio of total system to central processor value is expected to advance from about 2:1 in 1969 to about 4:1 in 1975, reflecting the growing value of the peripheral devices of the system. Total system value will rise from about $200-$250 million in 1969 to $2-$5 billion in 1975.

Over the past five years, the number of minicomputers in use at Bell Telephone Laboratories has grown to some 120, comprising 34 different types from 12 manufacturers. About half of these systems are employed in some sort of testing application, most of the other half are for laboratory systems and communication experiments.

Reprinted from *IEEE Trans Audio Electroacoust.*, vol. AU-18, pp. 354-379, Dec. 1970.

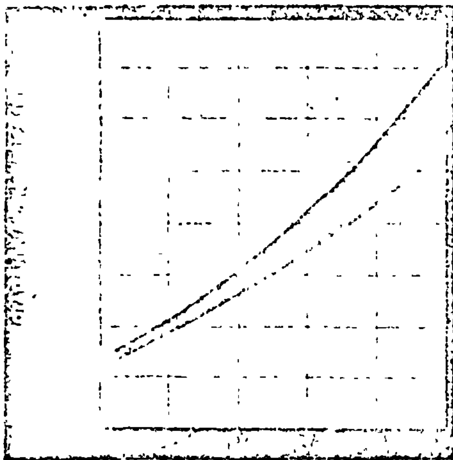


Fig. 1. Projection of the annual delivery growth rate of minicomputers (after Arthur D. Little, Inc. [1]).

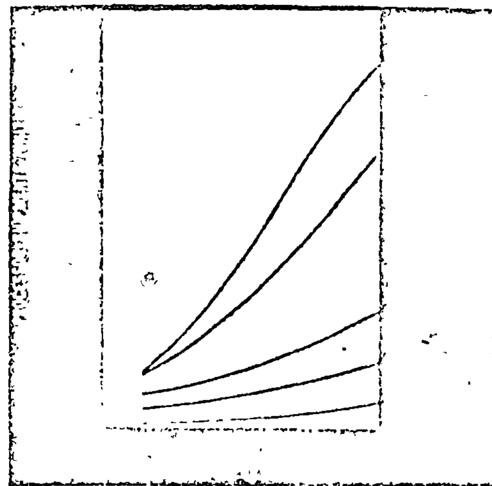


Fig. 2. Projection of the use distribution trend of minicomputers. The application categories are defined as follows (after Arthur D. Little, Inc. [1]).

1) Industrial: management control, such as data collection, and manufacturing including production control.

2) Data communication: batch terminal controllers; store-and-forward message switching and preprocessing equipment, and data concentrators.

3) Laboratory applications physical sciences, e.g., gas chromatography and acoustics, medicine, e.g., clinical analysis.

4) Business applications: fiscal management, e.g., payroll and invoicing.

Today's Minicomputer Industry

The Industry Structure: Early in its development, the minicomputer industry was dominated by a few mainframe manufacturers. The supporting software sector was comprised essentially of small firms; the larger software companies tended not to cater to this market. More recently, the industry has expanded to include many computer and system manufacturers, suppliers of peripherals, independent software suppliers, and systems houses which supply turnkey systems. Let us briefly characterize these industries.

The Mainframe Manufacturers: Today, Digital Equipment corporation, Varian, Hewlett-Packard, and the Computer Control Division of Honeywell, in order of importance, lead the industry, accounting for about 80 percent of total unit sales during 1969. Original equipment manufacturers (OEM) represented about 50 percent of total sales for Digital Equipment, the Computer Control Division of Honeywell, and Varian, and 20 percent for Hewlett-Packard. The remaining sales are for applications where the end user is purchasing a system in which the minicomputer is only a component rather than buying the minicomputer as such to incorporate in a system of his own design. Hewlett-Packard's greater emphasis upon end user systems is due to its activity in the industrial instrumentation market through its other divisions.

Other companies in the minicomputer business include large enterprises like Xerox Data System, which possess the resources necessary for penetration of this market. Additional firms will undoubtedly decide that this market is sufficiently compatible with their other corporate activities to address themselves to it for either OEM, end users, or both. Some companies aiming for the end user market attempt to enter it by initiating their own developments or by becoming associated with a significant system and service capability.

The Peripheral Manufacturers: Until recently, the input, output, and storage devices used with low-cost central processors were those developed for large systems, and in many instances their performance, capacity, and cost were not compatible with the requirements of systems using minicomputers. In fact, only a few of the minicomputer manufacturers have supplied peripheral devices of their own make.

During the last two years, a new class of peripherals for use with minicomputer systems has been emerging. The class includes lower cost and lower performance secondary storage media—disks, drums, and magnetic tapes; data entry storage using magnetic tape cassettes or cartridges which compete in cost and convenience with paper tape and card equipment; line printers; and alphanumeric cathode-ray displays. Higher performance serial printers are also now available.

In the next few years, more of the larger suppliers of minicomputers are expected to develop a strong capability for manufacturing their own line of components with broad performance and cost ranges, while a number of new companies specializing in subunits for minicomputers will offer limited product lines. The impetus for such a development is the ratio of total system cost to computer cost, which ranges from 2:1 to 10:1.

The Software Suppliers: System software and application programs are developed by computer manufacturers, end users, programming firms, and original equipment manufacturers (OEM) customers. Typically, computer manufacturers supply system software for their product line, and the larger firms assist with application programming and supply programs for tunkey systems. In general, however, application programming falls to the users rather than the manufacturers. The application libraries of established manufacturers are largely the product of user group associations.

Programming staffs of manufacturers tend to be small (with the exception of the major suppliers); therefore, independent software firms are important to both computer manufacturers and their customers. Expenditures for all software development have been estimated at about $100 million in 1969 and could reach about $1 billion by 1975 when the total population of machines in this class will be more than 100 000. The greatest number of these will be smaller machines in dedicated applications. These machines will utilize modularly constructed software systems that consist largely of a specialized combination of general-purpose software packages whose development cost can be amortized over many applications. This tends to keep the average software expenditure per machine on the low side.

The System Suppliers: Independent system suppliers offer a capability based primarily upon knowledge of specific application areas. These independent system companies compete with minicomputer manufacturers selling directly to end users. A typical company in this class will develop, or have developed for itself, the software for an application, but will purchase virtually all equipment from established computer suppliers, preferring a single source for all equipment, if possible.

These companies face the usual service problems of the equipment manufacturers as well as a few that are unique. To grow, they must recruit personnel with a strong combination of technical skills and the ability to apply them. In addition, the development of workable, integrated "turnkey" systems requires the selection of applications with a sufficiently broad market that the development costs can be amortized over a reasonable number of sales. Services demanded by customers, such as maintenance of software and hardware and training programs, are usually supplied by the combined efforts of personnel from both the systems firm and the computer manufacturer.

Profile of Contemporary Minicomputers

What is a Minicomputer?: Minicomputers are having a remarkable impact on the computer industry since in some respects their performance is even better than that of their big brothers which have built up the computer industry over the past 20 years. For instance, many minicomputers have core cycle times and peripheral transfer rates which are considerably higher than the conventional large-scale computers. Core cycle times of less than 1 μs are common and some machines are in the region of $\frac{3}{4}$ μs. Maximum I/O transfer rates are frequently determined solely by memory speed, and with 16-bit machines transfer rates of over two million 16 bit-characters per second are quite common.

It is interesting to compare these parameters with an IBM System 360/50 which has a core cycle time of 2 μs and a maximum I/O data rate of 800k bytes per second on the selector channel (which is the fastest channel). The 360/50 could in no sense be classified as a minicomputer and, of course, in other areas such as core size, instruction repertoire, range of peripherals, standard software markup, etc., it is far more powerful than any minicomputer. Nevertheless, this does indicate that for those applications where high-speed minimum-complexity processing is required and rapid I/O transfers are needed, minicomputers may well be more effective than their large counterparts.

How, then, are minicomputers defined?

It appears that any attempt at defining minicomputers must be in terms of their price, performance, and applications. Let us, therefore, consider these factors.

Characteristics of Minicomputers: Minicomputers have often been defined by their price rather than by their performance. As recently as early 1969 some observers were classifying minicomputers as those with a minimum-system cost of less than $50 000. Today a more reasonable figure would be $20 000 and some people may press for $15 000 or even $10 000.

Nevertheless, considering the performance of minicomputers offered today, it is found that they typically have fast processing rates, relatively short word lengths, and versatile input–output structures [2] (Figs. 3 and 4). Their cost is generally related to word length, scope of the instruction set, and the degree of versatility of the input–output structure.

[illegible]	[illegible]	[illegible]	[illegible]
MEMORY			
WORD LENGTH (BITS)	EIGHT	EIGHT OR 16	18
SIZE (WORDS)	1024 TO 4096	4096 TO 32,768	1024 TO 65 536
[illegible] SIZE (WORDS)	1024	4[illegible]	8192
CYCLE RATE (kHz)	125	571 TO 1000	2000
(CYCLE TIME μSEC)	EIGHT	(1.0 TO 1.75)	(0.5)
PARITY CHECK	NO	OPTIONAL	STANDARD
MEMORY PROTECT	NO	OPTIONAL	STANDARD
DIRECT ADDRESSING (WORDS)	256	256 TO 4096	ALL OF CORE
INDIRECT ADDRESSING	NO	[illegible] MULTILEVEL	MULTILEVEL
CENTRAL PROCESSOR			
GENERAL PURPOSE REGISTERS	ONE	ONE, TWO, THREE, OR FOUR	128
INDEX REGISTERS	NONE	ONE	15
HARDWARE MULTIPLY/DIVIDE	NO	OPTIONAL	STANDARD
IMMEDIATE INSTRUCTIONS	NO	HALF YES	YES
DOUBLE WORD INSTRUCT	NO	MOSTLY YES	YES
BYTE PROCESSING	NO	HALF YES	YES

Fig. 3. Typical ranges of minicomputer characteristics [2].

FEATURE/VALUE	MINIMUM	TYPICAL	MAXIMUM
INPUT/OUTPUT			
PROGRAMMED I/O CHANNEL	ONE	ONE	ONE
I/O WORD SIZE (BITS)	EIGHT	EIGHT OR 16	16
PRIORITY INTERRUPT LINES	ONE	ONE STD UP TO 64 OPTIONAL	TWO STD UP TO 256 OPT
DIRECT MEMORY ACCESS	NO	OPTIONAL	OPTIONAL
I/O MAX [illegible] TRANSFER RATE [illegible]	125 000	400 000 TO 800 000	1 000 000
OTHER FEATURES			
REAL TIME CLOCK	NO	OPTIONAL	STANDARD
POWER FAIL RESTART	OPTIONAL	OPTIONAL	STANDARD
LARGEST DISK (MEGABITS)	NO	2 TO 8 0	18 6
ASSEMBLER	YES	YES	YES
COMPILER	NO	BASIC FORTRAN	BASIC FORTRAN, STD FORTRAN, ALGOL
OPERATING SYSTEM	NO	NO	REALTIME, FOREGROUND/BACKGROUND
PURCHASE PRICE			
COMPUTER WITH 4K WORDS OF CORE AND TELETYPE ASR 33	$4200	$12K TO $15K	$24K

Fig. 4. Typical ranges of minicomputer characteristics.

The central processors are single-address binary processors with negative numbers expressed as two's complements. The processors vary the most with respect to the number of accumulators provided, the instruction sets implemented, the instruction decoding technique, and interrupt handling capability. Most processors have an instruction set of 64 to 100 instructions; some have many more, to a maximum of over 200. Many of the hardware features of larger processors have been carried over to minicomputer designs and can be recognized in the tabulation of Figs. 3 and 4.

Examining the typical applications of minicomputers, it is discovered that the minimal system is usually dedicated to control functions, data acquisition, and display. Typically, these systems are used for monitoring and control of a process, or monitoring a process and displaying appropriate data for manual control. The data reduction and analysis capabilities of the average equipment imply longer word lengths, a more comprehensive instruction set, larger primary memory, and usually a need for a secondary store as well as more sophisticated display equipment. The display equipment may be a printer, an x-y plotter, or a cathode-ray tube to generate hard copy and/or on-line interactive displays. The maximal equipment is usually dedicated to perform all of the functions of the lower-type systems and is programmed to operate in a time-sharing mode, supporting foreground and background modes. Thus, an operator may interact with the system in the background mode, e.g., for program development, as it monitors and controls instrumentation in the foreground mode.

It is clear that no simple definition of minicomputers exists. Even the characteristics of contemporary minicomputers have a wide range. Still, computers whose characteristics fall within the ranges summarized in Figs. 3 and 4, and which are used in dedicated real-time type of applications, are generally classified as minicomputers.

II. The Impact of Component Technology on Minicomputers

Technological Background

In the years since World War II, electronics has passed through two distinct periods and has entered a third one, namely that of integrated electronics.

The first period centered on the vacuum tube. It reached its culmination and simultaneously its economic limitations in the 1950's. The second period was introduced by the invention of the transistor. The basic building blocks of the transistor-based technology are transistors, diodes, resistors, capacitors, and inductors. These discrete components are mounted by mass-soldering on the ubiquitous printed circuit board.

Transistor technology developed at a rapid rate in the decade beginning in 1950, but leveled off in subsequent years. For example, frequency response increased from 10 to 10 000 Mc, failure rate per billion element hours decreased from 50 000 to about 1, and the cost per transistor dropped from about $10 to about $0.10.

The ensuring lower rate of improvement in transistor technology has been offset by the great strides made by integrated electronics, which requires extensions and refinements in the processes of transistor technology. A side-by-side comparison of discrete silicon transistor manufacture and silicon integrated circuit manufacture may best illustrate the significance of contemporary integrated electronics.

Both fabrication processes start with a slice of epitaxial silicon. For discrete silicon transistors this slice may pass through about 88 consecutive steps of diffusion, etching photolithography, and various cleaning operations. The fully processed slice will then contain some 3000 transistors that are separated to produce 3000 individual chips. Each chip is then further processed to produce a hermetically sealed discrete transistor. This "packaging" process is very costly compared to the intrinsic cost of the transistor chip.

In the fabrication of silicon integrated circuits the same kinds of processing steps are performed except that the number of steps is about 50 percent larger. The processed slice will typically contain 20 000 devices (i.e., 400 integrated circuits each containing about 50 elements). It is separated into individual integrated circuits each of which contains integral leads that permit easy attachment to thin-film substrates. Each integrated circuit is fully protected both chemically and metallurgically against the atmosphere and does not require an expensive protective can. Integration, in addition to reducing size and cost, also reduces the failure rate by orders of magnitude.

The level of integration of digital electronics has moved from one functional circuit, such as a gate or an amplifier, per chip to the dozen circuits per chip now found in computers. Several tens of circuits per chip are already a proven reality, and serious work is progressing towards the 100 or more circuits per chip area. The projected

number of circuits per chip available commercially appears to grow at the rate of one order of magnitude every six years.

The statistics are even more impressive when expressed in terms of actual components per chip Considering the logic in delivered machines, the density today is of the order of 100 components per chip, and is growing at the rate of about an order of magnitude every four years. Production announcement represents a density of about an order of magnitude higher and advanced development is attempting still another order of magnitude of higher density. Expressed in circuits per square inch, the level of integration used today is about 100 and has grown about two orders of magnitude over the past ten years, starting with a density of 0.1 circuit/in^2 in 1960 with discrete components, and progressing through a density of 2 circuits/in^2 with hybrid integrated circuits[1] (1964), 10 circuits/in^2 with integrated circuits (1967), and is expected to reach 200 circuits/in^2 with larger scale integration in 1972. Regardless of the degree of integration eventually reached, this approach already offers the possibility of reducing circuit costs to a few cents per circuit, of improving reliability, and of offering the ultimate in high-speed performance.

However, to obtain low-cost integrated devices, production must reach sufficient volume to make the startup cost insignificant. These startup costs include design and debugging, generation of test procedures, paperwork for production control, and the effects of initial inefficiency in making a new product. Today, for example, it can easily cost between $10 000 and $100 000 to develop a 50-circuit chip and put it into manufacture. Unfortunately, however, the greater the number of circuits on a chip, the more difficult it becomes to obtain multiple usage of this chip and thus achieve high production levels. This implies the existence of an optimum scale of integration for a given situation.

Two approaches are being pursued to enhance the potential of integrated devices. The startup cost is being reduced by computer-aided design methods and simplified/automated fabrication techniques, thus reducing the prove-in volume requirement. The other approach aims at defining standardized generic functional components from which logic systems can be constructed. Memories are one such standardized component. The benefit afforded by applying integrated electronics to such a standardized component is very substantial, especially where the restriction imposed by such a standardization is negligible or ignorable.

The significant improvements in digital electronics have been the key contributors to bringing down the cost of the early compact aerospace minicomputers, and thus opening up the explosive commercial market for minicomputers which we are currently witnessing.

Functional Minicomputer Devices

Which Functional Devices?: Since the advent of integrated electronics technology, several schemes have evolved for the utilization of large arrays to their full potential In a common and straightforward approach the designer restricts himself to the equipment under consideration at the moment. Faced with only a limited set of problems, he has little difficulty specifying the integrated array types that will efficiently complete the design. While the results are quite encouraging for specific cases, the drawbacks of any mass adoption of these techniques are obvious. This, the so-called "custom approach," would require the semiconductor manufacturer to be responsive to each customer by making numerous low-output production runs of highly specialized devices. The per-unit cost to the user, for his own efforts as well as those of the manufacturer, would be quite high because of the latter's inability to spread initial costs over many devices. Also, the complexity of 100-gate-plus arrays is such that it is difficult to substitute one for another (with efficient results) This would severely limit the off-the-shelf capabilities for both user and manufacturer.

An obvious solution to these problems is the introduction of a small set of standard, integrated, functional components Semiconductor suppliers, making tentative advances into integrated circuits product marketing, have already proposed such devices as adders, counters, and shift registers. However, this represents the solution to only part of the overall problem. A design heavily committed to the use of these devices must fall back on hybrid circuitry using elementary integrated circuits for the large remainder of the circuitry. The reason is that adders, counters, registers, and other orderly, well-defined functions represent the regions of the system with the highest ratios of logic gates to the number of pins through which the functional unit is accessed (i.e., gate-to-pin ratios). After these portions are lifted out of the system, the remainder is characterized by very low gate-to-pin ratios, notably within the control and data routing functions. Unable to continue satisfying the integrated electronics design criteria of high gate-to-pin ratios, the designer must look to more standard (i.e., discrete) components. Unfortunately, for the problem of partitioning logic systems to make best use of integrated electronics, any proposed solution that lacks a total system approach tends to drift toward this pitfall.

Two conceptually different approaches to partitioning are being pursued today: bit slicing and functional partitioning To illustrate the difference, consider the data portion of the computer. In functional partitioning one may specify an adder as one integrated array, registers as another, a shift register as a third, and so forth. On the other hand, in bit slicing one would design an integrated array consisting of a combined one- or two-bit adder, registers, shift registers, etc., then build up his system from this chip type according to the desired word length.

The bit-slice approach has resulted in some notable advantages, particularly the ability to achieve very high gate-to-pin ratios and implement systems using a small number of different array types. However, bit-sliced modules appear to be quite system-dependent [3]. Func-

[1] This term is often used to mean the combination of silicon ICs and thin-film components.

tional partitioning has also been applied successfully. Combining bit slicing and functional partitioning gives the approach the versatility to implement both complex and simple systems of different word lengths with equal efficiency [3].

There appears to be general agreement that control functions are more difficult to modularize than functions related to data operations. Micromemory control techniques using READ-ONLY memories with built-in sequencer and instruction registers lend themselves well to being partitioned into the large modules necessary for integrated implementations. These modules have a well ordered structure that makes them easier to produce than complex circuit arrays. Control functions in this form are then amenable to reproduction in large quantities of identical units.

A representative set of functional devices for minicomputers then could be: register storage, general logic, arithmetic logic, input/output, micromemory counter, microinstruction register, microarray, scratch pad memory, up/down counter, switch, and mainframe memory. Which set of functional devices will eventually form the staple components of minicomputer manufacturers remains to be seen. In any case, the assumption that integrated functional devices will be used in large quantities, and produced with improved manufacturing techniques, suggests a declining cost trend. As the costs decrease, the speed range of bipolar integrated circuits will continue to improve. The benefit from this speed improvement is likely to be small since present bipolar circuit speeds are more than adequate for most minicomputer applications. Typical MOS integrated circuit speeds, which generally are significantly lower than those of bipolar circuits, can be expected to improve by about a factor of 2 to 5 during this period. As the speed of MOS circuits increases, the construction of all-MOS high-performance processors will become feasible. However, present MOS circuit performance is already adequate for many applications.

Partitioning of minicomputers for the most effective use of integrated electronics illustrates the economic criterion underlying the engineering design for integrated embodiments. The conventional criterion of minimizing the number of components is changed by the introduction of the integrated electronic technology, with modularity (or commonality) becoming more important and with component count becoming less important.

Mainframe Memories: It turns out that more than half of the cost of the central processing unit of most contemporary computers is attributable to primary memory. Usually, the more sophisticated computers come with a larger minimum size of primary memory. Historically, the technology used in memory systems (magnetic devices) has been sufficiently different from the technology used in the rest of the mainframe of computers (semiconductor devices) that memory systems have become important functional devices.

Three memory technologies appear to be of greatest importance at this time. They are core, thin film (including plated wire and planar thin film), and semiconductor (including bipolar and MOS integrated circuits). Core memories will be challenged by plate wire and semiconductor memories in the immediate future.

Semiconductor memories promise to offer both cost and performance advantages over core memory. They also offer the potential for higher speed and smaller size than magnetic thin-film memories. However, the power level will generally be higher, although not excessively so. For the near term, the performance of semiconductor memories will be the most important consideration. The costs initially are higher than those for core memories, but are expected to drop rapidly.

Magnetic Memories: Through all of the 1960's, magnetic storage elements have been the predominant memory devices in all classes of digital computers. The speed/cost ratio improvement of several hundred to one that has been accrued during the 60's by system designers using ferrite core systems makes the core a rapidly moving target for any new technology. Core memories remain an elusive cost target due to continued improvements, foremost of which are trends toward two- and three-wire stack designs, radical unitized packaging, and the use of integrated circuits.

Core memories achieve about 0.5- to 5-μs cycle times and are expected to realize a factor-of-two improvement over the next five years. The cost per bit is a strong function of the size of the memory; it is halved for every order-of-magnitude increase in bit capacity (e.g., 3 cents/bit for 100 000-bit capacity in 1970).

The higher speed, nondestructive READ-OUT (NDRO) capability, and lower power offered by plated wire provide a flexibility for accommodating a variety of system storage needs which cores cannot match. Plated wire has the greatest advantages relative to ferrite cores in applications requiring less than 500-ns cycle time. It is in these areas that the initial growth of plated wire will be concentrated.

Neither plated wires nor ferrite cores adequately satisfy the economical integrated-circuit compatibility requirement today. The plated wire bit current, 40 to 50 mA for 5 mil diameter wire, will drop to 15 to 25 mA as 2- to 2.5-mil diameter wire is introduced. The plated wire word drive current, 800- to 1000-mA turns today, can be reduced to 250- to 350-mA turns by several means. The length of the bit will be reduced from 50 to 60 mils to 20 to 25 mils. Ferrite core memories have a decided advantage in bit packing density compared to production plated wire memories. Ferrite core mats achieve 2500 bits per square inch relatively easily, while production plated wire memory planes are 550 to 1000 bits per square inch. Plated wire memories exhibit a temperature coefficient (approximately −0.07 to −0.1 percent per degree centigrade) that is substantially less than the temperature coefficient of standard ferrite cores.

In the past, commercial magnetic memories have not been designed to minimize power consumption. However, power consumption will become increasingly important as memory system prices continue to decrease and storage

capacity increases. It has been estimated that the lifetime cost of providing power to the system and removing the heat from the system and from the room ranges from $2 to $10 per watt.

This is no longer a negligible cost factor. For example, if extended main memory prices drop to 0.5 cent per bit, then 1 mW per bit raises the total cost to twice the acquisition cost. In this context, the reduced power consumption of plated wire memories compared to that of core memories (typically 1/25) may become a significant advantage for plated wire memories.

Semiconductor Memories: Improved processing, outstanding photolithography, and refined circuit configurations have resulted in active semiconductor memory cells having array densities outstripping both core and plated wire. The advantages of semiconductor memories include exceptional speed, nondestructive READ-OUT operation, smaller size, (possibly) lower power, compatibility with processor electronics, and realization of low-cost memory modules. These are the stepping stones for the adaptability to provide a universal memory potential that magnetics cannot meet.

Semiconductor technology offers a single-technology approach which can combine storage, decoding, and sensing on the same chip. This feature promises to have significant advantages in terms of cost, reliability, and flexibility.

Probably the strongest single argument that exists for the single technology realization is that by this means the total number of off-chip interconnections in the memory system is minimized. Such interconnections not only make a major contribution to failure rate, but contribute significantly to integrated circuit operating costs today. Their minimization thus is a long step toward lower initial cost and improved realiability of operation—probably the two most important measures of merit for a mainframe memory system once adequate speed is achieved.

Reliability has been one of the big problems of semiconductor memory. Field data are becoming available to indicate that 10^5 hours or more of mean time-to-failure for interconnected integrated-circuit chips can be assured. Store reliability appreciably greater than actual device reliability is possible through the use of error-correcting codes.

Store volatility is another aspect of the system reliability problem. Information is lost from a semiconductor memory when the power supply is interrupted. This limitation may eliminate semiconductor memories for certain applications, but appropriate system design may satisfy the recovery requirements of many applications. One solution is a backup power supply; another solution is the provision of a nonvolatile backup store; still another solution may be the development of nonvolatile semiconductor memories such as the experimental metallic nitride semiconductors (MNOS) memories.

Thus, it appears that the overall system reliability may be comparable for magnetic and semiconductor memories of comparable storage capacities.

READ-ONLY Memories: READ-ONLY memories find two dominant applications in computer mainframes: microprogramming and macroprogramming applications.

In the microprogramming application a READ-ONLY memory is used to interpret the instruction set by controlling the sequence of logical operations required (for example, for an add operation or a shift operation).

Fast READ-ONLY memories, at a cost comparable to or less than that of the much slower READ-WRITE memories, offer the capability of implementing sophisticated instruction sets, arithmetic routines, and major service programs in minicomputers using standardizable register banks and transfer gates. Bipolar READ-ONLY memories of 1024-bit size and with access times of less than 50 ns are now available. They permit implementations of instruction sets that have performance speeds comparable to those realizable with the more costly, fully customized handwired approach. Testing of computers whose instruction set is realized through a micropogrammed READ-ONLY memory is greatly facilitated.

In the macroprogramming application a READ-ONLY memory essentially replaces a portion of the main memory with a fixed rather than an erasable memory. The READ-ONLY memory then contains instructions in the same sense as would an erasable memory.

There are many variations for implementation of READ-ONLY memories. One possibility, which is applicable to both magnetic and semiconductor technologies, is to have the wiring pattern for input and output fixed to reflect the appropriate macro- or microprogram. This approach requires a new ROM if programming changes are made, and represents a disadvantage if frequent program changes are expected.

This problem can be avoided by using HARD-WRITE memory, such as the piggy-back twistor. In this approach new micro- or macroprograms are electrically written into memory and, once they are written, permit the memory to be operated in a READ-ONLY mode. Another possibility is the capstor-type memory where a removable mask contains the program.

Implementation of all of these types of ROM in semiconductor form is being pursued. The most important candidates at this time appear to be READ-ONLY diode matrix memories and semiconductor memories.

The Technological Impact

The rapid advances in component technology have had a spectacular effect on computer technology. Over the last decade, computer speeds have increased by a factor of 1000; costs of computation have decreased by a factor of the order of 500, and the memory capacity of digital computers has gone up by some three orders of magnitude. This remarkable technological progress has made possible the construction of low-cost and stripped-down computers, i.e., minicomputers, whose performance characteristics still make them most suitable for a wide range of applications and, despite their very general

capabilities, highly competitive with specialized digital controllers in terms of both cost and performance.

Since their introduction, the cost/performance ratio of minicomputers has itself improved by two orders of magnitude as a result of advances in integrated circuit, core memory, and packaging technologies. There is little doubt that the trend toward even more cost-effective minicomputer processor hardware will continue in the foreseeable future.

The potential of integrated electronics will have an increasingly beneficial effect on the cost/performance ratio of minicomputers as 1) the fixed costs associated with circuit designs are reduced, 2) the minicomputer manufacturers become larger so as to justify increasingly automated production facilities, and 3) an increasing standardization of product and functions is achieved.

In particular, the impact of semiconductor memory upon machine architecture will be great. As a result of the higher performance of semiconductor memory, the need for general registers which serve primarily as speed buffers for processors will be diminished. Also, in those instances where input/output buffers have been incorporated into the channel controllers, it will be feasible for many systems to make a direct data transfer to main memory. For minimum-cost systems it may also be possible to incorporate various control registers into the main memory. Thus it appears that integrated electronics may remove the designation of the mainframe memory as a historic distinct functional computer device.

It has been speculated [4] that an entire processor may be fabricated on a single chip in the not too distant future. The ultimate cost of such a processor, utilizing MOS technology with 1000–16 000 components per chip, may be in the $10 range. A computer can then be constructed from such a processor chip by combining it with a READ-ONLY memory and a semiconductor READ/WRITE memory.

Before a fully integrated processor is produced, however, there will appear an ever-increasing number of minicomputers [5] which will employ varying degrees of integration based on the availability of a rapidly growing number of integrated circuit devices which challenge the ingenuity of the system designer.

In the meantime, however, the rapid and significant advances in integrated component technology also appear to be causing a profound change in the structure of the component suppliers and computer manufacturers. The use of increasingly sophisticated integrated circuits has made the computer manufacturer increasingly dependent on proprietary integrated circuits to produce unique computer systems; the computer systems differ less in the combination of components used and more in the types of components employed. This has the effect that computer manufacturers tend to build up their own integrated electronics capabilities (this trend is sometimes referred to as a downward integration). Concerned by the gradual erosion of the dependence of their customers, the integrated circuit suppliers are expanding their systems development efforts, going even as far as developing their own computers, which puts them in direct competition with their customers (this is sometimes called upward integration). The big question today is how the computer manufacturers will deal with the suppliers-turned-competitors. It is quite conceivable that both computer manufacturers and components suppliers will gradually become largely self-sufficient suppliers of integrated systems of increasing sophistication that are aimed at different market segments.

III. Systems Architecture for Minicomputers

Functional Description of Minicomputers

General Sequence of Events: A typical mainframe consists of a memory section, a processing section, and an input–output (I/O) section. Each of these sections contains several registers; in particular, the processing section includes a program counter, an accumulator, an accumulator extension register, and one or more index registers. All of these registers are connected by several data buses.

During program execution, an instruction is moved from the location pointed to by the program counter into the instruction decoder. Depending on the instruction fetched, a data item is moved from one register (or memory) to another register (or memory) via the data buses; also, the instruction decoder may cause the data to be modified or combined with previously moved data. The instructions recognized by the instruction decoder are executed by the processor control unit which sequentially operates a set of transfer gates and thus implements the instructions. When the execution of the instruction has been completed, the program counter is advanced and the next instruction is fetched for execution.

The traditional control units are fixed wired. The use of microprogrammed control units is relatively new in minicomputers though old in concept. Microprogramming is the employment of a stored program module for the control unit of the CPU rather than the traditional fixed wired control unit. Within the stored program control unit, a set of microcommands is stored. These microcommands, when addressed from either a microprogram location counter or from a microcommand register, are decoded and executed. The microcommands normally perform elementary processing operations and are each executed in one machine clock cycle.

Stored Program Control Unit: In computers which use the microprogram technique, basic operations are determined by a stored program rather than by hard wiring. Since the stored program can be easily changed or expanded to suit the application, the hardware of the computer need never be altered. The result is a more flexible system at a lower cost.

Microprogrammed computers generally operate from two levels of stored programs: 1) a microlevel stored control unit, and 2) a macrolevel using commands in core to specify the microcommands performed by the control unit. In the latter a macrolevel instruction that can be

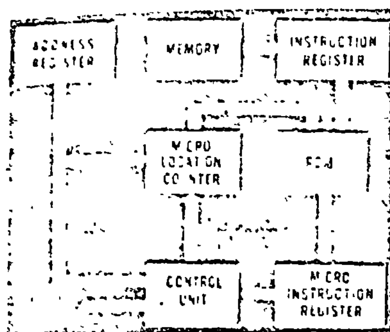


Fig. 5. Lower cost, more versatile computers are feasible through expanded use of ROMs for microprogramming. Computer instructions and counter supply address inputs to ROM, whose outputs are used as microinstructions.

user-generated calls a set of microcommands in the stored control unit to perform a specific function of subroutine. Hence, a microprogrammed computer is emulating the operation of a fixed control unit.

While the microprogram technique can be applied to the entire computer operation, present usage is a combination of hard-wiring and stored programs. The microprogram storage unit typically consists' of a READ-ONLY memory (ROM) since the stored program usually need not be electrically alterable during normal operation.

Minicomputers that use ROM to store microprograms attain a high degree of flexibility but sacrifice some speed in conventional instruction execution. Despite the high speed of ROM (typically 100-ns cycle time) and the high speed of microinstruction execution times, the total instruction execution time increase since a conventional instruction consists of several microinstructions. However, the user can define an instruction set specific to his application. This may mean that the overall throughput for the specific job may be increased and the core storage requirement decreased. Microprogramming is a technique for gaining faster execution of compound instructions without paying the price of linkages, jumpsave, and other features required in traditional general-purpose software.

Use of a ROM in a microprogram is illustrated in Fig. 5. Here, the combination of the computer instruction and a counter are used as the address inputs to a ROM and the resulting outputs are used as microinstructions. The full microprogram sequence begins with an instruction that has been read from memory at the location specified by the address in the address register. The operations code of the instruction selects the required microprogram by addressing a particular portion of the ROM. The first output word of the ROM (the microinstruction) is gated into the microinstruction register and an operation is performed in the control unit. If the result of the operation is incomplete, the control unit advances the microlocation counter and the new address selects a second microinstruction, and so on.

When the control unit completes its operations, it requests a new instruction by advancing the address in the address register, then resets the microlocation counter so that it is prepared for the next microprogram sequence.

Program Execution Interrupts:

Interrupt Capabilities: Interrupt systems equip minicomputers for quick response to input/output demands or other events that require immediate attention. A priority interrupt system provides immediate response, conserves memory space, and improves program running speed. The alternative technique of scanning (polling) devices in sequence is much more expensive in terms of program length, memory space, and program running time, and does not provide immediate response.

Interrupt systems for minicomputers are quite extensive and consist of internal and external interrupt levels. The internal interrupt levels include power fail-safe, memory parity, memory protect, and real-time clock interrupt levels. Usually, the memory protect level can be inhibited from the operator's console, but the other internal interrupt levels cannot. The basic external interrupt system usually consists of one or two levels; additional interrupt levels can be added in modules of two, four, or eight levels providing maximums from 16 to 256 levels.

External interrupt levels are under program control and can usually be disarmed or inhibited individually. A disarmed interrupt level ignores an interrupt signal; an inhibited interrupt level stores an interrupt signal but does not cause an interrupt until the inhibition has been removed.

Interrupt Sequence: Usually the interrupt procedure implemented in the hardware consists of suspending processing and completing a set of instructions out of sequence; the interrupt level provides the core address of the first instruction to be executed out of sequence. If the interrupt servicing subroutine consists of only one instruction, the contents of the program counter are not changed, and the interrupted program is continued after the interrupt instruction is finished. If the interrupt servicing subroutine consists of several instructions, however, the instruction stored at the location addressed by the interrupt instruction must be a transfer of control to the interrupt servicing subroutine. The interrupt procedure then executes a transfer of control instruction to an indirect address in a core location selected by the interrupt level. Sometimes the procedure includes storing the processor status before transferring control to the interrupt servicing subroutine.

Some hardware provision is made to block out all interrupts until the interrupt servicing subroutine has stored the status of the processor, contents of the accumulator, index register, program counter, overflow, etc. In addition, hardware provision is made to block out all interrupt levels of an equal or lower priority until the current interrupt level is released by instruction.

Functional Description of the Minicomputer Sections

The Memory: Contemporary machines use core memory exclusively for implementation of the memory sec-

tion. Memory sizes range from 1000 to 65 000 words and memory speeds from 0.5 to 8 μs. Options offered for memory include a parity check bit per word, which serves to detect read errors, and a memory protect bit, which prohibits writing into selected memory sections except when the computer operates in a restricted supervisory mode. The memory protect method is used to permit partitioning memory into separate banks, each containing a separate program, so that execution of a program in one bank cannot affect the program located in an adjacent memory bank. Another implementation of the memory protect feature is by upper- and lower-bound registers that define the protected core area. This feature also includes a set of instructions that is effective only if the machine operates in its supervisory mode and that permits setting up the protective fences. An associated interrupt level signals a protect violation or a parity failure.

Three memory addressing modes are customarily made available: 1) direct addressing by which the memory location specified in a memory reference instruction is accessed, 2) indirect addressing by which the location specified in a memory reference instruction contains a pointer to (i.e., address of) the location to be accessed, and 3) indexed addressing by which the content of an (index) register is added to the direct or indirect address to establish the effective address of the location to be accessed.

The entire core is usually addressable via indexing and/or indirect addressing. In some cases, an address extension register or double word length instruction permits direct addressing of all of core. Indirect addressing is usually recursive (chaining of indirect addresses) with indexing allowed at each level.

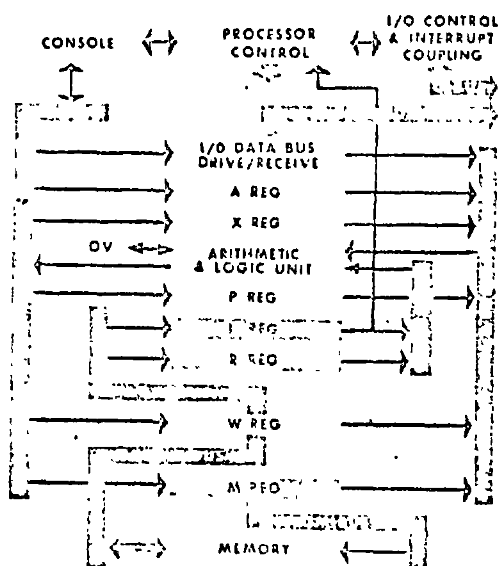


Fig. 6. Typical type I architecture using four buses. The following seven registers are provided in this example. The accumulator (A) and index register (X) are the principal registers used for working storage. Most instructions which operate on data (operands) make use of these two registers and the arithmetic logic unit. The instruction counter (P) holds the instruction address. Registers R, W, and M work in connection with the memory.

Because of the short wordlengths, most minicomputers use a paging technique to address core by a one-word memory referencing instruction. The length of the pages is defined by the number of core locations that the address field can specify. These instructions can directly address the local page that contains the instruction and/or a specified base page in core. This base page is usually the first page in core. Some machines permit specifying the location of this page by a page register. Despite the paged use of memory, overlapped timing for the pages is not used nor is it used for the banks of partitioned memories.

The 8-bit minicomputers differ little in addressing capabilities from the 16-bit machines because the memory reference instructions of the 8-bit machines use two words per instruction instead of one.

Several minicomputers include a READ-ONLY memory (ROM) either as basic hardware or as optional equipment. ROMs have a shorter cycle time than core memories or even semiconductor READ/WRITE memories. The ROMs are used in two different ways: 1) to store, protect, and decrease the execution time of real-time programs, executive routines that allocate the processor time to different application programs, and frequently used subroutines, and 2) to store microprograms that define the processor's instruction set.

The Processor: As mentioned before, the processor section typically consists of general registers, the arithmetic logic, and a control section which can use either microprogrammed or hardwired logic hardware. The principal variations in contemporary processors are in the number of registers provided, the instruction set implemented, the instruction decoding technique, the interrupt handling capability, and the bus arrangement. The general registers may vary from 3 to 256. In most cases the processor operates in a word mode, but in some machines the unit processed is a byte with the unit stored in a memory register being one or two bytes. Instruction sets implemented by the control unit in combination with processing units range from a very rudimentary set with functions such as multiplication, division, and floating point available only in software to a complex set which is a subset of that for the IBM 360 system. The instruction sets provided are discussed in more detail in the section on processor operational capabilities.

Two distinct bus structures are used. In the type I structure [6] a multiplicity of buses interconnect memories, accumulators, arithmetic units, instruction and index registers, and other key elements (see Fig. 6). In the more recently introduced type II structure [7], [8], often referred to as a Unibus structure, all the key elements are attached to two buses, i.e., the source and destination buses (see Fig. 7).

The type I structure is not as functionally oriented as the type II structure. In the former structure, the processor is primarily oriented to accumulate data from a subsystem, perform arithmetic calculations on it, and return

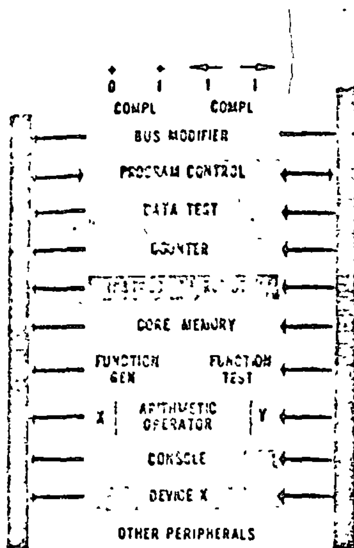


Fig. 7. The elements are brought outside the actual processor, and all computer devices are connected across the same bus structure as the devices to be controlled [7]. The bus modifier provides a programmable path that permits taking data from any input device and moving to any output device, and performing operations on the data as they pass from one device to the other. Program control provides the signals that indicate when a source device is to send data, sets up the route the data are to follow, and specifies the destination device that is to receive the data.

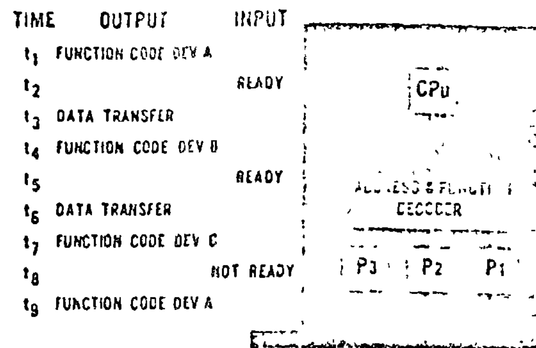


Fig. 8. Fully sequential I/O scheme. The left side of the illustration functionally indicates the sequence of events for a typical I/O operation. The right side shows a representative implementation. The peripheral devices (P_i) are enabled for I/O by the decoder circuitry. Data transfer takes place by the data bus.

the results. It was not conceived in terms of controlling and sequencing activities that may not involve mathematics at all. In fact, the user of a type I minicomputer must program control systems function in terms understood by the more arithmetically oriented structure despite the fact that these terms may bear little relationship to the functional requirements. The actual programming of the control function is then often turned over to a programmer specialist who relates the systems requirements to equivalent terms that can be translated as instructions to the computer. However, in so delegating system development tasks, project control becomes more difficult and development cost increases. The type II structure has been chosen to facilitiate the functional programming for control applications. This will be illustrated in the section on processor operational capabilities.

The I/O Structures:

Responsiveness of the I/O Schemes: In the many applications of minicomputers, the input and output of data is a dominant factor. Minicomputers are being used extensively for process control, data multiplexing, communication line concentration, switching, and other areas in which data are continuously transmitted between the computer and external devices. Requirements for minicomputer systems in even one I/O application area can vary greatly. For example, when multiplexing communication lines, a typical small-scale system might consist of an in-house time-sharing computer with a minicomputer front end. The minicomputer would be used to handle the communication line processing for say 10 to 30 low-speed lines, each connected to a single teletypewriter, or it can serve as a line concentrator for up to 64 full duplex lines at speeds up to 4800 bits per second.

The basic facility for transferring data between the computer and the I/O device is generally referred to as the I/O bus.

Signaling on the I/O bus may be either nonresponsive or responsive (handshaking). Nonresponsive signaling may be simpler and less costly: signals remain on the lines for a preset time (dictated by the worst anticipated conditions), and are then removed under the assumption that they have been detected at the receiving end of the bus. Responsive signal systems present data to the receiver, continuously or repetitively, until an "acknowledge" signal is returned. Advantages of the latter approach are: 1) less strict tolerance, or faster operation, 2) greater control, and 3) flexibility in adapting to variations in controller speeds or in bus lengths. Virtually all minicomputers use a responsive I/O scheme.

Two categories of responsive I/O schemes can be distinguished. They will be referred to as 1) fully sequential schemes, and 2) overlapped schemes. Most minicomputers use a combination of these schemes to match their intrinsic speed with the response rate of the I/O device.

In the fully sequential schemes (e.g., Fig. 8) an I/O operation is fully executed before the next operation is initiated. As a result the required handshaking sequence associated with a specific I/O operation is completed before the next sequence is activated. Because of the rigid handshaking protocol, most sequential I/O schemes use a hardwired I/O interface to perform the handshaking sequence within a computer instruction cycle.

The handshaking sequence associated with some I/O operations can be intrinsically slow. For example, acknowledgment of a "start I/O device" may take several

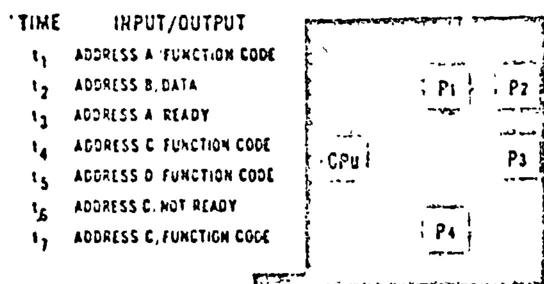


Fig. 9. Overlapped I/O scheme. The left side of the picture functionally describes a typical I/O sequence of events. The right side suggests a representative implementation comprising a set of peripheral devices (P_i) connected in series along a signal bus.

milliseconds, which is very long compared with the microsecond cycle rate of most contemporary minicomputers. In these instances it is preferable to initiate the I/O operation and determine the success of the execution of the I/O instruction at a later time.

Using an implementation similar to that used by a fully sequential I/O scheme, the processor can determine the I/O device response by querying a register associated with the device under program control. Thus, the handshaking sequence is broken down into programmed distinct I/O operations.

The implementation of an overlapped I/O scheme shown in Fig. 9 is assuming increasing importance in communications oriented systems. All I/O devices are serially located on a common I/O bus. Each device is equipped with circuitry that recognizes the operation codes that are addressed to the device; each device is also equipped with circuitry that associates the responses with an identification code designating the device from which the response originated. In some systems, the device addresses are explicitly given (serially coded or transmitted along a parallel bus); in other systems each device is associated with a distinct time slot (time multiplexing), simplifying the decoding circuitry.

The data stream to the peripheral I/O devices is placed into an output queue from which it is transmitted to the I/O devices. The inputs from the devices are placed into a response queue from which they are fetched to be analyzed by the processor and related to the proper I/O operation instruction.

Some overlapped I/O schemes provide means for rapidly detecting specific response codes which can cause a processor interrupt. These responses can thus be promptly recognized and the proper action immediately taken. Other schemes provide means for automatically ignoring codes that merely represent a standby condition (idling codes).

Sequential I/O Schemes: The most widely used I/O schemes are of the fully sequential type. Several schemes of this type are currently offered with virtually all minicomputers. Because of their importance, these schemes will be briefly discussed.

In the sense that overall system operation is under control of the program, it may be said that all input/output transfers are also under program control. However, there are wide variations in the amount of program control necessary to effect a transfer.

Fully Program-Controlled I/O: The standard method used is a programmed party line channel. This method is under full program control. Each word is transferred on the I/O bus between a working register and the I/O device, and stored in memory by means of program instructions. Thus, the maximum data transfer rate over a programmed party line I/O channel is limited by the program overhead for servicing the I/O devices requesting service.

Data transfers performed on the standard I/O bus require program intervention for each word transferred. There are two methods for using the program to control transfers. In the first method, the transfer of each word is effected by means of a program request. The program addresses the desired I/O device and then waits until the device is ready to transmit or receive data. This delay leads to processing inefficiency. Therefore this method is resonable only in cases where the I/O data is highly disciplined (predictable and program synchronized), or where processing requirements are low.

To free the program for other tasks during the waiting period, the interrupt method is used. After the I/O device is instructed on the function to be performed, the program may continue processing until an interrupt signal is received, indicating that the device is ready for the next instruction. With this method an interrupt followed by an interrupt processing sequence are still required for each word that is transferred.

Memory Access I/O: The throughput can be increased by reducing I/O interrupts and program control of I/O transfers, i.e., both direct-processing time for the transfer and the time required for program interrupts. This reduction is obtained by employing additional equipment in which I/O transfers between peripheral devices and mainframe memory are controlled by hardware instead of by the stored program. When data is transferred directly between an I/O device and memory, program intervention is required only at the beginning and end of the transfer, or when there is a detected error. Dedicated locations in memory provide the beginning and final core addresses that define block length. Thus, it is possible to transfer large blocks of information while simultaneously processing other data. This allows relatively high-speed devices to interface to the channel for data transfer.

Three methods are used for transferring data directly between the main memory and the peripheral devices. In the first method a direct memory access channel (DMA) provides a direct data path to the memory. It permits the high-speed transfer of a contiguous block of data. Control and addressing logic are in the external device which requests service, and connection is made directly to the main memory rather than through pro-

cessor registers. The DMA connection permits stealing of main memory cycles from the processing unit when the appropriate peripheral device demands service. One of two modes may be used for performing cycle stealing. One is to switch control to the DMA interface and perform a block transfer. The other is for the input-output control logic to check for a peripheral "ready" status; if the peripheral is not requesting service, control is returned to the processor until a peripheral transfer is ready. Because most minicomputers have only one memory bus, transfers via DMA suspend processing if the processor and DMA try to access memory simultaneously; the DMA channel has priority. Several devices can usually interfere to a DMA channel, but only one device can use the channel at a time.

A second interface method is direct multiplexed memory access (DMC). With this arrangement a number of peripheral devices are serviced by a hardwired input/output program that is executed when a data transfer is to occur. This program effects the data transfer by way of the standard input/output bus and associated processor registers in which the data are assembled and buffered. The multiplexor channel consists of a number of subchannels that are scanned in sequence for service or can request service by an interrupt. A priority channel is usually used to support high-speed devices.

Two pairs of core locations are dedicated to each subchannel to provide initial (or current) and limit addresses for continuous block transfers. The subchannels usually share a data buffer and current address and limit address registers.

Thus, several memory cycles are required to transfer each byte of data because the channel must automatically access memory to load the current address and limit address registers, to pack the current byte in the current word, and to restore the current address in memory. Up to six memory cycles may be required depending on the number of subchannels active, the order in which the subchannels require service, and the design of the channel. This causes the DMC to have a data rate noticeably below that of a DMA arrangement.

The third method is a block-multiplexed access method (selector channel). At any one time, only one I/O unit can access memory. The other units can gain memory access only at the end of a block transfer.

It is clear from the foregoing discussion that the hardware for memory access I/O must automatically perform memory addressing. The current address and the end of the data block must be stored and accessed. As each word of data is transferred, the current memory address is updated, and a check is made to determine if the final address has been reached. If so, an interrupt must be generated. The current address is initialized by the program at the beginning of the transfer operation as the starting address of the data transfer.

Current and final memory addresses may be stored in reserved locations of core memory or in separate external registers. The latter case results in more expansive but faster throughput. Those devices which are capable of interlacing transfers by word require a set of current and ending addresses for each device connected, or for the maximum number of devices capable of simultaneous operation.

Other registers are often provided, especially in communications oriented computers, to detect special control codes embedded within the data stream.

Since the register is associated with a control unit and is program loadable, the user has wide flexibility in his selection of criteria with which to end a data transfer and interrupt the program.

Priority Control: When a number of units are connected together on a common bus, several techniques are available for establishing priority and resolving contention between two or more units. If I/O data are disciplined, transfers can, with reasonable efficiency, be under program control. Then the selection of the I/O unit may be by any appropriate algorithm. Alternatively, if control is by means of interrupt, three basic techniques are employed, as follows

1) A single common line is used by all I/Os to signal an interrupt. When the interrupt is allowed, the program polls each device, according to a polling table, to determine which devices are bidding. The priority is implicit in the order in which I/Os are listed in the polling table.

2) A single interrupt line is used, and a single go-ahead is used. I/O devices are connected in a daisy-chain arrangement on the go-ahead line, with priority assigned in the order of connection to the line. Higher priority units obtain access first, and when not bidding, relay the go-ahead signal to the lower priority devices in order; the device which is successful in obtaining an interrupt sends its identity on the address lines.

3) Separate interrupt lines and separate go-ahead lines are provided for each device. Contention is resolved on a priority basis by some type of hardware.

In a number of computers, priorities are program controlled by what is usually referred to as a masking instruction. This type of instruction can selectively prevent an I/O unit from generating an interrupt until the instruction is negated. This facility provides an efficient means for allowing only higher priority interrupts to interrupt lower priority interrupts.

Operational Description of Minicomputers

The Minicomputer Instructions:

The Instruction Set: Most of the operation codes of the instruction set are used for memory referencing instructions. Nonmemory referencing instructions use additional bits of the instruction word as part or all of the operation code, so the number of instructions is not necessarily small. The modification field further qualifies the instruction by defining the addressing mode (direct, indirect, and/or indexed) or specifying that the address field contains a constant (i.e., the address field contains a literal).

As discussed earlier, because of the short instruction word length, most minicomputers use a paging technique to address core. Memory-referencing instructions can directly address the page that contains the instruction and/or a base page. In this case, the address field provides an address increment. The effective address is calculated in accordance with the address mode: usually a base address of zero or the content of the program counter specify the page number (i.e., the base address) and the address field specifies the core address relative to that page. The entire core is generally addressable via indexing or indirect addressing. In some cases, an address extension register or double wordlength instruction enables directly addressing all of core.

Most contemporary processors are single-address binary processors with negative numbers expressed as two's complement. The basic instruction set of type I structure machines usually includes the arithmetic operations of fixed-point add, subtract, multiply, and divide, although multiply and divide are frequently available as optional features. Double-precision fixed-point add and subtract are frequently provided. A few of the larger machines offer floating-point hardware as an optional feature. All offer some form of logic, compare, and shift operations; many also offer byte manipulation instructions. The I/O instructions are usually very general and complex. Commonly, the I/O instruction also provides control for operational features, which are addressed as I/O devices.

General-purpose machines have a greater number of instructions available, and therefore generally have a more traditional fixed-wired processor. The controller-type machines have a more limited number of instructions and frequently rely more on microprogrammed organization to permit specialized instruction sets and user-generated macroprogrammed languages.

When a minicomputer is equipped with a memory protect feature, the instruction set includes a protected set of instructions, such as those that load the upper- and lower-bound registers or change the interrupt status and I/O instructions.

Type II structure machines could specify an instruction set akin to that of type I machines. However, recognizing the capability of transferring data directly between the various processing units of type II machines and to orient these machines more toward control applications, the instruction set consists of one main instruction: "Device X to Device Y". Provision of this type of instruction has been found to greatly facilitate certain control application programs, and provides a natural environment for recognizing the numerous I/O devices connected to type II machines.

The Instruction Format: Most 16-bit processors of type I architecture use one-word instructions with the following format (see Fig. 10): a 4- to 6-bit operation code, a 2- to 4-bit modification field, and an 8-bit address field. The 8-bit minicomputers differ little in addressing capability from the 16-bit minis because the memory

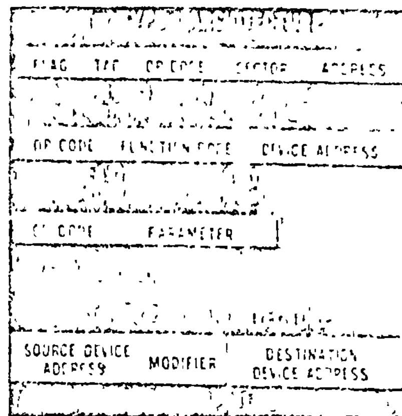


Fig. 10. Instruction formats of machines of type I and type II architectures.

referencing instructions of the 8-bit machines use two words per instruction instead of one.

Some 8-bit processors operate similar to the 16-bit processors by using two words per instruction. Others, however, require only one programmed byte for many instructions by using a byte-sharing technique. These instructions also use two bytes per instruction but only the first byte is stored as part of the program. The second byte is stored as a constant (shared byte) in a dedicated core area and is referenced by the first byte. The 8-bit instruction word format uses three or four bits as the operation code, none or one bit as a modification, and four bits as the address field.

The operation code specifies an instruction class, such as compare, and a dedicated core area of 16 locations. The address field of the instruction selects a location within the dedicated core area: the accessed core location contains the shared byte that further defines the instruction, such as type of compare and addressing technique. The shared byte does not increase the number of operation codes available, nor does it decrease the instruction execution time: it does decrease the amount of core storage required to store programs.

The instructions for machines of type II architectures have the single format shown in Fig. 10. The actual operation performed by an instruction is dependent on the unique combination of source address, destination address, and modifier. This type of format appears to naturally and meaningfully represent the typical control computer environment with its characteristically large I/O control devices. All internal and external system elements are thus directly addressable by use of a compiler-like functional language rather than a mathematically oriented language. The programmer does not have to develop the involved command sequences that will translate process data into a language that the computer can understand, and then back again into instructions that the equipment can react to. The unimpeded flow of data from device to device saves temporary storage locations for bookkeeping purposes and provides the de-

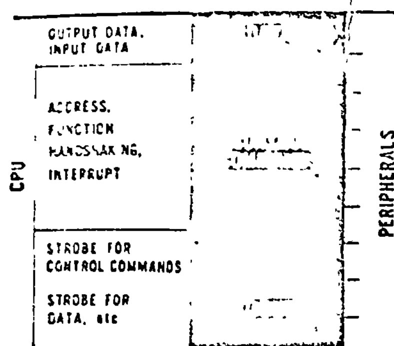


Fig 11. Simplified I/O interface structure from the processor I/O interface to the I/O device.

signer with modularity and flexibility to adapt the computer to specific system requirements.

The Minicomputer I/O Interface Systems: Since virtually all minicomputers are used within an environment that is strongly dominated by I/O devices, the I/O interface has an operational significance similar to that of the instruction repertoire. It is through the suitable application of the instruction repertoire in combination with a properly configured I/O interface that a minicomputer is made to perform a useful function.

Functionally, the I/O bus permits the transfer of data between I/O units and the working registers of the computer. Some computers have dedicated registers for I/O transfers, reducing housekeeping requirements. The number of devices which may be connected is determined by the size of the address field of the computer's I/O instruction, and correspondingly, the number of address lines in the I/O bus. Thus, a one-byte addressing structure will accommodate 256 individual I/O units. This is, of course, a theoretical maximum and does not imply that the logic devices can be contained in one cabinet, nor that the computer has the capability of accommodating the throughput of any combination of 256 devices working simultaneously.

The I/O bus will contain communication lines for the following purposes (see Fig. 11).

1) Data output lines for transferring data from the computer to the device, and data input lines for transferring data from the device to the computer. Transfers may be one word at a time or in multiples or submultiples of a word.

2) Output address lines for enabling the computer programs to select one of the I/O devices connected to the bus, input address lines for enabling an I/O device to identify itself to the program, function lines with which the program designates the functions to be performed by the selected I/O device, status lines whereby the device indicates to the computer program its status of busy, ready, etc., program interrupt lines with which devices signal the program to request an interrupt, parity error lines whereby a device indicates to the program the detection of an error, and various miscellaneous control functions such as setting and clearing of interrupt mask, system reset.

3) Timing lines whereby the device operations are synchronized.

To reduce costs, some of the above functions may be combined on the same lines. For example, the data lines are often used also for the device address and the function information. Additional signal lines are then used to indicate the type of data present. The time-multiplexing of lines by different pieces of information achieves lower cost at the expense of slower operation.

To connect an I/O device to the computer I/O interface, a device controller is required. Quite a large set of signal leads may have to be connected to facilitate the handshaking sequences of a responsive I/O scheme [9].

IV. Software Systems for Minicomputers

Overview of System Software

System software is designed to assist the user in program preparation and in system initialization and operation. For program preparation, an assembly language plus various aids to the programming of the machine are usually available including trace; editing, breakpoint, debug, and linkage routines. Quite a few manufacturers also supply a basic FORTRAN compiler to aid the user in developing arithmetic programs, and some also provide a standard FORTRAN compiler. For system initialization and operation, the manufacturers customarily make available such aids as peripheral operating systems, real-time monitors, partitioned (multiprogramming) operating systems, loader and utility routines, machine diagnostics, and restart/fallback routines.

Assembly languages vary in structure from one machine to another. Most assembly languages are indications of the specific machine's overall hardware architecture and logic. An increasing number of assembler systems for minicomputers that run on large general-purpose computer systems are becoming available. This avoids the necessity of equipping the minicomputer systems with general-purpose I/O devices that are usually not needed except for program development. Some of these systems are simulator packages for larger machines which permit program development for the minicomputer directly in the assembly language of the large machine, and thus make conveniently available the sophisticated macro-assembler capabilities of these large-system assemblers. Many of these systems have been derived from the compiler systems used for programming the large computer by providing suitable pre- and post-processors.

Efficient compiler operation is difficult to realize in small machines, and virtually impossible in 4000 words of core. Most minicomputer compilers require 8000–12 000 words of core. This represents a greater cost to the buyer for the use of a compiler facility, and makes more attractive the use of compilers that are operational on large machines of computer centers. It should be,

noted that while many of the translators are termed compilers, many are actually interpreters: in these the input code is translated to an intermediate language that is interpreted during the execution of the program. The advantage of this approach is that less time is required during compilation and less storage is necessary for intermediate and final forms of the translated code. However, the storage and compilation time savings are paid for by the increased execution time characteristic of interpretive systems.

Applications Software

Today, application libraries contain program modules or subroutines written for control of certain laboratory instruments, I/O interface routines, typesetting machines, graphic displays, and specific industrial processes. Despite the contribution of the large manufacturers, most application libraries are user generated and supplied through user associations.

Many application programs are supplied as part of a total "turn-key" package which consists of a completely self-contained hardware/software system. Offerings exist for laboratory, data communication, industrial instrumentation, and specialized business systems.

Real-Time Monitor Systems

Motivation for a Monitor System: For many of the applications of minicomputers an operating system that supports the use of the computer is superfluous. These systems comprise program routines that aid in the preparation of application programs (i.e., assemblers and compilers), facilitate the performance of input/output operations (i.e., input/output control systems), assist in the debugging of programs (i.e., debug routines), and control the execution of individual program subsystems (i.e., executive systems, often also referred to as monitors). However, when the system operates in a real-time environment and must respond to interrupts from a number of sources, then some type of an executive system is needed.

Some recently described executive systems have advanced capabilities. Of particular interest are several systems which include a limited form of multiprogramming. Some of these permit foreground operation in real time while other activities occur in the background. For computers intended for both process control and other activities, the foreground is generally used for the control application while the background may include the compilation of programs.

The desirability of using a real-time monitor system is based on the economic advantage of sharing resources among different tasks, if possible. Considering the cost/performance tradeoff of storage devices (see Fig. 12) makes it quite apparent that whenever tasks become inactive for more than some specific time, they should be moved from the expensive high-speed mainframe memory to a much lower cost bulk storage device. Most real-time applications are structured to permit this type of reallocation of task resources: the real-time monitor system is the instrument that controls the reallocation operation. The performance characteristic of such monitor systems are perhaps best outlined by a specific example.

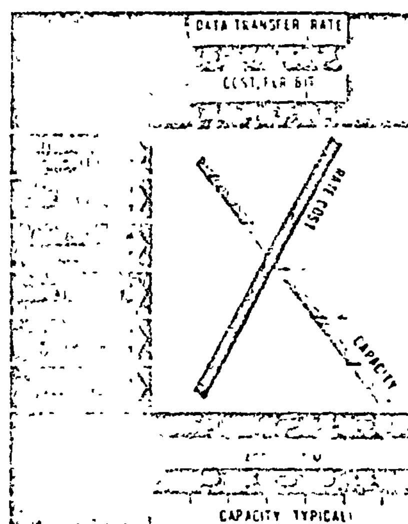


Fig. 12. Typical cost/performance relationship of store systems.

Example of a Multiprogramming, Virtual Memory System for Minicomputers:

Overview: The system described here for illustration supports a virtual-memory addressing scheme and a multiprogramming user environment on a Honeywell DDP-516 minicomputer. The system supports virtual addressing by providing a mechanism to convert virtual addresses to real mainframe memory addresses, a task that requires memory management if the addressed data are not currently in the mainframe memory. It manages memory by moving programs and data between the mainframe memory (core storage) and bulk storage (disk) on demand: it also provides a low-level interrupt handler for I/O. The multiprogramming support is provided in the form of the tables and memory management required to automatically switch control from one user to another without interference [10].

The various programs to be executed on the computer system are stored on disk as segments. The program segments are moved to core by the monitor system whenever they become active and are to be executed.

A segment is a contiguous block of storage which cannot be subdivided. It is the basic building block of programs and data. There are eight sizes of segments from 64 to 768 words, although 64 is the most common since all data files are organized into strings of 64-word segments. When a segment is allocated, it is assigned a unique name (ID) which is used to reference this segment. To implement the system, a segment also contains words of header or overhead for memory management purposes. This includes the type of segment, such as data, program, etc., its size, and an indication of usage to determine

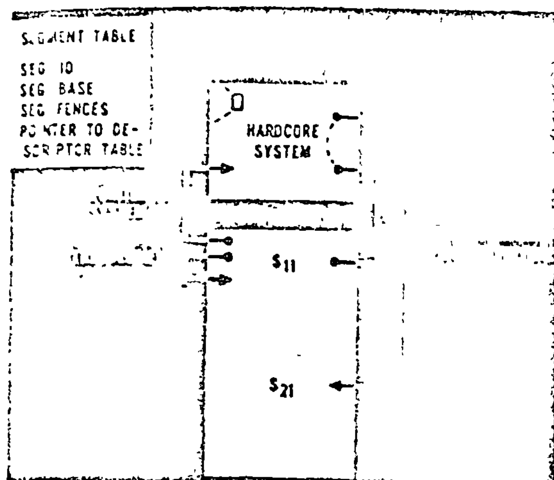


Fig 13 Memory addressing requirements. intrasegment, absolute, and virtual addresses.

which segments to throw out of core when more space is required.

The process of dynamically moving the segments from disk to core as they are referenced and back to disk when they are no longer needed is not apparent to the users, who get the impression of having a large virtual memory available. Of course, time is the price paid for not having all of a user's program and data in core at once But in a multiprogramming environent, while one user is waiting for a referenced segment to be transferred into core, another user will be making use of the processor.

There are problems associated with this type of dynamic memory management. For example, if a segment contains a program, its internal addresses must be relocated relative to the slot it occupies (relocation) after it is transferred into core. Also a mechanism must be provided to permit one program segment to call or address another (linkage). There are also memory management problems For example, fetching a segment from disk requires that one or more segments be pushed out of core to make room for it. These pushed segments may not be adjacent, so the system shifts the segments in core until all holes obtained by pushing segments are adjacent and can be merged. The new segment can then be transferred into core.

It has proved convenient to provide four distinct addressing modes for segments (see Fig. 13).

1) Intrasegment addresses provide a method for program segments to reference themselves

2) Absolute addresses allow segments to reference the system.

3) Virtual addresses are the generalized intersegment linkages.

4) Direct addresses are a specialized form of intersegment linkage.

Let us discuss these addressing modes in turn.

Addressing Modes: The *intrasegment address* is the normal type of intraprogram reference—a reference to data assembled into a program or a transfer of control within the program. The problem arising from the use of segments is address relocation– the addresses must be correct wherever the segment happens to be located within core at the time it is executed. Relocation difficulties have been avoided by using the index register as a base register. The index register is loaded with the address of the currently executing segment. Thus, all memory-reference instructions and indirect addresses which point within the segment are assembled using the relative address within the segment and have the index bit set. Notice that this scheme works since indirect addresses contain the index bit which are automatically set by the assembler.

The fact that a memory-reference instruction has a 9-bit address field makes it convenient to limit program segments to 512 words, which is quite a reasonable limit considering the small size of core memory. However, larger segments may be written by making appropriate use of indirect addresses.

The *absolute address* points to a fixed core location in sector O which is part of the area of the monitor system. To reach sector O, the index bit is simply set to zero. Absolute addresses are used to identify frequently referenced information. This use will be discussed below.

Virtual addresses provide intersegment linkage. They are interpreted by software routines and consist of two words. For example, assume that segment A contains a subroutine call to segment B. At assembly time the call statement is converted into a subroutine transfer to a system CALL program, followed by a two-word virtual address. The first word of the virtual address is the identifier (ID) of segment B. The second word contains the relative address (RA) within the referenced segment and the LL field (seven bits wide). The LL field is a "loose link" which speeds up the process of locating the referenced segment; its function will be explained below. The ID contains the address where the segment is stored on disk (13 bits; $2^{13}\times 16 = 500$k) and the segment size (three bits, designating one of eight possible sizes).

The *direct address* is a more privileged addressing mode. It also acts as an intersegment link, but it is simply an absolute pointer to a segment. Hence, it must be updated each time the referenced segment is removed. This requires that the referenced segment be held in core as long as any direct address points to it. This address mode is desirable because it is much faster than virtual addresses (software interpretation), even though it is substantially more difficult to set up and take down (greater system overhead). It is, therefore, used for linkages to data which are referred to frequently.

Direct addresses are explicitly set up by the program. By command within the program, a specified virtual address is interpreted to produce a direct address that is stored in a specified location in the thread block. Simultaneously, the direct address count of the referenced segment is augmented by one. This direct address count is

stored in the descriptor table which is physically located at the top of every segment (see Fig. 14). The program can then refer indirectly to the virtual address location via the address stored in the known location in the thread table. Whenever an existing direct address is removed, the direct address count in the descriptor table of the segment containing the address is decreased by one. Thus, whenever the direct address count is zero, there is no reference to that segment by direct address and therefore nothing that would require the segment to remain in core.

The Memory Management Routines of the Monitor System: The segments that currently reside in core are listed in the segment table of the monitor system (see Fig. 8). When a virtual address reference is made, a system routine searches this segment table to determine whether the particular segment (identified by its ID) is in core. This routine begins its search at the location pointed to by the LL field of the virtual address block. If the actual entry is different from that pointed to by the LL field, then the LL field is updated with the new location. The virtual address (consisting of ID and RA) is converted to the proper core location by adding the relative address (RA) to the base address found in the segment table. If control is to be transferred to this new segment, the base register is loaded with the base address, and control is transferred to the relative address within the segment. If the LL field points to the correct entry in the segment table, then the conversion from virtual address to physical core address is relatively rapid (in the order to 20 μs).

If the corresponding ID of the desired segment cannot be found in the segment table, then the segment is not in core and must be fetched from disk. Remember that the ID is a segment identifier and also contains the information required to fetch a segment from disk (location and size). This greatly facilitates the fetching operation. Upon moving the required segment into core, the proper segment table entry is made in a vacant table location.

When core is filled with segments and a new segment is required, one or more of the in-core segments must be pushed (written onto disk or discarded) to make room for the new segment. The algorithm for choosing the segments to be pushed out of core is simple and uses the descriptor tables. A sequential scan of the descriptor tables produces push candidates. The scan begins where the last push scan ended and ends when successful pushes have yielded the desired amount of space. A candidate is pushed if and only if the direct address and interrupt address counts of the descriptor table are both zero. The leading bit of the segment-type entry tells whether the segment must be written on disk when it is pushed or can be simply discarded.

When enough segments have been pushed out of core to make the desired amount of space, the holes left by the pushed out segments are gathered at the top of core. This is accomplished by moving all the segments above the holes down over the holes. Moving the segments in core requires that all direct addresses be changed to reflect the core shift. These include the segment addresses in the

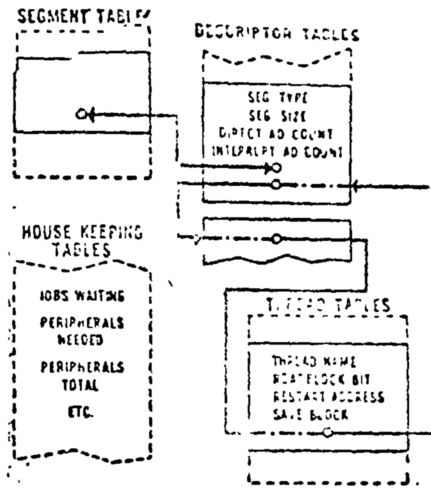


Fig 14. Additional tables used by the monitor system for implementing a virtual memory and multiprogramming environment. The descriptor table contains such items as the segment type, segment size, direct address count, and interrupt address count.

segment table, the users' call pushdown lists, the direct addresses in the thread table, and various pointers to system subroutines which are relocatable. Since this is a long list and shifting the segments down is a long task (about 50 ms), an attempt is made to free a large block of space instead of just the amount requested. This makes the next few space requests easier to fill since a segment push and core shift are not required.

When a call to a subroutine is made, the return direct address of the calling segment and the direct address to the called segments are pushed on a list contained in the thread table, and the direct address count for the called segment is incremented. The direct address of the called segment is pushed to store the base address of that segment. This base address may be needed in combination with the return address of that segment should this segment execute a call to still another segment. Upon execution of a return, these direct addresses are retrieved from the list to find the return address from which point processing is to resume, and the count is decremented. Hence, all segments on user pushdown lists are locked into core, as are all segments pointed to by direct addresses established by the user. I/O interrupt handlers are also allowed to be segments; when in use, they are locked into core by incrementing the interrupt address count. This technique for keeping track of the return path of a series of subroutine calls intrinsically permits the use of reentrant subroutines.

At assembly time, programs refer to external segment locations by name. During load time, a name versus ID table is first constructed (during the first load pass); this table is then used to insert the virtual address into the segments (during the second load pass) and to construct the proper descriptor table entry.

Routines for Multiprogramming: Processor control is automatically allocated to various users in a multiprogramming environment In general, time can be allocated either by roadblocks or by interrupting events such as a timer. When allocating time by roadblocks, a user is processed until he needs I/O, such as a message from a teletype, or an out-of-core segment from disk. Then he is roadblocked until I/O completion, and the monitor sequentially scans the list of inactive users until it finds one ready for further processing Thus, service is granted on a round-robin basis, not according to some priority scheme.

The heart of the multiprogramming routines in general is a thread table (Fig. 14) which is continuously present in core. It contains an entry for each of the possible threads. Thus, a user is defined by his thread table entry. This forms the basis for the processor allocation described above.

Part of the thread table is moved into a predesignated area in core sector zero whenever a thread becomes active. The "thread save" block contains all the data and pointers required by system and user to implement pure procedure programs. Before a thread save block is moved into core sector zero, the save block in core sector zero is restored to the previous thread save block. This data movement constitutes most of the overhead involved in changing threads (in the illustrative example it takes about one ms). However, most of the roadblocks that occur when a thread is using hard-core system programs require only four of the data cells in the thread table entry to be in core sector zero. Thus, moving the thread table by parts can reduce the thread changing time by a factor of 20.

A thread can roadblock for several reasons. If the thread requests input or output, a roadblock occurs and the I/O proceeds under interrupt control. When the I/O is complete, the thread is no longer roadblocked. A thread can also address a segment which is not in core, which causes a roadblock until the segment is brought in from disk. If a thread is still roadblocked (I/O not completed yet) when its turn comes around again, it will be skipped. Thus a thread is given control only when its roadblock is removed and its turn comes around.

The desirability of moving the thread table by parts into core sector zero is shown by the following example. When an out-of-core segment is addressed, a thread could be roadblocked three or four times for things like reading the disk ID table, making space for the segment, and finally transferring the segment into core. Only after the segment is in core and the address is about to be computed is the complete thread save block required to be in sector zero.

There are several other interesting roadblocks which can occur. For example, a low-usage program may be more compact and simpler if it is not pure procedure (required by multiprogramming). This is allowed by using a GATE statement at the start of the program The gate allows only one thread to be in the program. Any other threads that tried to enter would be roadblocked until the first thread opened the gate on its way out.

When the system is otherwise idle it scans the thread tables for a thread that is not blocked (roadblock bit = 0). When such a thread is found, its four temporary data cells are transferred to core sector zero, and the thread is restarted at the address specified by the thread table entry. This restart address is always within the hard-core system; before control is passed to an outside program segment, the thread save block is moved into core sector zero.

Routines for I/O: It is customary to perform all input/output operations under the interrupt system with the aid of the I/O table. This table contains an entry for every I/O device attached to the computer, specifying the interrupt handler address for the device, the buffer address, the buffer size, the buffer cursor (current character pointer), escape character, the initial character, and the link to the thread using the I/O device.

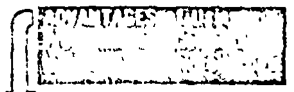
When input or output are desired, the appropriate program is called with the buffer segment and the escape character supplied as arguments. The called I/O program then fills in the I/O table entry, primes the I/O device, and roadblocks the thread. When an interrupt occurs, the system gives control to the location specified by the interrupt handler address in the I/O table entry for the interrupting device.

On interrupt, a character is transferred between the I/O device and the buffer specified in the I/O table, using the buffer address and the current character pointer. When an escape character match or full buffer are encountered, the thread table pointer in the I/O table entry enables the program to clear the roadblock, and the I/O function is complete.

To accommodate a large number of different I/O devices, with only a few of them active at one time, interrupt handlers are allowed to be program segments. When an I/O device becomes active its interrupt handler segment is fetched and locked into core for the duration of its activity.

The key I/O routine for the system is the disk handler. Disk I/O is controlled by the disk I/O queue which contains a maximum of twenty entries, one for each disk I/O request. Each entry contains information such as the disk address, the core address, and the thread table pointer. A disk transfer is initiated by finding an empty queue entry and inserting the address of the appropriate disk transfer program and its arguments. The requesting thread is then roadblocked.

The disk I/O handler is an autonomous process which goes on in the background of thread processing. When a disk I/O task is finished, the current disk rotational position is read and the disk addresses on the disk I/O queue are scanned to pick the I/O operation that can be performed most immediately (i.e., the task with the least latency). Control is then given to the chosen entry's disk transfer program, which sets up the disk I/O. Upon completion of the task, the thread is unblocked using the thread table entry address in the queue entry.



- ACCELERATED SYSTEM DEVELOPMENT
- HIGH GROWTH POTENTIAL
- LONGER USEFUL LIFE
- LOWER COST

Fig. 15. The merits of a stored program controlled system.

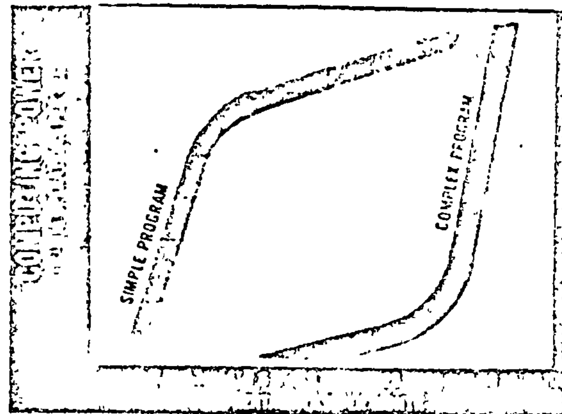


Fig. 16. Illustrative cost per computation for simple and complex data-processing tasks as a function of the computer system size (expressed in terms of rental cost).

V. Application Areas of Minicomputers

The Merits of Minicomputer Applications

The explosive growth of the minicomputer market is a result of employing minicomputers in a new set of applications that were previously unexploited by the larger computer manufacturers.

In the past the larger computer companies first focused most of their attention on commercial uses for data processing. These companies pioneered the application of computers in finance, manufacturing, and marketing. More recently, they have made the computer encroach progressively into the engineering design process. Initially, the computer was merely put to the task of analytical computing, taking advantage of its most obvious capability; then it was more initimately linked into the total engineering design cycle with the introduction of improved man–machine communication techniques such as graphic input devices and problem-oriented languages.

Still, the larger computer manufacturers, while they have great computer experitise, have in general had neither the knowledge nor inclination to automate real-time processes such as the control of chemical plants, the control and acquisition of data from scientific experiments, and the automation of test processes. These applications, for the most part, have been served by companies who have an understanding of the process. It is this type of organization that has pioneered the bulk of the minicomputer applications, predominantly in the area of real-time applications.

There are five compelling motivations for using a programmable minicomputer instead of specialized hard-wired logic (see Fig. 15).

One of these motivations is the possibility of developing the hardware portion of the application system concurrently with the software portion through a clearly definable interface, i.e., instruction repertoire and timing diagrams. This can significantly accelerate the development of the system. Often written for a computer data installation is a simulation program that permits extensive debugging and system testing long before the hardware system is operational. This capability is a highly effective development aid, since the software system usually contains the details of the application which are always subject to substantial changes during the system definition stages. Such a simulation program also permits the writing of diagnostic programs that can be used for the efficient debugging of the hardware system.

As a result of this limited yet crucial interdependence between the hardware and the software portions of a system, the system offers a high potential for growth, and associated with this, a much longer useful life. Experience has shown that most real-time applications are dynamic in their capabilities. Each successful application reveals other associated applications that are economically, technically, or administratively desirable.. Programmable systems provide the flexibility to economically accommodate these ever-changing system requirements.

Despite their flexibility, contemporary programmable systems often even represent the lowest cost implementation. This is probably the most important motivation of all for using minicomputers. The possibility of sharing resources with a programmed system (as was discussed in Section IV) and the use of general-purpose integrated components made in large quantities (as discussed in Section II) are the two main reasons for this cost advantage.

The Economics of Minicomputer Applications

The relationship of the economy of size in computer systems has been widely discussed [11], [12]. It is generally agreed that the cost effectiveness increases with the size of the computers (see the trends depicted in Fig. 16). What is interesting to note are the points of diminishing and increasing returns For complex programs, such as accounting, it has been shown that a minimum configuration is needed to reach a region where the economy of size is significant. However, it can be argued that for simple programs, such as that for communication which uses an elementary instruction set and few peripheral devices, a point of diminishing return clearly limits the realizable economy of size. For these applications it appears that minicomputers clearly offer the most economical solutions, since they lie below the point of diminishing return.

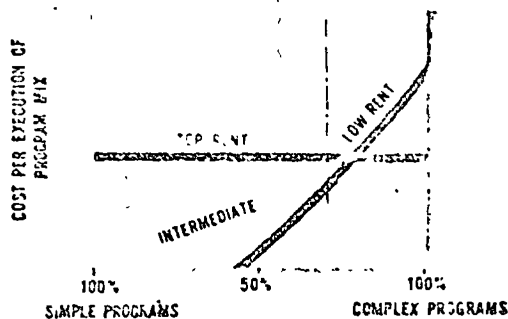


Fig. 17. Illustrative cost per computation as a function of the complexity-mix of the data-processing tasks, shown for different size computer systems.

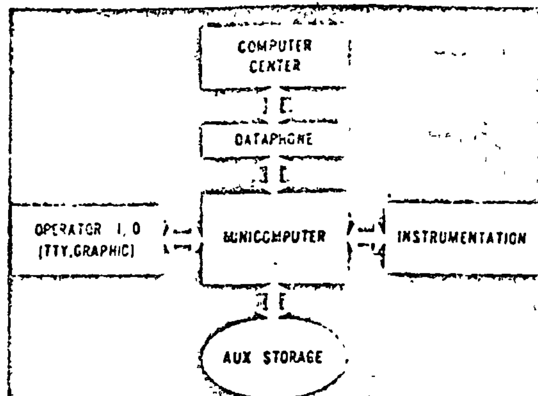


Fig 18. Components of the most general minicomputer system configuration. The components include a minicomputer, I/O devices for man-machine communication, I/O devices for control applications (sensors and controllers), bulk storage devices, and devices for communicating with other computers.

What are the consequences of these economy-of-size trends in a data processing environment? Let us consider a hypothetical example (refer to Fig. 17).

Assume that a "low-rent" computer is operating with a program mix of which 20 percent are simple programs. Assume next that the program load doubles, requiring either two "low-rent" computers or one "intermediate-rent" machine. If the program mix ratio remains the same, then it is more economical to trade up to the larger computer. However, if the program mix increases in simple programs (e.g., 50 percent), then it is clearly preferable to add a second "low-rent" computer. In fact, it may be even more economical to replace one of these computers with a minicomputer and the other by a slightly more powerful machine specifically designed to process complex programs only. Most minicomputers handling the communication tasks of large data processing installations (i.e., front-end communication processors) are doing just that.

Typical Minicomputer Applications [13], [14]

It is useful, for discussion purposes, to group the minicomputer applications into the following categories: communication, control, laboratory, and data processing. In all of these applications the minicomputer operates in an environment where real-time responses are important. This is evident in the application of minicomputers to operating departments where these systems are already used to transact routine business (e.g., inventory control or credit verification).

Generically speaking, all minicomputer systems comprise a combination of the following elements (see Fig. 18): 1) one or a multiplicity of minicomputers that provide the central system control, 2) input/output devices that furnish an effective man-machine communication facility for such uses as operator control, 3) bulk storage, such as disk files, to augment mainframe memory so as to permit the economical implementation of effective system-operating and application software, 4) communication equipment for gaining automated access to other computer systems for such purposes as augmenting the compute and storage capabilities of the minicomputer system, and 5) instrumentation that couples the system to other systems such as a manufacturing plant. Consideration of a few specific examples within the applications areas listed before will illustrate the use of these elements.

Data Communications Applications: The cost of small computers is decreasing faster than the cost of data communication facilities. Notwithstanding this, the utility of centralized data bases and extensive program libraries available only with large computer systems offsets the added communication line charges of using large remote-access computer systems instead of using small, locally installed systems. However, most large computers were designed basically for batch processing, and the concept of high-speed real-time interaction with these machines often has been added as an afterthought. Thus, when it is attempted to use these machines for systems such as airlines reservations, time sharing, and message switching, it is found that it is relatively easy to burden the large processor with the simple tasks of handling communication lines, attending to external interrupts, and interrogating large data files, thereby leaving no time for the basic computation that may need to be done and for which these computers were specifically designed. The realization of this has led to the off-loading of the simple jobs handled by large machines onto minicomputers which can be economically dedicated to the high-speed but relatively simple tasks and which provide a system of greater cost effectiveness.

Data communication applications can be meaningfully grouped into pure telecommunication applications where minicomputers operate as an integral part of a communication network, and preprocessing (front-end) applications where minicomputers provide a flexible interface between input/output devices and a data processing facility. In some configurations the two applications are

integrated into the same minicomputer systems [15], [16].

One telecommunication application is for message switching. Available telephone communication systems using line switching facilities appear to be economically and technically inefficient to meet typical communication requirements of interconnected computer systems. The traditional method of routing information through the common-carrier switched network establishes a dedicated path for each conversation. With present technology, the time for this task is in the order of seconds. For voice communication, this overhead time is negligible. But in the case of many short transmissions, such as may occur between computers, this time is excessive. To meet the requirements for burst-type communication profiles, it is preferable to employ wideband leased lines over which messages are routed by computer-controlled switching equipment according to the address that each message carries. Minicomputers have been effectively used for implementing message-switching networks of this type.

Another telecommunication application is for data concentration, which is often made an integral part of message-switching networks. Data concentrators subdivide a high-speed communication channel into several low-speed communication channels to realize some of the economy of size in the telecommunication field Minicomputers have been found to be an effective method for multiplexing in time several low-speed channels onto one high-speed channel.

Still other telecommunication applications use minicomputers for controlling switching matrices of private branch exchanges (i.e., private switchboards), and for the digital processing of signals for such uses as reduction of data redundancy. In all telecommunication applications the minicomputers are often used for such auxiliary tasks as converting codes and data rates, inserting error control information, formating and assembling characters and messages, performing echo check control, conducting traffic accounting, and accumulating network statistics.

Let us now turn our attention to the preprocessing type of minicomputer applications. Several functions must be provided at the remote-access computer site facility to control data communication. They include means for assembling bits to characters and then the characters to messages, means for converting character codes (e.g., EBCDIC to ASCII), means for controlling the communication lines and the input/output devices attached to them, and means for buffering the messages to smooth out the processing workload. This is particularly necessary when the messages arrive in a random fashion and in fluctuating quantities, as in commercial time-sharing systems. These functions are increasingly delegated to preprocessor minicomputers which communicate only completed messages to the data processing computers.

Additional functions, which benefit neither from a complex and extensive instruction repertoire nor from the considerable bulk storage devices available on the large data processing computers, are often also delegated to a minicomputer preprocessor. These functions include

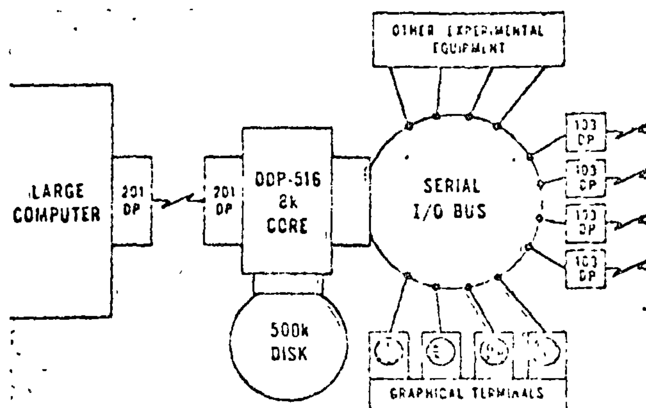


Fig. 19. A typical minicomputer system providing a flexible interface between I/O devices and a general-purpose computer center.

editing capabilities, interactive compilation systems, and centralized control functions for sophisticated input/output devices.

The use of minicomputer preprocessors not only increases the cost effectiveness of most systems, but also greatly facilitates interfacing equipment from different manufacturers. Without preprocessors the technical problem of interfacing is usually a very difficult one, involving both hardware as well as software incompatibilities. Today minicomputer preprocessors for all major data processing systems (e.g., IBM 360, Univac 1108, Burroughs B-5500, and Control Data CDC-6600) are available commercially.

A rather effective and advanced use of a minicomputer configured to interface graphical terminals to a large batch-processing computer is shown in Fig. 19. It uses a system software system of the type described under the section on real-time monitor systems. The small computer can provide the graphical terminals with real-time processing for generating, editing, and manipulating graphical or text files. The small computer passes requests along to the large computer for large tasks and provides access to the data base in the large computer. The configuration also provides remote concentration. The terminals are connected to the system directly or through several low-speed data sets (300 bits/second). The small computer then is connected to the large computer through a single higher speed data set (2400 bits/second). This configuration reduces communication costs for a group of terminals located remotely from the large computation center. Thus, the system appears as a large autonomous file facility to the minicomputer systems which can also be attached to the system.

A minicomputer system of this type has been constructed [10]. In its present form it consists of a 16-bit Honeywell DDP-516 with 8k of core, and a 500k fixed head disk. The system accommodates up to eight simultaneous users and allows them to access program and data segments on disk as if they were in core. Thus the core memory appears to the user to be very large. The DDP-516 with its disk connected through a high-speed direct memory access port provides the hardware founda-

tion for a virtual memory system. A 201 Data-Phone* set with automatic calling unit provides communication with the large computer, which is a General Electric 635 operating under the GECOS III operating system.

Laboratory Applications: The automation of laboratory instruments by computers is an accomplished fact. Whether the automation computer is a minicomputer dedicated to a specific application, or a large time-shared system that services several experiments simultaneously, depends on such factors as the rate at which data are taken, the accuracy required, and the computational complexity of data reduction. It also depends on such managerial questions as budgets, the willingness (and desirability) of scientists to learn simple programming, the availability of support personnel, and the rate of change of the laboratory environment. In either case the system requires an effective man-machine communication interface that facilitates control of the automated laboratory.

A properly automated laboratory leaves the scientist or engineer more free time to spend on truly creative parts of his job, i.e., research and development. Such a system can free people from the time-consuming chores of collecting data. At the same time it can improve the accuracy of the data by removing human errors resulting from the performance of repetitive operations. In addition to speeding up the acquisition of data, the data reduction process can be expedited and facilitated to become a real-time process. Techniques of information theory can be applied to the data in real time to immediately produce results that are mose meaningful and lend themselves best to interpretation. These techniques include filtering, spectral analysis, correlation analysis, signal-to-noise enhancement, and statistical estimation.

The automated laboratory can open up new experimental dimensions. It makes possible experiments with more variables, faster measurement rates, and more data. This permits the investigation of phenomena heretofore inaccessible by classical instrumentation.

There is a growing acceptance of dedicated minicomputers for controlling the instrumentation of laboratories. These computers are usually more economical than large time-sharing systems which require costly executive monitor systems, involve sophisticated input-output devices, and necessitate complex interfaces. Also, dedicated minicomputer systems allow the user complete freedom of action within the limitations of his computer. They do not impose the limitations of large time-sharing systems, such as timing interferences between different experiments, and programming restrictions associated with the executive monitor.

Most programs of dedicated systems are written in assembly language. To enhance the computational capabilities of these systems, there is a growing tendency to interface them to a large time-sharing computer which provides computational capabilities from high-level languages, as well as large secondary data stores and powerful input output capabilities. This hierarchical structure of interconnected computers appears to offer immense potentials.

The applications of minicomputers for automating laboratory systems are many. In the physical sciences they include facilities for research in signal processing, for speech and visual research [17], for chemical analysis (e.g., gas chromatography), for nuclear physics studies (e.g., nuclear magnetic resonance), and for seismographic investigations (e.g., probing for oil sources) In the medical sciences they include facilities for the clinical laboratory (e.g., autoanalyzers), for automated waveform analysis (e.g., EKG and EEG), and for patient monitoring (e g., in the intensive care units). The dedicated automated laboratory even has found its way into the arts for such applications as musical composition and choreography, and will probably establish its place in areas not conceived of today.

Industrial Applications: Minicomputers have been controlling production facilities for some time. In some applications they are used strictly as a data source that transmits sequences of commands to hardwired numerically controlled (N/C) machines. These machines perform such tasks as metal working, assembling (e g , bonding or robot-type assembling), and mask-making (i e., photolithographic semiconductor device fabrication). In other applications minicomputers send actuating pulses directly to the servomotors attached to the machine tool. At the same time they may perform calculations for contouring and interpolating, and they may even control another machine tool.

Extensive computational power may be required to prepare the sequences of instructions for the N/C machines (e.g., for contouring N/C machines or N/C mask-making machines), and large amounts of core storage are usually required to implement higher level programming languages (as for APT and XYMASK). It is then customary to produce a compact instruction sequence on a large general-purpose computer and submit this sequence to a minicomputer for post-processing. The results may be retained in the minicomputer system which then directly controls an N/C machine; often the results are punched out on paper tape which is then mounted on the N/C machine for execution.

Virtually all computer-controlled test facilities employ a minicomputer for generating the sequences of test instructions that control the instrumentation (e.g., voltage sources, current sources, pulse generators, and voltmeters). In these applications the minicomputers also analyze the results, which may determine the subsequent test steps to be executed and automatically produce a log of the items that have been tested.

Minicomputers are also extensively used to regulate the flow in continuous operations, such as the flow of material in chemical processes and the flow of electricity in power systems. Two approaches are used - set point control and direct digital control. In set point control, which is the traditional approach, the computer serves only as a supervisor for the analog controllers, adjusting the desired values for the process (i.e., the set points). If the minicomputer or one of its associated peripherals fails, the process can continue operation under manual

control, since the analog controllers are present. In the more recently introduced direct digital control approach, the analog controller functions are performed by the minicomputer via software programs executed by the computer. The minicomputer drives the process actuators directly, using operational amplifiers to maintain the proper driving condition between successive output calculations for each loop. In this approach, if the minicomputer or an associated peripheral fails, the entire process is without control unless extensive backup facilities have been provided.

The required computer performance depends on the rate of process variation. High rates of variation require higher measurement sampling rates and higher rates of control commands than slow rates of variation. Also, the direct digital control approach places a much larger computational load on minicomputers than does the set point approach, since the computer must simultaneously serve a multiplicity of control loops. To implement this multiple control loop environment, each control loop comprises the control programs and a status table that contains a listing of the control parameters for that loop, intermediate values which are carried forward from previous control calculations, and branch codes that indicate which of several control algorithms is being applied. However, there is an increasing trend toward implementing the more sophisticated control systems by the direct digital control approach, thereby avoiding the expense of complex analog hardwired controllers and affording the flexibility which permits the control systems designer to adjust the control methods and parameters to obtain a more efficient process operation at minium cost. Contemporary direct digital control systems may have typically from 50 to 1000 control loops.

Still another increasingly important industrial application of minicomputers is in the area of material handling and warehousing. For example, minicomputers are now used to control the full operation of a fleet of forklifts and picking vehicles. Each vehicle communicates automatically with a centralized control minicomputer up to several times a second, reporting its exact position in the warehouse. The control computer sends the traveling and operation commands directly to each vehicle via completely buried cables. The cabling both guides the vehicles accurately and allows two-way data communication between the control computer and each vehicle. Such an automated system offers several advantages; they include faster and more accurate order processing, higher vehicle usage, possibility for better inventory control, and improved space utilization.

Similar material movement systems exist using stacker cranes and rail-based carrier vehicles. Each carrier is marked so that a control computer can activate the proper switches for each carrier.

The use of minicomputers in industrial applications promises to facilitate the administration of industrial activities by automatically supplying timely and accurate production and inventory data to administrative systems (i.e., mangement information systems). These systems can be programmed to initiate routine activities automatically and single out significant or exceptional data on which management must act.

Administrative Applications: The administrative applications, as understood here, include computer-based systems for production management (e.g., activity scheduling, activity reporting, order processing, inventory control, and material management), personnel managment (e.g., payroll, status reports, vacation and sick leave, personnel inventory, labor distribution, turnover statistics, schedule of hours report, and audit reports), fiscal management (e.g., invoicing accounts receivables, general ledger accounting, nonoperating expenses, operating expenses, and budget control), and statistical analysis (e.g., sales analysis, sales forecasts, production planning and simulation). Using minicomputers for these applications has been only a recent development, largely for two reasons: 1) the minicomputer manufacturers are just beginning to commit resources to the provision of training, product servicing, and software support as required by small commercial data processing installation, and 2) low-cost peripheral equipment for business applications that make minicomputer business systems economically attractive are only now becoming available. In most business applications the price/performance ratio of the peripherals, the software provided, and the I/O speeds of the central processor are far more important than the computation features of the central processing unit.

The chief advantages in the use of minicomputer systems for administrative applications are their low cost and rapid response time, provided the systems come equipped with the applications programs that completely meet the user's needs. Many organizations that could not justify a large data processing center find that they can easily afford a minicomputer installation that comes as a turn-key package where the program development cost is amortized over many customers. Such a dedicated minicomputer system located directly in a user department usually can then be made much more responsive to that department's needs than a centrally operated computer center run in the batch mode ever could. Time-shared systems will eventually be capable of providing the kind of responsive service obtainable with dedicated minicomputers. However, the use of minicomputers promises to facilitate the evolution to large, integrated computer networks. This is based on two opposing applications that have recently emerged.

There is an increasing trend toward centralizing data processing in large and sophisticated computer centers that are staffed and operated quite autonomously by exceedingly capable specialists. These centers strive to increase their processing efficiency by making best use of the most modern equipment available and by developing highly sophisticated program systems. Eventually, the minicomputer centers will probably want to tap the sophisticated resources available only at these large computer centers and to make increasing use of them. They will gain access to these resources by highly efficient data communication networks designed to meet the specific

needs of computer-to-computer communication. Thus, minicomputer systems can be viewed as rather autonomous system elements of large scale computer networks which permit the evolutionary installation of complex computer systems.

In the process of attaching minicomputer systems to large computer centers, the actual data processing load of the minicomputer system will probably decrease. Still, it will probably be advantageous to maintain some files locally so as to not swamp the central computer with excessive details, which could make the composite file unmanageable, and limit the data flow between computers. Thus, minicomputer systems will probably assume increasing importance in the area of data communication and data base management, and gradually lose some importance in the area of actual data processing.

Despite the gradual loss of the functional importance of minicomputer centers, their use promises to make the realization of complex management information systems practical. The reason is that the use of minicomputer centers permits breaking the total data processing job into manageable proportions. It permits the development of a modular information system where the interaction between modules can be minimized and where each module can be made to pay for itself before the next module is added. One benefit of this modularity is the possibility of using individual applications long before they have been perfected and completely integrated into the total system, thus facilitating a think-do-think-do cycle which enhances the controlability of a project. Another benefit is a reduction of the total capital investment required to achieve a flexible information system, since individual applications can be made to pay for themselves as soon as they are operational. Still another benefit is a reduction in the probability of failure of the system development, since the individual modules can be carefully tested and individually debugged.

Other Applications: A rapidly increasing number of dedicated systems for a variety of other applications where minicomputers play an integral part are now being developed.

For example, minicomputers are used to control keytape and keydisk systems in the preparation of data-processing data through keyboard entry devices. Instead of punching the data on cards on these systems, the data are directly assembled on magnetic tapes or disks from which point they are then transferred to the data-processing system. Minicomputers are usually used to control several keyboards in combination with one bulk storage device.

They are also part of large data-processing systems where they increase the throughput for such functions as computation of the fast Fourier transform and data sorting.

Furthermore, minicomputers are being applied to typesetting tasks where they control the justification and hyphenation of text, and the preparation of input media for typesetting machines such as photocomposition equipment.

VI. Epilog

Minicomputers are here to stay. They represent a new and powerful building block in their own right and will find increasing use. In fact, this trend will be accelerated because of the continuing decrease in the cost of minicomputers as a result of progress in semiconductor and memory technology fields.

Eventually, many minicomputers which are now external units will become an integral portion of the system, actually built right into many of the individual devices. And because of the increasingly higher efficiency of the central processing unit, many of these minicomputers will be operated in a multiprogramming mode to make the best use of the peripheral devices attached to them.

Minicomputers will also have a profound effect on the component technology. They permit the definition of functional blocks of increasing complexity without making these blocks narrowly specialized, and thus unlikely to reach large production levels where the startup costs become insignificant. Minicomputers, because they are usually sold in larger quantities, generally offer more integrated electronics part numbers than do the conventional large computers. This helps realize the economies of integrated electronics even more, even for nonmemory components. This will result in minicomputer structures that are even lower in cost. One day an entire processor will be fabricated on one chip at a cost of perhaps $10–$20. With from 1000 to 5000 MOS devices per chip, this is completely within the realm of possibility today. Combining this processor with a ROM with less than 50-ns access time will make the resulting minicomputer also substantially faster than present models.

Thus, minicomputers are becoming the embodiment of tomorrow's functional devices. In fact, people will eventually wonder how they ever did without them. To make this happen is one of the great challenges of the 1970's.

Acknowledgment

This paper reflects the very valuable discussions held with W. F. Chow, C. Christensen, A. D. Hause, and H. S McDonald of Bell Telephone Laboratories; G. C. Henry and C. B. Newport of the Honeywell Computer Control Division; N. S. Zimbel of Arthur D. Little, Inc.; S. A. Goldstein and R. Denzau of the Diebold Group, Inc.; and many others with whom the author has had the privilege of interacting. The author would also like to acknowledge the valuable comments received in the preparation of this paper from K. M. Poole, the editorial help from Mrs. E. Blair, and the patient typing assistance from Mrs. N. Firestone.

References

[1] N. S. Zimbel, "Outlook for minicomputers, 1969–1974," Service to Management Report, Arthur D. Little, Inc., March 1970.

[2] J. J. Bartik, "Minicomputers turn classic," *Data Processing*, p. 42 ff, January 1970.

[3] M. E. Hoff, "Impact of LSI on future minicomputers," presented at the 1970 IEEE Convention.

[4] F. J. Langley, "Small computer design using microprogramming and multifunction LSI arrays," *Comput. Design*, April 1970.

[5] F. D. Erwin and J. F. McKevitt, "Characters—universal architecture for LSI," in *1969 Fall Joint Computer Conf., AFIPS Proc.*, vol. 35. Montvale, N. J.: AFIPS Press, 1969.

[6] *A Pocket Guide to HP Computers*, Hewlett Packard Inc., and *Small Computer Handbook*, Digital Equipment Corporation.

[7] S. B. Dinman, "The direct function processor concept for system control," *Comput. Design*, March 1970.

[8] G. Bell *et al.*, "A New architecture for minicomputers," in *1970 Spring Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

[9] *A Pocket Guide to Interfacing HP Computers*, Hewlett Packard, Inc.

[10] C. Christensen and A. D. Hause, "A multiprogramming, virtual memory system for a small computer," in *1970 Spring Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

[11] K. E. Knight, "Evolving computer performance 1963-67," *Datamation*, January 1968.

[12] B. Schwab, "The economics of sharing computers," *Harvard Bus. Rev.*, vol. 46, no. 5, September-October 1968.

[13] F. F. Coury, "A systems approach to minicomputer I/O," in *1970 Fall Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

[14] C. B. Newport, "Applications and implications of minicomputers," in *1970 Fall Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

[15] L. G. Roberts and B. D. Wessler, "Computer network development to achieve resonance sharing," in *1970 Fall Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

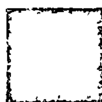
[16] F. E. Heart *et al.*, "The interface message processor for the ARPA computer network," in *1970 Fall Joint Computer Conf., AFIPS Proc.*, vol. 36. Montvale, N. J.: AFIPS Press, 1970.

[17] P. B. Denes, "On-line computers for speech research," this issue, pp. 418-425. See also other contributions on the use of computer-based laboratory systems in this issue.

sifting the minis

A TECHNIQUE FOR SELECTING SMALL COMPUTERS

by Robin T. Ollivier



Computer salesmen have multiplied nearly as fast as the machines they sell. The systems engineer selecting a giant number cruncher probably doesn't recall any selection problems—he never drew a sober breath. On the other hand, I've been installing minicomputers. The salesman takes *me* to the automat—I have to contend with dyspepsia, as well as headaches from reading fine print.

The marketing principle implied in this little story demonstrates the necessity of having a quick, analytical method for comparing small computers. It has to be quick. One can't spend $20,000 worth of engineering time to buy a $10,000 computer. It has to be effective. Different applications demand different approaches. As a matter of fact, each of the more than 30 cpu manufacturers thinks his uniquely designed product is the best for most tasks.[1]

terms and conditions

A selection technique is proposed in this paper that has proved both quick and effective. The assumptions on which this technique is based are listed below:

1. Qualified vendors will make competitive proposals.
2. Vendor proposals are factual.
3. The system designer has analyzed the problem to be solved.
4. Evaluators are capable of relating computer characteristics to a detailed task description.

The procedure may be summed up in the following definitions:

Basis. Cpu selection will be based on performance and on effective cost.

Performance. Performance (P) is defined to be the weighted sum of equipment and vendor capability.

Effective cost. Effective cost ($) consists of quoted price plus the software and engineering costs of implementing a given computer.

Equipment capability. The weighted sum of discrete computer characteristics. Each characteristic is evaluated on a 0-4 point basis.

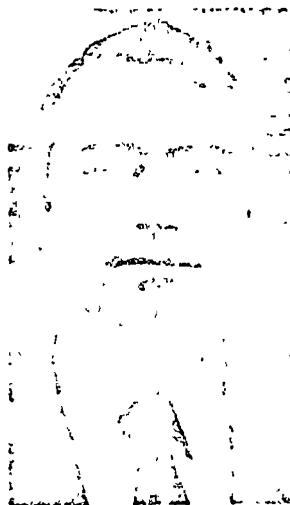
Vendor ability. The weighted sum of discrete vendor performance factors, each rated on a 0-4 point basis.

Quoted price. The number of dollars the buyer sees on the contract.

Software costs. The number of dollars worth of programming service required for this task plus hardware add-ons or deletions to the quoted equipment required by software.

Engineering costs. The number of dollars worth of engineer-

1. For a current survey of processors see one of the following:
D.J. Theis and L.C. Hobbs, "Mini Computers for Real-Time Applications," Datamation, v. 15, no. 3, 1969.
R. Ollivier, "Revolt Within the Rack" EDN, v. 14, 1969.
J. Cohen, "Mini-computers," Modern Data, v. 2, no. 8, 1969.



Mr. Ollivier is vice president of Digital Data Engineering in Pasadena, Cal., a systems and software house specializing in computer-based real-time applications. Mr. Ollivier is widely known for his bonhommie as well as for his data systems designs. He was formerly with the Jet Propulsion Laboratory, MESA, and with Burroughs Corp. He has a BS in engineering physics from the Univ. of Michigan.

Reprinted with permission from *Datamation*®, vol. 16, pp. 141–145, Jan. 1970. Copyright 1970, Technical Publishing Co., Barrington, Ill.

ing services such as design, fabrication, and documentation plus hardware add-ons or deletions to the quoted equipment.

Weighted sum. The relative importance of one factor to another is determined by assigning each factor a multiplier. The score for a given factor is multiplied by its weight and then added to the scores of other factors.

Factor. A discrete, measurable characteristic of some importance to the task at hand.

the method

Computer procurement (of machine, not by machine) proceeds in four distinct stages:

Design. The task is analyzed and one solution (from among many) is chosen. An envelope of constraints is defined. Minimum specs and maximum dollars are established for the computer. Initial hardware software tradeoffs are made. A list of qualified vendors is developed.

Solicitation. A problem description, minimum specs, and approximate job-scope information is supplied to vendors. Proposals are requested. Two to four weeks should be allowed for preparation. (Shorter times restrict competition: longer periods suggest larger systems and more extensive selection procedures.) Prior to receipt of proposals, the evaluators must complete the list of evaluation criteria and assign weights to factors. System design should be reviewed and refined during this period.

Evaluation. Scoring of proposals proceeds quickly. Impartiality is guaranteed by the objective requirements of the weighted factors.

Greater care must be taken in developing effective costs. Software kernel routines for key processes must be flow-charted or otherwise designed to provide a basis for estimating I/O timing, memory requirements, and manpower resources. Engineering and fabrication costs may be highly vendor dependent. The task should be broken down to the cost of each logical function or identifiable module.

The evaluation procedure succeeds in ranking proposed equipments according to their performance and cost. This data may be presented in two ordered lists or plotted as cost vs. performance. This completes the objective evaluation of competitive equipments.

Negotiation. Competing proposals have been analyzed and fairly evaluated. The analytical data forms the basis for final selection. The thought processes that effect this decision are subjective in nature. Only some of the more obvious considerations will be reviewed here.

Guidelines may have been established in the solicitation phase for a maximum effective cost and minimum performance score. Using these guidelines one might choose any of the following.

1. The best performer whose effective cost is less than the maximum.
2. The lowest cost equipment whose performance exceeds the minimum standard.
3. The equipment that satisfies cost and performance standards and has the highest performance/cost ratio.

a sample recipe

The method has been briefly sketched in preceding paragraphs. It is intended that this technique can be applied almost directly from the cookbook. An example of minicomputer selection is given in this section as a means of further defining the technique.

Task Definition. The dynamics of an object falling through a tube will be studied by analyzing time and pressure information. The data is generated by eight pressure ports located along the tube and as many as 16 presence sensors. Enough data will be taken to adequately define the pressure vs. time plot. Event-timing resolution to 5 usec is required. Placement of the pressure sensors, sampling strategy, number and relation of timing events, and extent of postexperiment analysis was (and is) still under discussion. The maximum number of samples for any run should be in the range of 1000 to 2000.

data acquisition

This description by the scientist resulted in the data acquisition system specification shown in Fig. 1. A programmed digital computer provides on-site control and calibration functions. Run data is buffered in core memory. Those runs requiring more analysis than that provided at the site are transferred serially by phone line to a data center computer. System design was undertaken. The approach was to develop a multi-application data logger with limited processing and display capabilities. The computer was required to provide for independent calibration runs and to permit rapid modification of sampling strategies. Analog multiplexing schemes were keyed to the occurrence of digital events, bit rates, order of channels. Number of channels, time offsets, etc. are selected by the experimenter. These options are selected by English language commands from a teletypewriter. This method of control provides a hard-copy record of each run.

1. Digital Inputs
   - 16 Channels
   - Pulse width 1 usec. minimum
   - Provision for electrical signal conditioning
2. Digital Outputs (number of bits in parentheses)
   - Enable/disable acquisition (1)
   - Start/stop A/D scan (1)
   - Set end scan channel (3)
   - Select sampling rate (3-7)
   - Select timer rate (4-9)
   - Set submultiplex channel (3)
3. Analog Inputs
   - 7 primary scan channels
   - 8 submultiplex channels
   - Scan rate to 50 KHZ maximum
   - Input impedance > 10 megohms
   - Input voltage ± 10 v full scale
   - Conversion to 10-bit digital
   - All channels single ended
4. Timing Measurements
   - Clock increments 8-16-bit register
   - Interrupt on overflow
   - Parallel read on command

Fig. 1 Data acquisition specification.

weighting the factors

Evaluation criteria for the computer were developed and weighted. The same was done for manufacturing aspects of the procurement. The results are shown in Figs. 2 and 3. Note that specifications are firmly tied to objective quantities.

Since we intended to build a single system—or at most two—we felt that the quality of the vendor was relatively important. We therefore assigned an over-all weight of two to the computer and one to the manufacturing criteria. Put

another way, the sum of the computer weights was twice as large as vendor weights. No attempt was made to get "neat" numbers for the sum of weights, only to see that the factors be reasonably related to each other. An agreeable way to start is by assigning a weight of one to the least important factor and proceeding comparatively up an intuitively known ladder of significance. Computer characteristics were weighted first. Then half the total of the computer

FACTOR	WEIGHT	SCORING BASES
Word Size	10	4: 16 bits or more; 2: 12 bits 0: 8 bits or less
Cycle time	6	4: 1 usec, 3-1: 1-2 usec 0: 2 usec
Instruction set	5	4,3: Extensive; 2. Adequate; 1-0. Primitive
Arithmetic	2	4: Hardware multiply/divide; double precision and floating point options; good precision 3-1: Adequate capability; hardware mul/div or fast subroutines 0: Very little arithmetic capability
Addressing	4	4-0: Score one for each of the following: indirect, relative, indexed, direct to greater than 4096, or by addressing
Programmable registers	6	4: Many; 3-1: More than one, 0: One
Interrupts	7	4: 3 or more priority, no identification necessary, 3-1. Adequate for 3 devices 0: None quoted
Input/Output	8	4: 2 or more automatic channels at rates to 1.3 megabits/sec, 3-1: At least one 1.0 megabits/sec with good accumulator I/O; 0: Marginal I/O capability
Physical size	1	4-0: Subtract one point for each 5 inches over 11 inches
Console	3	4-0: Sense switches, displays, debugging aids

Fig. 2 Computer criteria.

FACTOR	WEIGHT	SCORING BASES
Delivery time	7	4,3 Less than 45 days ARO, 2,1: 45-75 days ARO 0. Over 75 days ARO
Past performance	4	4-2: Many reports of on-time delivery and good service 1-0. Known for late delivery, poor service
Maintenance	3	4-2 24-hour turnaround on cpu, on-call maintenance, 2-0 No experience, remote or difficult corporate interface
Location	2	4. Southern California 2 Within 500 miles 0: Distant
Alternative sites	1	4. Same computer installed at JPL 3-1: Locally available 0: No Alternative site
Number installed	4	4: Over 100; 3-1. 10-100 installed 0: Less than 10 in field
Documentation & training	5	4: Excellent hardware and software manuals, or training provided 3-1· Adequate interface and programming manuals 0: Little or no documentation

Fig. 3 Manufacturer criteria.

weights were distributed among the vendor characteristics.

Factors are also rated as to their significance to the project. The following statements illustrate what I mean.

1. Most factors are significant to the project and *must* be scored whether or not the vendor supplies adequate information to do so in his proposal.
2. Some factors are peripheral in nature and need not be scored (changing the basis), or may be given a nominal score if insufficient data exists.
3. A few factors have critical limits. A zero score on any one of these factors would result in disqualification of that proposal.

In this example, three factors—interrupts, i/o capability, and timely delivery—had critical limits. A score of zero on delivery eliminated two machines, and redefined the model number of a third.

A vendor list of eight was prepared and proposals solicited. In due course the evaluation was completed. The results are summarized in Figs. 4, 5, and 6. The euphemism of numbered rather than named computers was used to spare the editors. However, the discerning eye may distinguish the "made in" Orange County, Framingham, or Maynard features.

For this particular task, characterized by few, but high rate, data sources, some interesting observations result.

1. Eight-bit machines were disappointing. i/o characteristics were not adequate or were relatively expensive.
2. An 8K memory was required for all machines with less

FACTOR/CPU	A	B	C	D	E	F	G	H
Computer								
Word	40	20	0	0	40	40	0	40
Cycle time	6	12	18	6	12	12	0	24
Instruction	15	0	10	10	15	15	15	15
Arithmetic	4	2	0	2	4	4	0	4
Addressing	12	8	16	16	8	12	4	16
Registers	12	0	24	18	18	12	0	18
Interrupts	28	7	21	7	21	14	28	28
Input/output	32	24	8	16	24	24	8	8
Physical size	4	4	4	4	4	3	4	2
Console	6	6	9	6	9	6	6	9
Subtotal	159	83	110	85	155	142	65	164
Vendor								
Delivery time	14	21	28	28	28	0	28	21
Past performance	12	12	8*	8*	16	4	8*	8*
Maintenance	9	6	3	12	9	6	9	9
Location	4	0	8	8	8	0	8	8
Alternative	2	4	2	2	2	4	2	2
Number installed	12	16	4	16	16	8	4	8
Training	20	15	5	10	15	15	10	10
Subtotal	73	74	58	84	94	37	79	76
TOTAL	232	157	168	169	249	179	144	240

*Nominal value, no data

Fig. 4 Evaluation results.

ITEM/CPU	A	B	C	D	E	F	G	H
Quoted	11.9	6.4	8.1	12.7	16.2	11.4	8.8	12.0
Software								
Programming	5.0	5.0	7.5	7.0	6.0	5.5	6.5	5.5
Modifications*	0	4.0	2.5	2.5	0.5	0	3.0	0
Hardware								
Interfacing	1.5	0.5	2.4	0.4	0	0.3	1.6	0.6
Modifications**	4.5	0	2.0	(1.5)	1.7	2.0	0	0
	22.9	15.9	22.5	21.1	24.4	19.2	19.9	18.1

*Modifications are for additional 4096 core memory, except E, additional level of interrupt.

**Addition of I/O channel, except D, deletion of special interface hardware. A represents upgrade to next model computer to get required I/O performance.

Fig. 5 Effective costs.

Cost (1000's $)	
B	15.9
H	18.1
F	19.2
G	19.9
D	21.1
C	22.5
A	22.9
E	24.4

Cost Maximum

22.0

Performance	
E	249
H	240
A	232
F	179
D	169
C	168
B	157
G	144

Performance Minimum

150

Fig. 6 Data summary.

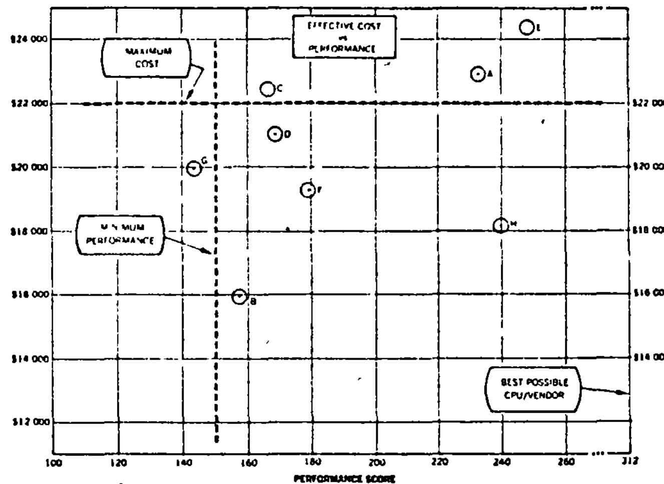


Fig. 7 Performance score.

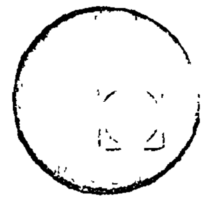
than 16-bit word length. Shorter words meant longer programs and double length buffers.
3. Programming costs were relatively tightly grouped. The nature of the job and the size of the computer dictated assembly language coding. Better instruction sets required longer learning curves.
4. Interface costs were difficult to estimate unless the vendor provided literature treating this issue in depth.

The cost vs. performance plot in Fig. 7 provides a good visual presentation of analytical results. Processors H and B are the most likely choices. Computer F is a possible but unlikely candidate. Note that the 8-bit computers C, D, and G deliver less bang per buck than the 16-bit cpu's E, A, and F. It is clear from this data that this task is not suitable for an 8-bit processor.[2]

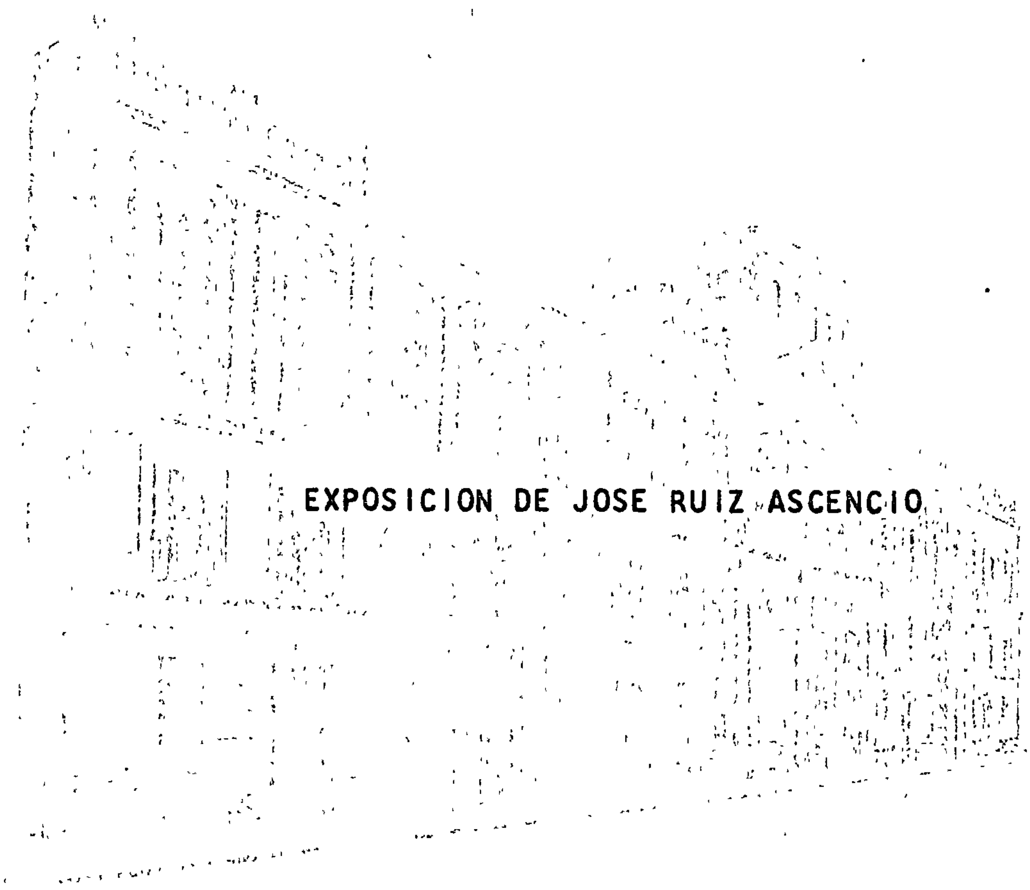
The results in this case were relatively straightforward. The technical group selected the best performer within the cost envelope. There was a sufficient dollar pad to allow for contingencies. Should some unforeseen fiscal calamity befall the project, management can quickly and reliably shift to a lower-priced computer.[3] ◻

2. A line of constant performance/dollar can be drawn for items C, D, and G and for items E, A, and F. This represents a de facto standard for this particular evaluation. It can be seen that such lines are roughly parallel and that the vertical separation is about $3,200. Thus even a 4K 8 bit machine is found to be a poorer performer per dollar than a 16-bit processor.
3. Several additional 16-bit machines received a cursory evaluation during and after the example procurement. One could quickly determine whether further evaluation was desirable.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



APLICACION DE LA MINICOMPUTADORA



EXPOSICION DE JOSE RUIZ ASCENCIO

PALACIO DE MINERIA
Tacuba 5, primer piso. México 1, D. F.
TELEFONOS: 513-27-95
512-31-23 521-73-35

CHAPTER 4

MINICOMPUTER PROGRAMMING WITH ASSEMBLERS AND MACROASSEMBLERS

INTRODUCTION AND SURVEY

Assembly-language programming will enable us to obtain the greatest possible effort from a digital computer, i.e., to optimize computing speed and/or memory requirements. This is because assembly-language instructions correspond, more or less, to the actual hardware operations possible with a specific machine and permit us to exploit its features cleverly. This advantage of assembly-language programming is especially pronounced for small digital computers, whose algebraic compilers (which must fit into 4K to 8K words of memory) may not produce very efficient code.

Modern symbolic assemblers not only *translate instruction mnemonics into machine code* but also permit *symbolic memory references* by assigning binary location numbers to symbols (Sec. 4-2). The better symbolic assemblers can also *compute addresses by evaluating symbolic expressions* (Sec. 4-3), can *reserve blocks of storage locations* (as well as single storage locations) for data or instructions, and can arrange for storage and formatting of decimal, double-precision, and floating-point data (Sec. 4-5). Good general-purpose assemblers further free the programmer from assigning program pages and work with a companion linking-loader program to facilitate *relocation and linkage of multiple program segments* (Sec. 4-17; see also Sec. 3-6). Finally, *macroassemblers* can generate useful multi-instruction sequences from one-line commands (Sec. 4-21) and, together with *conditional assembly* (Sec. 4-23), can combine some of the programming simplicity of a compiler language [illegible] assembly-language efficiency.

With a suitable operating system, assembly-language program segments can be neatly combined with FORTRAN programs (Sec. 4-20) so that even a little knowledge of assembly language can be used to improve important or frequently used routines.

ASSEMBLY LANGUAGES, ASSEMBLERS, AND SOME OF THEIR FEATURES

4-1. Machine Language and Primitive Assembly Language. A typical program sequence for a 12-bit minicomputer, say

2	5	
3	...	
4	LOAD INTO ACCUMULATOR (the contents of)	2
5	INVERT ACCUMULATOR	
6	STORE ACCUMULATOR IN	3

specifies the contents of successive memory locations 2, 3, 4, 5, *and* 6. Location 2 contains a *data word* (5) given by our program, but location 3 is only *reserved* for an as yet unspecified data word to be stored there by the program. The program proper (i.e., the first instruction) starts at location 4. *The program counter will be initially set to* 4 and will step to 5, 6, and on to 7 as each instruction is executed.

Such a program is actually entered into the computer in binary machine language, viz.,

000	000	000	010	000	000	000	101
000	000	000	011	...	...	...	...
000	000	000	100	001	000	000	010
000	000	000	101	111	000	100	001
000	000	000	110	011	000	000	011

perhaps from a binary paper tape or from front-panel toggle switches. The first 12-bit word on each line is the memory address of the second word. The first line again locates the data word (5). The second line reserves location 3 for a data word which is not supplied by the program, but will be stored there at run time by our last instruction; some assemblers would deposit 0 in such a location for the time being.

The first word of the third line is, again, the address of the second word. This time, this stored-program word represents an *instruction code* and, since this is a memory-reference instruction, some address bits needed to determine an effective memory address. In our simple example, the five leading instruction-code bits 001 00 signify LOAD INTO ACCUMULATOR with the "page 0" direct-addressing mode (Sec. 2-7). In this case, the remaining seven address bits 0 000 010 directly represent the binary address. The remaining two instructions are similarly translated.

To work with long programs in this machine-language form would be decidedly uncomfortable even if we make the program easier to read and write by using *octal code* (Sec. 1-4*b*)

```
0002   0005
0003   ....
0004   1002
0005   7041
0006   3003
```

which the machine could decode quite readily from typed input. We have actually seen people program in octal code to avoid paper-tape assembly! In practice, even the simplest minicomputers have assembler programs which translate programs written in terms of mnemonic instruction codes, e.g.,

```
0002   0005
0003   ...
0004   LDA   002
0005   NEG
0006   STO   003
```

Mnemonics like LDA, NEG, and STO approximate English words; the assembler program translates mnemonics into binary code by table lookup. We have supplied addresses and address bits in octal form, just as in octal machine language.

To improve our primitive assembly language, it would be convenient if we could specify the actual 12-bit *effective address* of each memory-reference instruction, say

```
0006   STO   0003
```

Note that now the assembler must not only translate STO by table lookup, but it must also *compute* the correct address bits determined by the addressing mode (implicit in STO without extra character codes) together with the effective address. If the desired address cannot be reached by direct current-page or relative addressing, the assembler will either stop and print an error message, or (preferably) it will automatically substitute indirect or two-word addressing (see also Sec. 2-7).

4-2. Symbolic Assembly Language. Most practical assemblers are symbolic assemblers, which permit the user to refer to instruction and operand addresses in terms of symbols. In a symbolic assembly language, the sample program segment of Sec. 4-1 might look like Fig. 4-1. Each symbol (a string of up to 5 or 6 alphanumeric characters) represents a location (symbolic memory address). The word in the location-tag field (label field) of a line represents the location of the corresponding instruction or data word.

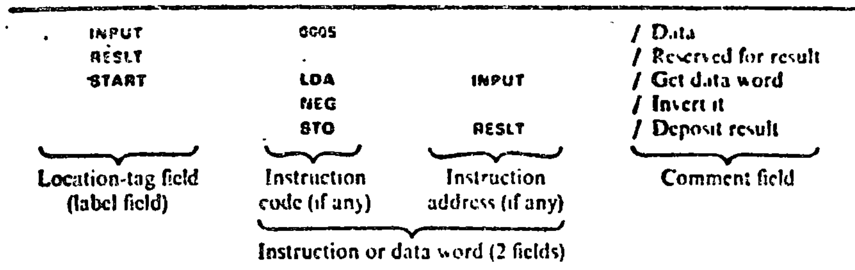


Fig. 4-1. The program segment of Sec. 4-1 written in symbolic assembly language. Note that each line (assembly-language statement) has four fields, all but one of which could be empty. When such statements are typed or punched (one to a teletypewriter line or punched card), the fields must be separated by spaces, teletypewriter tabs, or other *field delimiters* (colons, slashes, etc.) so that the assembler can recognize the end of each field.

Unless the contrary is specified, consecutive lines still represent consecutive program words. Therefore, if INPUT represents location 2, RESLT *must* represent location 3, START *must* represent location 4, and the last two instruction words *must* go into locations 5 and 6, *although we omitted their location tags in Fig.* 4-1.

The first pass of a symbolic assembler scans the user's source program and creates a symbol table which lists all user symbols defined as location tags together with their location numbers relative to a starting address. Symbolic instruction addresses as well as mnemonic instruction codes can then be translated by table-lookup operations. Taking due account of each addressing mode used, the assembler still has to compute the address bits for each memory-reference instruction. Good symbolic assemblers will automatically introduce indirect or two-word addressing when a symbolic address is not within reach of one-word paged or relative addressing (Sec. 2-7).

Multiply defined symbols will stop the assembly process and/or produce an error printout (Fig. 4-3). Some assemblers also indicate an error if a symbolic address has no location-tag counterpart (*undefined symbol*). A

PROGRAMMER			DATE	PAGE
PROGRAM				CHARGE
LOCATION 1 6	OPERATION 6 10	ADDRESS, X 12	COMMENTS 30 72	
STRT	LDA	CØNS	LØAD CØNSTANT	

Fig. 4-2. Some people like to use *coding forms* similar to the one shown for assembly-language programming. The column numbers indicated on the form are for punched cards. In teletypewriter-prepared programs, the fields are delimited by tabs, colons, slashes, etc. (*Honeywell Information Systems*.)

good assembler, though, may automatically supply each undefined symbolic address with a corresponding storage location at the end of the program so that the programmer is relieved of this task. You can see that a good symbolic assembler is a fairly complex system program.

NOTE: Precisely, A represents the assembler-determined **address** of the memory location whose **contents** (A) at run time will be the **value** of a constant *A* or values of a variable *A*.

```
                      /SAMPLE
                                              .EXT     X        ./ External reference
                      000  011  A    A1       .EQ      11        / Pseudo instruction
    000  000  R       000  100  A             100
    000  001  R       200  010  A    START    LOAD     10
    000  002  R       400  011  A             STORE    A1
    000  003  R       200  005  R             LOAD     (10)      / Literal
D   000  004  R       701  006  E    START    JUMP I   X         / Indirect jump
                      000  001  R             .END     START
    000  005  R       000  010  A  <------------------+
    000  006  R       000  006  E  <------+           |
                                          |           |
                      / 1 ERROR LINE      |           |
                                          |           |
                    / SYMBOL TABLE        |           |
                 A1        000  011  A    |           |
                 START     000  001  R    |           |
                 X         000  006  E    |           |
                                          |           |
--- Error (multiply      External symbol -+  Literal -+
    defined symbol)
```

Fig. 4-3. Listing produced by a symbolic assembler. The assembler has started at location 0 and has marked each location and address as absolute *A*, relocatable *R*, or external *E* for the linking loader (Sec. 4-19). Words which do not contain addresses are marked absolute (their *locations* may be relocatable). An *error line* has been found and marked with an error code by the assembler.

If requested by a front-panel switch setting or typed command, the assembler will print an **assembly listing** in an extra pass. The listing shows *the user's symbolic source program and the resulting octal machine code side by side* with some extra annotations (Fig. 4-3; see also Sec. 4-2). The assembler can also produce a **symbol-table printout** for reference (either in alphabetical or numerical order). You should consult your minicomputer manual for the *maximum number of symbols* and the *maximum number of program lines* which can be handled with a given computer memory.

4-3. Symbolic Expressions and Current-location References. Since correct addressing requires computations at assembly time in any case, many

symbolic assemblers improve programming convenience further by permitting symbolic expressions in address fields. For example,

```
03    ADD INTO ACCUMULATOR                     SYM          / Adds 000 173
      STORE ACCUMULATOR IN                     LEAP-HOP+2   / Address is 06
HOP   JUMP TO                                  SYM + 2      / Address is 11
      000 000
SYM   000 173
      101 201
LEAP  JUMP IF ACCUMULATOR NEGATIVE SYM-4                    / Address is 03
```

Note carefully that each expression involves **addresses** and not data. Integers will be interpreted as *octal* unless the contrary is stated (Sec. 4-5*c*). Such more elaborate assemblers usually make *two* passes through the source program, plus an optional listing pass.

Some assemblers also admit *multiplication and division* in address expressions, but these operations may not have the customary precedence, and no parentheses may be allowed. Thus, A + B * C *may* be interpreted as (A + B) * C in an address expression; *consult your assembler manual*. Some assemblers also permit bit-by-bit AND, OR, and XOR operations with symbolic-address words.

NOTE: Numerical values of symbols and expressions are necessarily fixed at assembly time. The program can only change the *contents* of symbolically addressed locations at run time.

As a further convenience, it is usually possible to **reference the location of the current instruction,** say as . , so that

```
03       LOAD ACCUMULATOR                 . + 3      / Address is 06
BOUND    JUMP IF ACCUMULATOR ZERO         . - 1      / Address is 03
         JUMP                             . + A - 1  / Address is A + 4
```

NOTE: To establish addresses like SYM - 4 or . + 3 in our examples, we have treated each source-program instruction as *one word* in memory. In general, two-word instructions (Sec. 2-7) will count as *two* locations. *Check your assembler manual on this point and on the manner of counting byte locations (if any).*

4-4. Immediate Addressing and Literals. Some minicomputers permit you to specify the *operand* (rather than the address) of a memory-reference instruction through *immediate addressing* (Sec. 2-7*e*), e.g.,

LOAD ACCUMULATOR, IMMEDIATE 010 711

where 010 711_8 is the actual number loaded, *not* an address. Similarly, LOAD ACCUMULATOR, IMMEDIATE SYMBL + 2 loads the *numerical value* of the symbolic address SYMBL + 2, *not* its contents.

For computers without true hardware immediate addressing, a symbolic assembler may implement memory-reference operations on literals like

(010 711) or (SYMBL + 2), which are defined as follows:

(010 711) is a symbolic memory location *which contains* 010 711.
(SYMBL + 2) is a symbolic memory location *which contains the numerical value of* SYMBL + 2.

The assembler automatically assembles memory locations containing each literal value at the end of the program (Fig. 4-3). It follows that

LOAD ACCUMULATOR (010 711)

actually loads 010 711. Note also that

LOAD ACCUMULATOR, INDIRECT VIA (010 711)

produces the same result in the accumulator as

LOAD ACCUMULATOR 010 711

4-5. Pseudo Instructions. (a) Introduction. The assembler can perform still more operations to improve programming convenience. To request operations to be done at assembly time, we enter pseudo instructions into the source program. To distinguish pseudo instructions from true instructions (which directly correspond to operations at *run time*), we will write a word in each pseudo instruction with a preceding dot (.). The remainder of this section will help you to interpret advertised lists of assembler features.

(b) Pseudo Instructions for Defining and Redefining Symbols. As we have seen, one can *define* a symbol (i.e., give it a numerical value) by using it as a location tag (label). Another way to define a symbol is through the pseudo instruction

.DEFINE ADDRESS SYMBL

which assigns SYMBL the value of the current location, just like

SYMBL ... ...

As it stands, either statement leaves the *contents* of SYMBL unspecified. *With computers permitting repeated indirect addressing and/or post-indexing, however, the* .DEFINE ADDRESS *pseudo instruction can be used with indirect or indexed addressing to set the indirect or index bits of the specified location.* For example,

```
.DEFINE ADDRESS, INDEXED        SYMBL
 STORE ACCUMULATOR, INDIRECT    SYMBL
```

would produce an effective storage address equal to the sum of the contents of SYMBL and the contents of the index register.

The pseudo instruction (assignment statement)

SYMBL1 .EQ SYMBL2 + SYMBL3 − 1

assigns the value of the expression on the right to SYMBL1 in the *following* program statements. Such assignment can be used to define or redefine a symbol before or after it is used as a label or address. Note, however, that with the usual two-pass assembler

```
SYMBL1    .EQ    7
SYMBL2    .EQ    SYMBL1-2
```

is legal, but

```
SYMBL2    .EQ    SYMBL1-2
SYMBL1    .EQ    7
```

will cause an error message ("UNDEFINED SYMBOL") unless SYMBL1 was defined (as a label, by a .DEFINE ADDRESS statement, or by another assignment statement) earlier in the program.

(c) Pseudo Instructions Defining Data Types. Most minicomputer assemblers normally interpret integers in source-program addresses, expressions, or data as single-precision octal integers, so a statement like

ALFA 017002

reserves *one* memory location. Some assemblers can define double-precision quantities (still in octal code) by pseudo instructions like .DOUBLE, so

BETA .DOUBLE 7173514

generates *two* words (in locations BETA and BETA + 1).

The pseudo instruction .DECIMAL permits you to enter decimal integer constants in your source program; thus

GAMMA .DECIMAL 1982

will generate the correct one-word binary number.

Some assemblers will correctly assemble binary floating-point numbers when .DECIMAL is followed by a real number containing a *decimal point*. (Either 10.73 or 0.1073 E + 02 will work.) Other assemblers require a separate pseudo instruction, such as .FLOAT. It is similarly possible to declare double-precision floating-point data; the assembler will correctly assign *three or more* words for each data entry.

Some assemblers also accept hexadecimal integers following the pseudo instruction .HEX.

The pseudo instruction .ASCII followed by alphanumeric text (usually delimited by quotation marks) causes ASCII characters to be packed into successive computer words, where they can be accessed, for example, by output routines for printing error messages, for example,

.ASCII 'BOOBOO IN LINE 12'

NOTE: In some assemblers, a pseudo instruction like .DECIMAL remains valid for subsequent words until it is revoked by another data-defining pseudo instruction, such as .OCTAL.

(d) Pseudo Instructions for Reserving Storage Blocks. A pseudo instruction like

```
SYMBL    .BLOCK    N
```

where *N* is a positive integer, reserves *N* storage locations, starting with the location SYMBL, for data or instruction words. N can be a symbol or, in fact, an expression; the block size is, in any case, given its numerical value at assembly time. Some assemblers automatically reset all reserved locations to 0 if their contents are not specified. Some assemblers can also reserve blocks *ending* at a specified location.

NOTE: Locations reserved for noninstruction words *must be situated at the beginning of a program, at its end, or immediately following an unconditional-jump instruction.* Otherwise, the machine might execute a noninstruction as the program counter advances, with regrettable results!

(e) Pseudo Instructions for Controlling the Assembly Process. The pseudo instruction

```
.ORIGIN    (address)
```

causes the subsequent program to start (or continue) from the specified address (which can be relocatable, Sec. 4-18).

Every assembly-language source program *must* terminate with the pseudo instruction

```
.END    (starting address of program)
```

to tell the assembler that no more program statements follow. The assembler may then add extra words for literals and for previously undefined symbolic addresses if it has these features. The *starting address* is the address of the first *instruction* to be executed and is usually appended to the .END pseudo instruction, so that a loader program (Sec. 4-19) can insert a jump to the starting address and a HALT for convenient restarting. With simpler operating systems, absolute starting addresses are set up with front-panel switches (Sec. 3-4).

NOTE: The *pseudo instruction* .END signifies the end of *assembly*. *Program* execution may end with the *instruction* HALT or with a jump to an executive program.

(f) Other Pseudo Instructions. Additional types of pseudo instructions are used to *link programs* (Secs. 4-17 and 4-19) and to define *macros* (Sec. 4-21) and *conditional assembly* (Sec. 4-23). Some assemblers also have pseudo instructions *to control or format listings*, but it is probably more convenient to do this with front-panel switches or, preferably, with keyboard-executive commands (Sec. 3-11).

4-6. The .REPEAT Pseudo Instruction. The pseudo instruction .REPEAT is a programwri convenience. The statement

```
.REPEAT    m,n
```


where the count *m* is a positive integer and the increment *n* is a signed integer (positive, negative, or zero), causes the immediately following program word (instruction or data word) to be repeated *m* times with 0, *n*, 2*n*, . . . , (*m* − 1)*n* added to successive words. For example,

```
.REPEAT    3 2
 0001
.REPEAT    2,-1
 0002
```

generates

```
0001
0003
0005
0002
0001
```

Note that addresses as well as data can be incremented. Some assemblers have more elaborate .REPEAT pseudo operations capable of repeating *groups* of words.

INTRODUCTION TO PROGRAMMING

4-7. Program Documentation: Use of Comments. Unless you intersperse your program statements with plenty of explanatory comments, not even you yourself (and surely no one else) will be able to understand your program one month later. This is true for FORTRAN programs and any other program as well as for assembly-language programs.

Comments are not restricted to the comments fields of assembly-language statements; the assembler will recognize any line preceded by / (or similar delimiters such as ; , *, etc.) as a comment line, say

```
/ THIS IS A COMMENT LINE
```

Such comment lines can also be used for *program titles*. Comments will not cause any program words to be assembled, but comments will be reproduced in the assembler listing for future reference.

4-8. Branching and Flow Charts. Many minicomputers do not have conditional-jump instructions but combine unconditional jumps with conditional skips (which fit better into short instruction words):

```
/ THE FOLLOWING COMPARISON OF THE CONTENTS OF
/ LOCATIONS A AND B IS AN EXAMPLE OF A
/ THREE-WAY DECISION USING CONDITIONAL SKIPS
TEST   LOAD ACCUMULATOR                A
       SUBTRACT INTO ACCUMULATOR       B / A - B in accumulat
       SKIP IF ACCUMULATOR POSITIVE      / A > B?
       SKIP                              / No, test for A = B
       JUMP TO                       POS / Yes, branch to POS
       SKIP IF ACCUMULATOR  ZERO         / A = B?
       JUMP TO                       NEG / No, branch to NEG
ZERO   (program continues)               / Yes, go on
```

In general, specific operation-code bits of conditional-skip instructions correspond to conditions such as <, >, =, carry, and overflow. Such conditions can then be ORed together to form *combined conditions* such as ≤ (see also Secs. 2-11, 6-1, and 6-5).

Figure 4-4 is a flow chart for our program-branching example. Such flow charts are helpful when there are many complicated decisions and

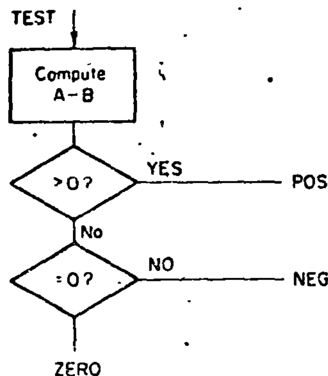


Fig. 4-4. Flow chart for the example of Sec. 4-8

especially when there are program loops (Sec. 4-9 and Fig. 4-5). A flow chart is a topological model of the actual paths traced through the computer memory as the program executes instructions along different branches. The source program itself, on the other hand, is a *one-dimensional* rendering of each path in turn, together with listings of memory locations reserved for data and addresses (these do *not* appear on flow charts). It can be helpful to supplement your flow chart with a memory map listing data-storage locations.

4-9. Simple Arrays, Loops, and Iteration. A one-dimensional array of, say, 1,000 variables $A1, A2, \ldots, A1000$ will be stored in the computer memory as an example of a **data structure** arranged to simplify access to the data during common operations with this type of data. For our one-dimensional array, we simply reserve 1,000 consecutive memory locations with

```
        .DECIMAL
A1      .BLOCK      1000
```

or (in octal code)

```
N  .EQ 1750          / Permits N to
A1    .BLOCK    N    /   be changed at assembly
```

(Sec. 4-5). You should always check carefully whether the starting value of the array index I in AI is $I = 1$ or $I = 0$. This is a frequent source of errors.

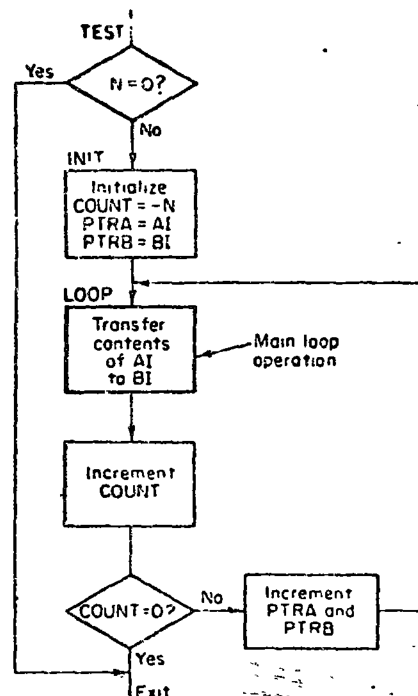


Fig. 4-5a. Flow chart for a simple program loop (Sec. 4-9)

```
/THIS ROUTINE MOVES THE CONTENTS OF AI TO BI FOR I = 1 TO I - N, OR SKIPS
/ENTIRE LOOP IF N   0
TEST    LOAD ACCUMULATOR                 NN
        SKIP IF ACCUMULATOR POSITIVE            /  N = 0?
        JUMP TO                          NN + 1 /  Yes, exit
INIT    INVERT ACCUMULATOR                      /  No, initialize loop
        STORE ACCUMULATOR IN             COUNT  /  Contains −N
        LOAD ACCUMULATOR                 (A1)   /  Note literal!
        STORE ACCUMULATOR IN             PTRA   /  Points to A1
        LOAD ACCUMULATOR                 (B1)
        STORE ACCUMULATOR IN             PTRB   /  Points to B1
LOOP    LOAD ACCUMULATOR, INDIRECT VIA   PTRA   / }This is the actual
        STORE ACCUMULATOR, INDIRECT VIA  PTRB   / }  loop operation
        INCREMENT, SKIP IF ZERO          COUNT  /  N operations done?
        SKIP                                    /  No, repeat loop
        JUMP TO                          NN + 1 /  Yes, exit
        INCREMENT, SKIP IF ZERO          PTRA   /  Points to next AI
        INCREMENT, SKIP IF ZERO          PTRB   /  Points to next BI
        JUMP TO                          LOOP
A1      .BLOCK                           N      /  Good place to store
B1      .BLOCK                           N      /    data, following
                                                /    unconditional jump!
PTRA    ...  ...
PTRB    ...  ...
NN      ...  ...                                /  Contains N ≥ 0
```

Fig. 4-5b. A simple loop programmed *without* an index register.

```
/ THIS ROUTINE MOVES THE CONTENTS OF AI TO BI FOR I = 1 TO I ≤ N, USING INDEX
/ REGISTER, OR SKIPS ENTIRE LOOP IF N = 0
TEST   LOAD INDEX REGISTER                 NN
       SKIP IF INDEX POSITIVE                       / N = 0?
       JUMP TO                             NN + 1   / Yes, exit
LOOP   LOAD ACCUMULATOR, INDEXED           A1 - 1   / Loads AN first
       STORE ACCUMULATOR, INDEXED          B1 - 1   / Stores BN first
       DECREMENT INDEX, SKIP IF ZERO                / N operations done?
       JUMP TO                             LOOP     / No, repeat loop
       JUMP TO                             NN + 1   / Yes, exit
A1     .BLOCK                              N        / Good place to
B1     .BLOCK                              N        /    store data!
NN     .. ...                                       / Contains N ≥ 0
```

Fig. 4-5c. The same simple loop programmed *with* an index register.

Programs for typical array operations, e.g., moving the contents of AI to BI, or

$$CI = AI + BI \qquad I = 1, 2, \ldots, N$$

$$S = \sum_{I=1}^{N} (AI)(BI)$$

require execution of a number of instructions proportional to N. Since memory capacity is limited, it is not just convenient but quite necessary to use program loops, which repeat the same instructions with successively incremented addresses AI, BI, and/or CI; a *counting operation* will be set up to advise us when the loop has run N times (Fig. 4-5).

Figure 4-5*a* shows how a simple loop can be programmed for a primitive minicomputer *without index registers*. Some minicomputers (PDP-8 series) would simplify the incrementation of PTRA and PTRB by *autoindexing* (Sec. 2-7*c*); the ISZ instruction would still be needed to increment COUNT since it is necessary to sense when N operations have been completed. But *by far more efficient loop operations are possible with an index register.* Figure 4-5*b* shows how *a single index register is used to step two data addresses as well as the loop count.* Many minicomputers, though, will require separate instructions for stepping an index register and testing it for 0.

We have stepped the loop index *after* each actual loop operation. We could do this *before* the loop operation instead. Note also:

1. The loop index (or COUNT, PTRA, and PTRB in Fig. 4-5*a*) must be *initialized* before the actual loop processing begins. While assembly-language statements like

   COUNT 000 000

   would initialize the loop before it runs *for the first time* after assembly, subsequent runs would *not* be initialized!

2. Since a loop may be traversed many times, *it is uneconomical to include unnecessary operations in the loop.* For instance, in the computation of

   $$\sum_{i=1}^{n} ah_i = a \sum_{i=1}^{n} h_i$$

   the multiplication by a is common to all terms and should *not* be included in the loop. The same is, of course, true in FORTRAN or BASIC programming.

An array may well contain two-word or multiword items, such as multiple-precision or floating-point data. In such situations, index-register incrementing becomes only a little more complicated. To access, say, every fourth word of an array *without index registers*, however, is a more cumbersome (but still straightforward) operation.

Every loop must contain a test to branch out of the loop when a desired condition is met. In our simple example, this condition was the completion of exactly N elementary operations, but a loop could be determined before N operations, e.g., when a sum exceeds a specified value or when an error becomes small enough.

In fact, *the loop technique is in no way restricted to operations* with elements of stored arrays; array elements could be *generated* by the loop. This is the case for *iterative-approximation operations*.

4-10. More Data Structures. (a) Two-dimensional Arrays. **Two-dimensional arrays, like**

A11	A12	...	A1N
A21	A22	...	A2N
...	...	...	...
AM1	AM2	...	AMN

($M \times N$ array), are usually stored in the computer memory as one-dimensional arrays, say *by rows*, as

A1, A2, . . . , A(MN)

where the single subscript J in AJ is related to the subscripts I and K of AIK by

$$J = (K - 1)N + I \qquad I = 1, 2, \ldots, N; K = 1, 2, \ldots, M \qquad (4\text{-}1)$$

To access the location AIK of the array element AIK, the computer will have to add $J - 1$ to the address A11 (starting address), i.e.,

$$\text{AIK} = \text{A11} + (K - 1)N + I - 1$$
$$I = 1, 2, \ldots, N; K = 1, 2, \ldots M \qquad (4\text{-}2)$$

Larger digital computers permit computation of such addresses by *double indexing* (adding contents of two index registers), but this is *not* possible with most minicomputers even if two index registers are available. Accessing of the individual array elements AIK (as in matrix computations) will, therefore, be somewhat cumbersome unless *postindexing* (Sec. 2-7) is available (as in the Honeywell 316/516 and the Varian Data Systems 620/f).

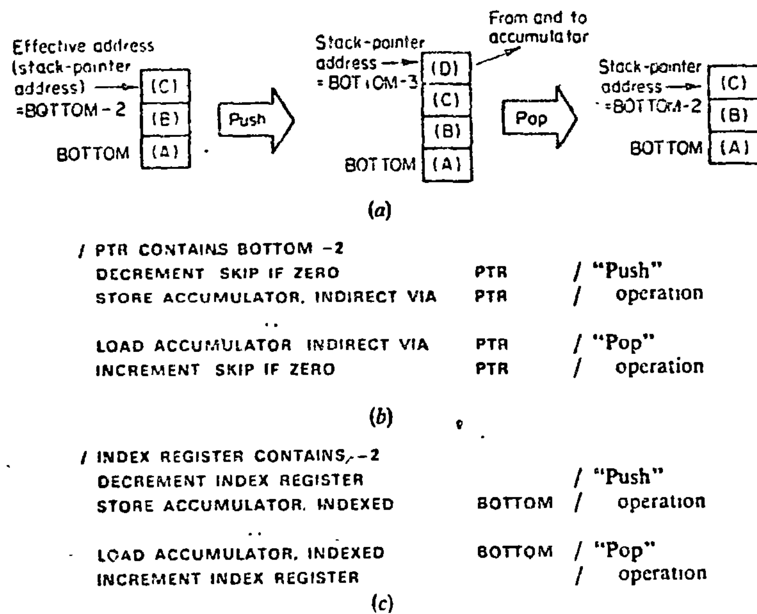


Fig. 4-6. Push and pop operations in a simple (one word per item) pushdown stack: memory map (*a*) and programming without an index register (*b*) and with an index register (*c*). Addresses *increase* toward the bottom of the stack in this example; the contrary could be true.

In that case, $(K - 1)N$ can be generated in the index register by successive additions of N, while $A11 + I - 1$ will appear in an indirectly addressed memory location which is incremented to advance I.

NOTE: As with one-dimensional arrays, you should make sure that row and column subscripts of given arrays really start with 1 and not with 0.

(b) Stacks (Pushdown Lists). A practically important class of data structures are stacks, i.e., arrays permitting words or subarrays (items) to be adjoined, removed, or accessed from the top of the stack *on a last-in–first-out basis*. Such stacks are also known as pushdown lists or LIFO (last-in–first-out) lists. Stacks are especially useful for orderly intermediate-result storage and for various systems-programming applications (Fig. 4-6; see also Secs. 4-16, 6-8, and 6-10).

(c) Other Data Structures (see Refs. 1 to 4). Structures of multiple (and possibly variable-length) subarrays which can be *created* and *deleted* in the course of computation are often organized as various types of (linked) lists, rather than as multidimensional arrays, which might waste permanently assigned storage space. A (linked) list or chain is an ordered set of word arrays (items), *each comprising a pointer to the next item in the list or to a directory array of item starting addresses*. Individual item arrays can be located *wherever memory space is available*. One usually keeps a separate list of available space; an item is deleted from this available-space list whenever an item is added to another list, and vice versa.

List structures are used to store and access program lines (character strings) in editing programs, catalog and inventory items, bibliographical references, graphic-display items (Sec. 7-11), and rows or columns of sparse matrices (i.e., matrices with many 0 elements—this would make simple two-dimensional-array storage uneconomical). List items can also contain *backward pointers* to preceding items, pointers to subitems, and/or counters indicating sizes of item arrays. Reference 4 is a good introduction to your study of list processing, which has opened up many interesting new programming techniques.

4-11. Miscellaneous Programming Techniques. (a) Table-lookup Operations. Section 4-8 illustrates a *triple* branch implemented with conditional skip-jump instructions. When a decision has more than a few possible outcomes, though, it may be best to store the jump-destination addresses in an array (table) addressed in the manner of Sec. 4-9. The result of each decision will correspond to the value of an array index I placed in an index register or address pointer to access an address in the array. If the decision depends on more than one factor, we can use a multidimensional-table array with an index computation like the one in Sec. 4-10*a*.

Such table-lookup operations are, of course, precisely those needed to look up values of tabulated numerical functions. To reduce the size of the function table needed to compute a continuously differentiable function with suitable accuracy, we can *combine table lookup and interpolation*.

Figure 4-7 illustrates a high-speed method for fixed-point table-lookup/interpolation approximation of a function $Y = F(X)$ in the form

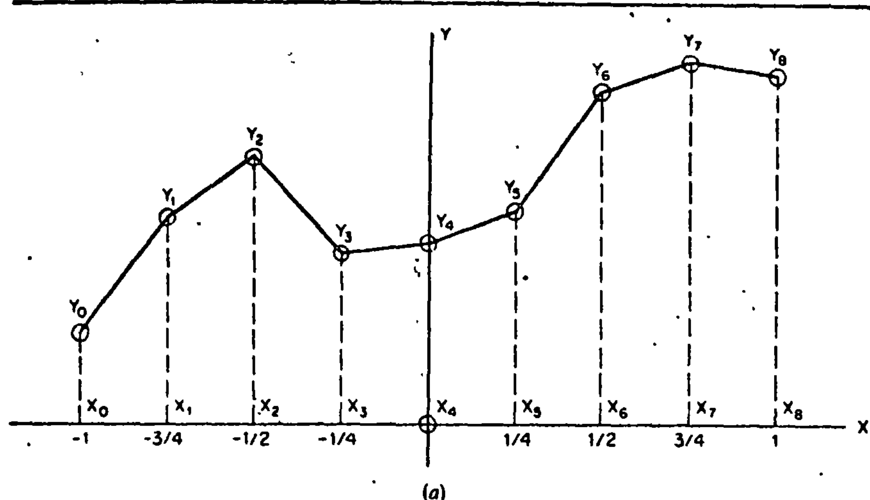
$$Y \approx (Y_{i+1} - Y_i)\frac{X - X_i}{X_{i+1} - X_i} + Y_i \qquad (4\text{-}3)$$

where scaled function values $Y_i = F(X_i)$ are tabulated for $2^N + 1$ uniformly spaced breakpoint abscissas

$$X_i = 2^{1-N}i - 1 \qquad i = 0, 1, 2, \ldots, 2^N \qquad (4\text{-}4)$$

between $X_0 = -1$ and $X_{2^N} = 1$ (Fig. 4-7*a*). The program of Fig. 4-7*b* begins with the n-bit 2s-complement fraction

$$X = X_i + (X - X_i)$$



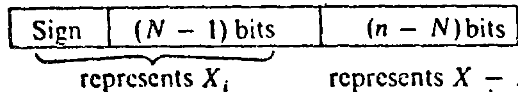
(a)

```
LOAD ACCUMULATOR 2                          X
ADD INTO ACCUMULATOR 2                      (100 000)  / Complement sign
                                                       /   (for 16 bits)
CLEAR ACCUMULATOR 1 AND CARRY FLAG
SHIFT ACCUMULATORS 1 AND 2 LEFT N BITS
MOVE ACCUMULATOR 1 TO INDEX REGISTER                   / This is i
SHIFT ACCUMULATOR 2 RIGHT                              / This produces
STORE ACCUMULATOR 2 IN                      TEMP1      /   2^(Y-1)(X - X_i)
                                                       / = (X - X_i)/(X_(i+1) - X_i)
CLEAR ACCUMULATOR 1
SUBTRACT INTO ACCUMULATOR 1, INDEXED        YO         / Produces -Y_i
STORE ACCUMULATOR 1 IN                      TEMP2      /   and stores it
INCREMENT INDEX REGISTER
ADD INTO ACCUMULATOR 1 INDEXED              YO         / Produces Y_(i+1) - Y_i
MULTIPLY ACCUMULATOR 1 BY                   TEMP1
SHIFT ACCUMULATORS 1 AND 2 LEFT 1 BIT                  / Because of fractions
SUBTRACT INTO ACCUMULATOR 1                 TEMP2
STORE ACCUMULATOR 1 IN                      Y
/ YO IS FUNCTION-TABLE ORIGIN
/ ACCUMULATOR 2 IS LESS-SIGNIFICANT ACCUMULATOR
```

(b)

Fig. 4-7. Table-lookup/interpolation approximation of a function $Y = F(X)$ with 2^N equal breakpoint intervals (Sec 4-11*b*)

in the form



Note that the right-hand $(n - N)$ bits represent the nonnegative difference $X -$ ¨ needed for interpolation; *we shift them into a second accumulator*

(accumulator extension, MQ register, Sec. 2-10) and save them for multiplication. The sign bit of X_i is complemented to form the breakpoint index i, which is added to the table origin YO (location of Y_0) to produce a pointer to Y_i. The entire operation requires under 40 memory cycles (typically less than 40 μsec) and can be generalized for nonuniform breakpoint spacing and functions of two or more variables (Ref. 10).

(b) Program Switches. A program switch stores the result of a binary or multiple branching decision to implement the actual branching *later on* in another part or parts of the program. An example is precomputation and storage of decisions for use inside loops (Ref. 3) to free the latter of repeated decision making. The decision result can be stored in a memory location, in a processor flag (if it is not otherwise in use), or, if possible, in an index register.

EXAMPLE:

```
LOAD ACCUMULATOR                          A1
ADD INTO ACCUMULATOR                      A2
ADD INTO ACCUMULATOR                      A3
SUBTRACT                                  B
CLEAR INDEX REGISTER
SKIP IF ACCUMULATOR NOT POSITIVE
INCREMENT INDEX REGISTER                        / Positive
SKIP IF ACCUMULATOR NOT ZERO
INCREMENT INDEX REGISTER                        / Positive or zero
```

The index register now reads 0, 1, or 2 if A1 + A2 + A3 - B was negative, zero, or positive respectively. The desired three-way branch can be obtained now or later with

JUMP, INDEXED, INDIRECT VIA PTR

The program will jump via PTR, PTR + 1, or PTR + 2

(c) Miscellaneous Examples. The program segments of Fig. 4-8 illustrate useful programming techniques possible with typical minicomputer instruction sets (see also Chap. 6).

```
LOAD ACCUMULATOR                A
SHIFT LEFT, UNSIGNED
STORE ACCUMULATOR IN            TEMP
LOAD ACCUMULATOR                B
SHIFT LEFT UNSIGNED
CLEAR CARRY FLAG
ADD INTO ACCUMULATOR            TEMP
LOAD ACCUMULATOR                A
ADD INTO ACCUMULATOR            B
SKIP ON CARRY FLAG CLEAR
JUMP TO                         OFLO     / Overflow-error routine
STORE ACCUMULATOR IN            C
```

Fig. 4-8*a*. Overflow check for 2s-complement addition ($A + B = C$) on a machine having carry flag but no true overflow flag. Carries from the most significant bit and from the sign bit are both allowed to complement the carry flag in turn, so that they are effectively XORed (see also Secs. 1-9*a* and 2-10*a*).

```
CLEAR CARRY FLAG
LOAD ACCUMULATOR          A2
ADD INTO ACCUMULATOR      B2
STORE ACCUMULATOR IN      A2
LOAD ACCUMULATOR          A1
SKIP IF NO CARRY                / Used instead of
INCREMENT ACCUMULATOR           /   ADD CARRY instruction
ADD INTO ACCUMULATOR      B1
STORE ACCUMULATOR IN      A1
```

Fig. 4-8*b*. Double-precision addition on a minicomputer without DOUBLE ADD or ADD CARRY instructions. A double-precision number is added from B1, B2 into A1, A2. A1 and B1 hold signs and most significant bits. No overflow check is included

```
/ ONE-ACCUMULATOR MACHINE           / TWO-ACCUMULATOR MACHINE
/ NEEDS 12 CYCLES                   / NEEDS 8 CYCLES
  LOAD ACCUMULATOR        A           LOAD ACCUMULATOR 1      A
  STORE ACCUMULATOR IN    TEMP        LOAD ACCUMULATOR 2      B
  LOAD ACCUMULATOR        B           STORE ACCUMULATOR 1 IN  B
  STORE ACCUMULATOR IN    A           STORE ACCUMULATOR 2 IN  A
  LOAD ACCUMULATOR        TEMP
  STORE ACCUMULATOR IN    B
```

Fig. 4-8*c*. Multiple accumulators can often save time-consuming memory references by serving as quickly accessible temporary-storage locations. As an example, the Data General NOVA/SUPERNOVA manual compares routines for interchanging the contents of two memory locations A, B (e g , in sorting operations)

```
LOAD ACCUMULATOR, IMMEDIATE   777000  / Load mask
AND INTO ACCUMULATOR          Y       / Mask 9 low-order bits
STORE ACCUMULATOR IN          TEMP    / Save result
LOAD ACCUMULATOR              X
SHIFT RIGHT 9 BITS, UNSIGNED          / Shift right
ADD INTO ACCUMULATOR          TEMP    / Combine with Y

         (Store in array, or output and display packed word)
```

Fig. 4-8*d*. This routine truncates two 18-bit numbers X, Y to 9 bits and packs the truncated words into one 18-bit word for a cathode-ray-tube display (Sec. 7-9). Y is truncated by masking, and X is truncated by shifting.

SUBROUTINES AND CALLING SEQUENCES

4-12. Introduction: Subroutines without Direct Data Transfer. In many applications, a reasonably involved program section is used over and over again in the course of a computation. We may then save a great deal of memory if we store such a subroutine only once, jump to its tagged starting location whenever the subroutine is needed, and make a return jump to the calling program when the subroutine is finished. Besides saving memory, the use of subroutines can give our programs a more easily understood "modular" structure, but subroutines will *not* save time compared to straight-line programming. They will (at the least) add extra jump instructions as "overhead" and can provide excellent chances for making programming errors, especially when subroutines must call one another.

The simplest subroutines do *not* process data passed to them directly by the various subroutine-calling sections of the main program. A good example is a subroutine which, at several points of a data-processing program, transfers the words of the same buffer area in memory to a line printer, perhaps doing some reformatting and checking on the way. This can be a rather long subroutine (100 or more instructions, see also Sec. 5-27) It will be a real relief to store it only once in memory and to have to write it only once in our program. Calling this particular subroutine is simple, for the calling program need not tell the subroutine what data to process: the subroutine always operates on the same buffer.

When we jump to the subroutine, we must *save the return address* for our later return to the calling program. Most minicomputers do this with the instruction

JUMP AND SAVE (effective address)

which will store the correct return address (incremented program-counter contents) at the effective address, say SUBR, which precedes that of the first subroutine instruction.

After our subroutine is finished, an indirect jump via the location SUBR (where the return address is stored) will return us to the calling program (Fig 4-9).

The instruction JUMP AND SAVE can also be used with indirect addressing.

NOTE: Contents of processor registers (accumulators, index registers, processor flags, page register, interrupt mask) needed later by the calling program may have to be saved in memory before we call a subroutine which uses these registers In some computers, the JUMP AND SAVE instruction automatically saves processor flags and the page register in an extra location following the return address.

4-13. Argument and Result Transfer through Processor Registers. Many subroutines will process arguments (parameters) passed to them by each program section which calls the subroutine. Arguments can be data words

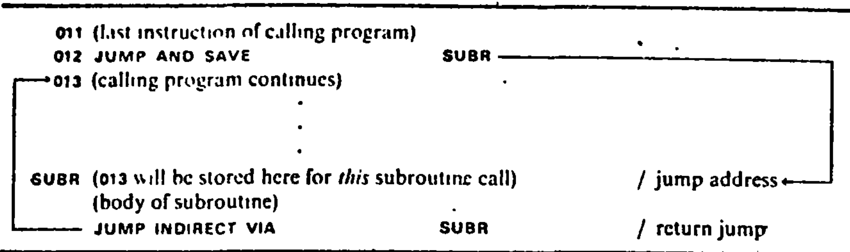


Fig. 4-9. Simple subroutine call and return. No special provisions are made to transfer arguments or results

but may also be symbolic addresses. Subroutines will also have to return results to calling programs. Quite often, only one argument and/or result or only a few arguments and/or results must be passed, as in a function-generating subroutine (e.g., square root, table-lookup function). Note that while the code for the subroutine remains the same for each call, argument(s) and/or result(s) will differ. A simple way to pass one data word is to place it into an accumulator or index register during the subroutine jump or return jump; several words can be passed if several registers are available.

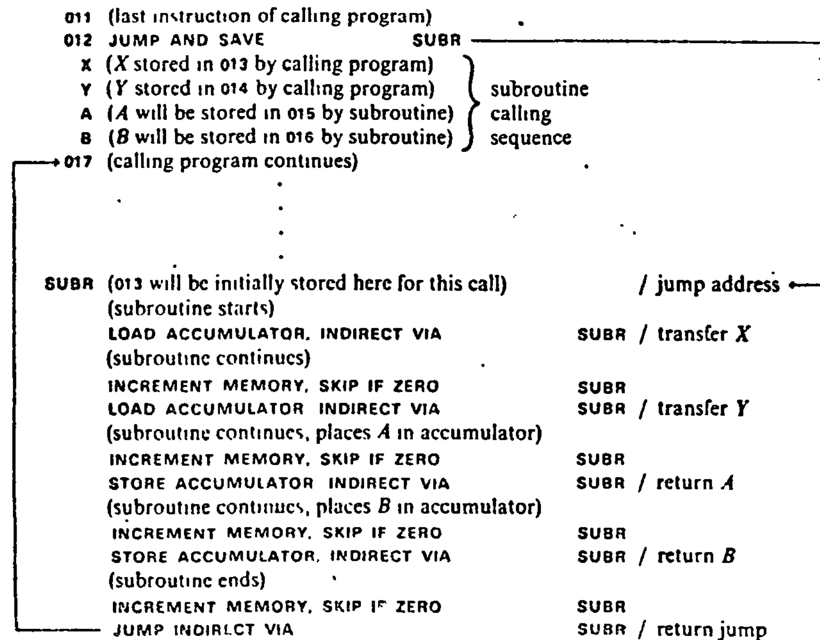


Fig. 4-10*a*. A calling-sequence method for passing two arguments *X*, *Y* and returning two results *A*, *B*. A computer permitting autoincrementing or postindexing of the indirect-address pointer SUBR would make this program simpler and faster

4-14. Argument and Result Transfer through a Calling Sequence. Use of an Index Register. Quite often, we must pass more subroutine arguments and/or results than we have processor registers, or our registers are otherwise occupied. In such cases, we can employ a **subroutine calling sequence.** In Fig. 4-10*a*, the calling program reserves locations for the arguments, say X, Y, and for the results, say A, B, immediately following the subroutine jump. The subroutine can then access X, Y, A, and B in turn by indirect addressing and successive incrementation of the jump address SUBR. The last incrementation produces the correct return address.

A few minicomputers can avoid the repeated ISZ instructions in Fig. 4-10*a* by **postindexing the indirect address or by supplying an autoincrement addressing mode,** which increments indirectly addressed memory locations when a special operation-code bit is set (Secs. 2-7 and 6-7*c*). Other mini**computers use an index register to store the return address** (or the first calling-sequence address) through the instruction JUMP AND SAVE IN INDEX (Secs. 2-11*b* and 6-10). In this case, we can access the subroutine arguments and transfer results more rapidly through indexed addressing. It also becomes much easier to deal with arguments and results *in random order* rather than strictly consecutively (Fig. 4-10*b*).

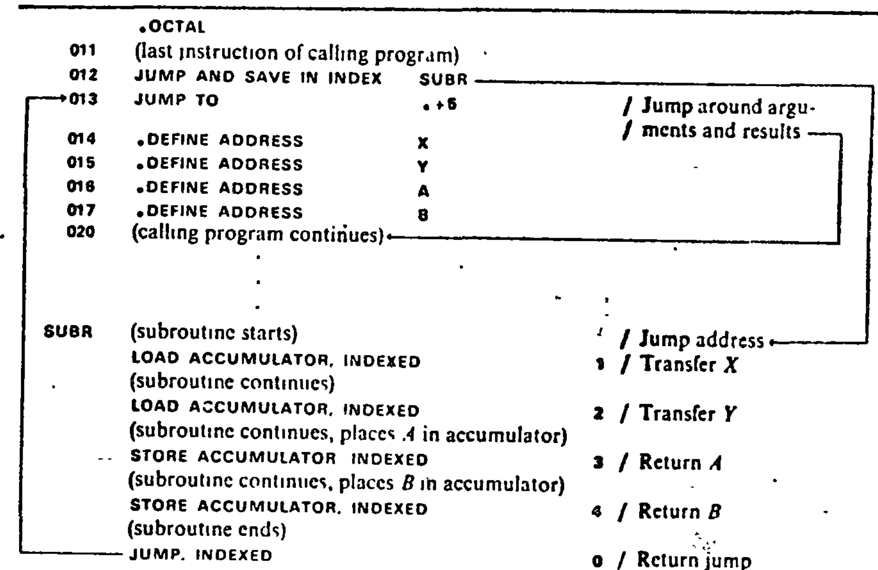
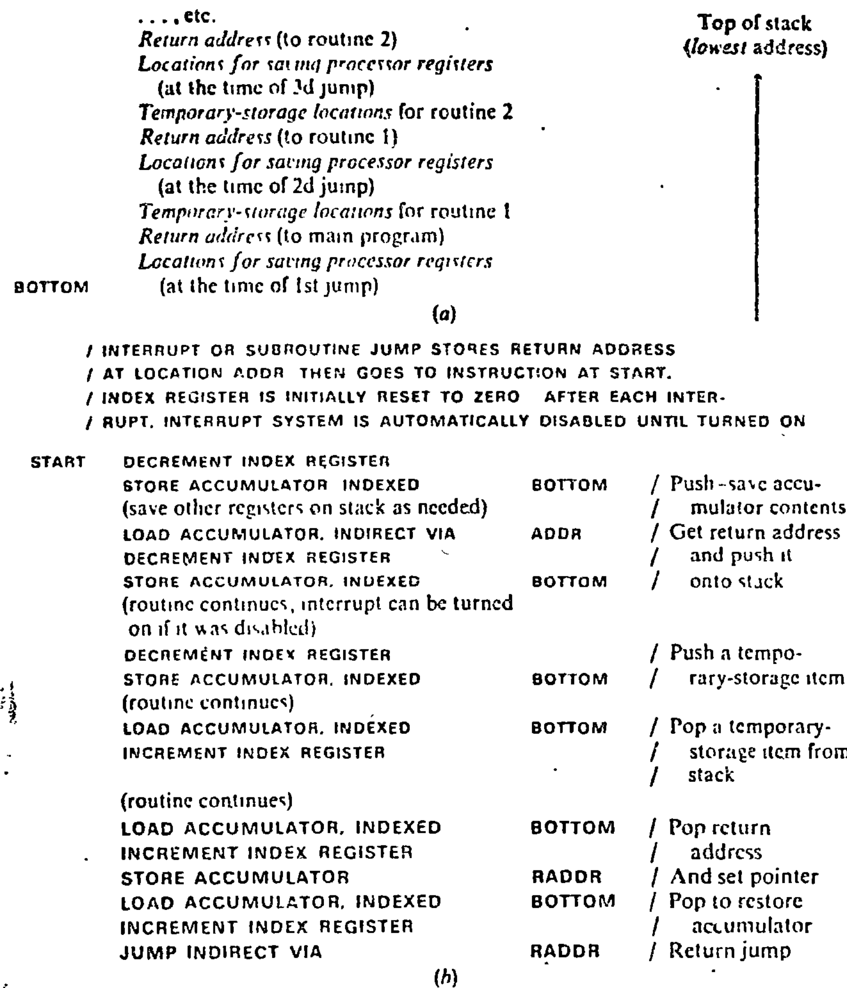


Fig. 4-10*b*. Subroutine and calling sequence employing the JUMP AND SAVE IN INDEX instruction. Note that arguments and results could be just as easily accessed in any other order. The return jump was made to location 013 (immediately following the subroutine jump) with an extra jump around the calling-sequence items. This is a convention expected of subroutines called by system programs in some computer systems. Otherwise, a return jump through JUMP, INDEXED 5 would be simpler and faster.



```
        . . . , etc.                                              Top of stack
        Return address (to routine 2)                             (lowest address)
        Locations for saving processor registers
          (at the time of 3d jump)
        Temporary-storage locations for routine 2
        Return address (to routine 1)
        Locations for saving processor registers
          (at the time of 2d jump)
        Temporary-storage locations for routine 1
        Return address (to main program)
        Locations for saving processor registers
BOTTOM    (at the time of 1st jump)
                                  (a)

    / INTERRUPT OR SUBROUTINE JUMP STORES RETURN ADDRESS
    / AT LOCATION ADDR THEN GOES TO INSTRUCTION AT START.
    / INDEX REGISTER IS INITIALLY RESET TO ZERO  AFTER EACH INTER-
    / RUPT, INTERRUPT SYSTEM IS AUTOMATICALLY DISABLED UNTIL TURNED ON

START   DECREMENT INDEX REGISTER
        STORE ACCUMULATOR  INDEXED         BOTTOM   / Push-save accu-
        (save other registers on stack as needed)   /   mulator contents
        LOAD ACCUMULATOR, INDIRECT VIA     ADDR     / Get return address
        DECREMENT INDEX REGISTER                    /   and push it
        STORE ACCUMULATOR, INDEXED         BOTTOM   /   onto stack
        (routine continues, interrupt can be turned
         on if it was disabled)
        DECREMENT INDEX REGISTER                    / Push a tempo-
        STORE ACCUMULATOR, INDEXED         BOTTOM   /   rary-storage item
        (routine continues)
        LOAD ACCUMULATOR, INDEXED          BOTTOM   / Pop a temporary-
        INCREMENT INDEX REGISTER                    /   storage item from
                                                    /   stack
        (routine continues)
        LOAD ACCUMULATOR, INDEXED          BOTTOM   / Pop return
        INCREMENT INDEX REGISTER                    /   address
        STORE ACCUMULATOR                  RADDR    / And set pointer
        LOAD ACCUMULATOR, INDEXED          BOTTOM   / Pop to restore
        INCREMENT INDEX REGISTER                    /   accumulator
        JUMP INDIRECT VIA                  RADDR    / Return jump
                                  (b)
```

Fig. 4-11. A pushdown stack for saving return addresses, processor-register contents, and temporary-storage items for reentrant nested subroutines and/or interrupt-service routines (*a*) and typical programming (*b*). An index register is used to produce the effective stack-pointer address, but an indirect-address pointer can be used if no index register is free (see also Fig. 4-6). A few minicomputers have special *stack-pointer registers*, which are automatically decremented and incremented when an interrupt-service routine starts or is completed (see also Sec. 6-10). Note that some register contents and/or temporary-storage items may not get stored before the next interrupt occurs.

4-15. Subroutines Calling Other Subroutines. If, as is frequently the case, a subroutine calls another subroutine (nesting of subroutines), then the following points may require attention:

1. If the computer stores subroutine return addresses and/or computer status words in special registers (e.g., an index register) or in fixed memory locations, then their contents must be saved prior to the subroutine call. It will be necessary to establish an orderly procedure for saving and restoring these items with each new subroutine call.
2. A special problem arises if a subroutine calls itself (*recursive* subroutine call, Ref. 3).

4-16. Interrupt-service Routines and Reentrant Subroutines. An interrupt-service routine is a subroutine called into action by a signal (interrupt request) from outside the computer or because of an alarm condition in the computer (power-supply failure, violation of memory protection), rather than by a computer program call. Interrupt-system hardware and programming will be discussed in some detail in Chap. 5, but it will be useful to see right here how interrupt-service programming differs from ordinary subroutine programming.

The essential point is this: We *know* where in our program a subroutine will be called, and we can prepare data, save registers, etc., beforehand. But we do *not* in general know where an interrupt request will cause our program to jump to an interrupt-service routine (at best, we can *suppress* interrupts during critical program phases, Chap. 5). Hence interrupt-service routines cannot employ calling sequences, and they must do any saving and restoring of return addresses, register contents, and status words themselves without any help from the main program.

A special program arises when a subroutine (say a library routine for computing the square root) is interrupted, and the interrupt-service program calls *the same* subroutine. The original subroutine call may cause intermediate-result storage in temporary-storage locations, say TEMP1 and TEMP2. Unless special precautions are taken, intermediate results from the second subroutine call can *overwrite* TEMP1 and TEMP2 so that the program will fail upon return from interrupt. *The library subroutines of most FORTRAN systems fail in this manner.*

Subroutines designed to work properly when they are interrupted and recalled for interrupt service are called reentrant. Since "real-time" computations involving many interrupt-driven program segments are important minicomputer applications, reentrant programming is often desirable. A good way to obtain reentrant subroutines, as well as assured saving of return addresses, register contents, etc., is to store all temporary-storage and saved items in a stack (Sec. 4-10*b*); a stack pointer is advanced and retracted as the subroutine is called and completed (Fig. 4-11, see also Secs. 5-16 and 6-10).

RELOCATION AND THE LINKING LOADER

4-17. Problem Statement (see also Sec. 3-6). Minicomputers loading mainly single special-purpose programs or interpreter programs (such as BASIC, Sec. 3-8) will not require program relocation. For general-purpose computation, though, *one will want to combine different program segments and library subroutines*, so it must be possible to *relocate programs* anywhere in the computer memory. Program segments will, moreover, want to call other program segments as subroutines, and it will be necessary to pass arguments and results between programs. This requires techniques for **program linkage**, i.e., for associating the proper relocated addresses with symbolic names of external references. Programming systems permitting relocation and linkage will require:

1. An assembler (or compiler) *specifically designed* to permit relocation and linkage
2. A relocating/linking loader program, which supplies the correct addresses and cross references at load time

4-18. Relocation. An assembler (or compiler) designed to produce relocatable code creates a preliminary version of the object program, with addresses and program-counter readings normally referred to location 0 as a fictitious origin. The assembler (or compiler) will, moreover, **mark** every word, address, and symbol to be relocated with a **relocation bit, byte, or word** so that the loader will know which words and addresses to modify. These words and addresses usually appear marked with an R in the assembler listing (Fig. 4-3) and include:

1. Most of the normal *instruction and data words* of the program, with the exception of special pointers on page 0
2. Symbolic and numerical *addresses* in the program, again with the exception of special references to page 0

The nonrelocatable addresses are known as **absolute** addresses (see also Fig. 4-3).

The relocating/linking loader will complete the assembly (or compilation) process to produce the actual executable object program. The loader determines the true relative origin (relocation base) for each program segment, normally the first free location following the instructions and data of a preceding program. **This relocation base is then added to each address marked as relocatable by the assembler.**

Special problems may arise with the relocation of addresses specified as *symbolic expressions* (Sec. 4-3). While an expression like A + 2 will be relocated correctly if we simply add the relocation base to A, A + B + 3 will cause trouble if *both* A *and* B are relocatable addresses; the assembler may mark the line containing A + B + 3 as a "possible relocation error." The expression A - B, on the other hand, defines an *absolute* address if both A and B are relocatable.

Note also that minicomputers making extensive use of *relative addressing* (Sec. 2-7) will require fewer computations in the relocation process.

4-19. Linking External References. Assemblers intended for use with a linking loader usually require the user *to list all external references* (and frequently all symbols to be used as external references by other program segments) somewhere in the program, thus,

```
.EXT    A1, ARG, SYMB
```

Note that these "global" symbols must be uniquely defined, while symbols not used as external references can be used with different meanings in different program segments without causing any trouble.

A typical linking loader for a minicomputer operates much like another assembler. It creates a loader symbol table which includes the global symbols identified in each program segment, and then supplies the correct addresses after the relocation base for each program has been established. The loader symbol table is then used much like an assembler symbol table for the loader's "reassembly" job.

When a linking loader is given a library tape containing a set of utility programs (such as arithmetic or input/output routines), it will usually load only those routines which are actually requested by other programs.

4-20. Combination of Assembly-language Programs and FORTRAN Programs. Combinations of assembly-language and FORTRAN program segments are of substantial practical importance because:

1. FORTRAN READ, WRITE, and FORMAT statements are often the most convenient way to call the complicated formatting and I/O routines required to deal with numerical data on standard peripherals such as card readers and line printers. This is true even for minicomputers with relatively convenient input/output macros (see also Secs. 5-27 to 5-32).
2. Frequently used or special-purpose program segments may be written in assembly language for efficient execution and called as subroutines or functions by FORTRAN programs. Again, input/output routines, this time for nonstandard peripherals, are good examples.

In general, the FORTRAN compiler for a given minicomputer will expand a call to an assembly-language subroutine, say

```
CALL    SUBR (I,K)
```

into code corresponding to a standardized assembly-language calling sequence specified in the computer reference manual, e.g.,

```
.EXT                             SUBR / External reference
JUMP AND SAVE INDIRECT VIA       SUBR / Subroutine jump
JUMP                             .+ 3 / Jump around arguments
                                      / after return
.DEFINE ADDRESS                  I
.DEFINE ADDRESS                  K
```

and the assembly-language subroutine must access I and K according (Sec. 4-14). The FORTRAN compiler will expect a similar calling sequence when an assembly-language program calls a FORTRAN subroutine SUBR (I,K). Refer to your minicomputer manual for the specific conventions used to access floating-point or double-precision data.

MACROS AND CONDITIONAL ASSEMBLY

4-21. Macros. (a) Macro Definitions and Macro Calls. A macroassembler allows the user to define an entire sequence of assembly-language statements as a **macro instruction (macro)** called by a symbolic name. Each

macro is created by a *macro definition*, e.g.,

```
.MACRO       SUM      Z,X,Y
LOAD ACCUMULATOR             X
ADD INTO ACCUMULATOR         Y
STORE ACCUMULATOR IN         Z
.ENDMACRO
```

The pseudo-instruction words .MACRO and .ENDMACRO delimit the macro definition; SUM is the macro name, and Z, X, Y are dummy arguments. Once the macro is defined (which could be anywhere in a program), the user can employ a one-line macro call to generate the entire code sequence with new arguments as often as desired. Thus, the user-program sequence

```
START    SUM    A,A1,A2
         SUM    B,B1,B2
```

will produce code (and, if desired, a listing) corresponding to

```
START    LOAD ACCUMULATOR         A1     Expansion
         ADD INTO ACCUMULATOR     A2       of
         STORE ACCUMULATOR IN     A        SUM A,A1,A2
         LOAD ACCUMULATOR         B1     Expansion
         ADD INTO ACCUMULATOR     B2       of
         STORE ACCUMULATOR IN     B        SUM B,B1,B2
```

Note that the label START was not part of the macro-call expansion.

A macro may or may not have arguments. Arguments can be symbols, expressions, numbers, or literal constants and can appear as location tags as well as addresses. The better macroassemblers permit calls to other macros within a macro definition (see Sec 4-22*b* for an example).

Calls to *the same* macro (recursive macro calls) lead to complications but can produce interesting program sequences when used in conjunction with conditional assembly (Sec 4-23; see also Ref 9)

Beware of unintentionally using symbols other than arguments in macro definitions; they will stay the same in different macro-call expansions and may cause overwriting. Thus, if a sequence like

```
TEST    SKIP IF ACCUMULATOR POSITIVE
        JUMP TO                 ACT
        JUMP TO                 ACT +2
ACT     INVERT ACCUMULATOR
```


appears in a macro definition, *it should be replaced with*

```
SKIP IF ACCUMULATOR POSITIVE
JUMP TO                 .+2
JUMP TO                 .+3
INVERT ACCUMULATOR
```

Some macroassemblers have the facility of automatically "creating" new symbolic labels in such situations when the macro is called more than once, but the necessary procedures rather complicate programming.

(b) Importance of Macros. Macros are not simply a programming convenience or shorthand notation: their power in enlarging the scope of assembly-language programming can hardly be overemphasized. A macroassembler permits you to create and use entire classes of new computer operations, which can *simplify programming* and/or *help with applications-related modeling*.

(c) Macros and Subroutines. As we saw in Sec. 4-14, functional program modules can also be called as *subroutines*, with arguments and results in appropriate calling sequences. It is important to distinguish between subroutines and macros. Each macro call will generate *new in-line code* so that long macros will *not* save memory like long subroutines. Short macros can be more economical than short subroutines because of the overhead associated with subroutine jumps, calling sequence, and data transfers; in any case, macros will execute more quickly.

Macro calls with multiple arguments are more "natural" for most programmers than subroutine calling sequences. Hence it is convenient to define the complete data-transfer and calling sequences of frequently used subroutines as subroutine-calling macros. This technique is used, in particular, to define *system macros calling input/output subroutines* (Sec. 5-31).

4-22. Two Interesting Applications. (a) Computer Emulation. To emulate the operation (instruction set) of a different digital computer on an existing "host" computer, we can write a macro for each computer instruction to be simulated. If we can take care of input/output, our "host" computer should then be able to run any assembly-language program written for the emulated "target" computer.

EXAMPLE: Emulation of indexed addition on a single-accumulator minicomputer. The memory location INDEX simulates a single index register in the "target" computer.

```
.MACRO    ADDX    A
STORE ACCUMULATOR IN     SAVAC / Save accumulator
LOAD ACCUMULATOR         (A)   / Compute
ADD INTO ACCUMULATOR     INDEX /      indexed
STORE ACCUMULATOR IN     ADDR  /      address
LOAD ACCUMULATOR         SAVAC / Restore accumulator
ADD INDIRECT VIA         ADDR  / Perform addition
.ENDMACRO
```

Note that in this example the temporary-storage symbols SAVAC and ADDR will *not* cause trouble in later macro calls.

See also Sec. 6-13 for other computer-emulation techniques (microprogramming).

(b) Writing Simple Procedural Languages. Macros make it possible to write application programs solely in terms of operations directly related to the user's application. Once the macros have been written (perhaps by a professional programmer), the lucky user will be able to write his application programs using only a few simple rules (syntax) *without knowing any assembly*

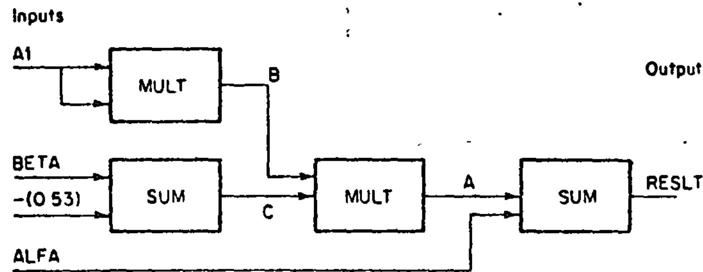


Fig. 4-12. A block diagram producing
RESLT = A1²(BETA − 0.53) + ALFA

language at all. As a generally applicable example, we shall develop a block-diagram language suitable for doing any sort of arithmetic and/or function generation with fixed-point numbers (integers and scaled fractions, Sec. 1-8). The user need not know assembly language; the block-diagram language will actually be a simple substitute for a compiler language and will generate remarkably efficient code.

Scaled-fraction *inputs* $X1, X2, \ldots$ and *outputs* $Y1, Y2, \ldots$ will be referred to as symbolic locations, whose contents can be accessed by input/output routines (which can also be called in macro form) as needed. We now define a set of macros for *algebraic-operation blocks* (Table 4-1), plus extra blocks for function generation (sine, cosine, table-lookup functions) if we need them. *We can now combine such blocks to compute any reasonable scaled expression*, say

$$\text{RESLT} = A1^2(\text{BETA} - 0.53) + \text{ALFA}$$

in terms of a corresponding *block diagram*, much like an analog-computer block diagram (Fig. 4-12). We can then write an assembly-language routine producing the desired expression by simply listing the block macros

TABLE 4-1. Macros for a Simple Block-diagram Language (Sec. 4-22b)
Extra blocks, such as function-generator blocks (sine, cosine, table lookup functions) can be added at will. *X*, *Y*, and *Z* are scaled fractions.

.MACRO SUM Z, X, Y		/ $Z = X + Y$
LOAD ACCUMULATOR	X	
CLEAR OVERFLOW FLAG		
ADD INTO ACCUMULATOR	Y	
OTEST	Z	/ Overflow-test macro, see below
STORE ACCUMULATOR IN	Z	
.ENDMACRO		
.MACRO NEGATE Z,X		/ $Z = -X$
LOAD ACCUMULATOR	X	
INVERT ACCUMULATOR		
STORE ACCUMULATOR IN	Z	
.ENDMACRO		
.MACRO SCALE Z, X, M		/ $Z = 2^M X$, where M is a positive integer
LOAD ACCUMULATOR	X	
CLEAR OVERFLOW FLAG		
LONG SIGNED SHIFT LEFT, M BITS		
OTEST	Z	
STORE ACCUMULATOR IN	Z	
.ENDMACRO		
.MACRO MULT Z, X, Y		/ $Z = XY$
LOAD ACCUMULATOR	X	
MULTIPLY BY	Y	/ $XY/2$ in accumulator (Sec. 1-9)
LONG SIGNED SHIFT LEFT, 1 BIT		/ XY in accumulator
STORE ACCUMULATOR IN	Z	
.ENDMACRO		
.MACRO DIV Z, X, Y		/ $Z = X/Y$
LOAD ACCUMULATOR	X	
CLEAR OVERFLOW FLAG		
DIVIDE BY	Y	
OTEST	Z	
MOVE MQ TO ACCUMULATOR		
STORE ACCUMULATOR IN	Z	
.ENDMACRO		
.MACRO OTEST	Z	/ Test for overflow of fraction
SKIP ON OVERFLOW FLAG		/ Overflow?
JUMP TO	.+3	/ No, go on
LOAD ACCUMULATOR	(Z)	/ Identifies guilty variable for error-message routine
JUMP TO	ERROR	/ Error-message routine
.ENDMACRO		

with their input and output variables:

```
                .DECIMAL
        START   MULT    B, A1, A1
                SUM     C, BETA, (-0.53)
                MULT    A, B, C
                SUM     RESLT, A, ALFA
```

As in any procedural language, *we have taken care to write any intermediate result* A, B, C, *as well as the result* RESLT *only if it has been computed (as a block output) in a preceding line*. Such a program is most easily written when we start with the *last* block: Our simple scheme is, in fact, simulating the reverse Polish string generated by an algebraic compiler! After some practice, we may not even have to draw the block diagram.

Our simple block-diagram language generates quite efficient code (probably better than most minicomputer FORTRAN). It includes an error-message routine (not shown in Table 4-1) which will print out the symbol-table number of any block-output variable which overflows because of faulty scaling. A similar set of blocks could readily be written for floating-point arithmetic.

Expansion of our macro blocks shows that almost all block macros end with STORE ACCUMULATOR IN Q, which is often followed by LOAD ACCUMULATOR Q in the next macro. Such store/fetch pairs are redundant; each wastes four memory cycles. Our block-diagram language can be modified (or reprocessed) to cancel redundant store/fetch pairs (Refs. 9 and 10); the resulting code can be as efficient as that written by a good assembly-language programmer.

4-23. Conditional Assembly. Conditional assembly directs the assembler *to suppress specified sections of code unless stated conditions are met by the program at assembly time*. Conditional assembly is available with some ordinary assemblers but is most useful with macroassemblers. One can, in particular, modify the definitions of user-defined or system macros if named symbols do or do not appear in the program or if certain symbolic variables are zero, positive, or negative at *assembly time*.

Specifically, all statements (instructions and/or data) between the pseudo instructions .IFDEF X and .ENDCOND will be assembled if and only if the named symbol X is defined anywhere in the program. Other conditions are similarly employed by the pseudo instructions .IFUNDEF, .IFZERO, .IFNONZR, .IFPOS, .IFNEG. Note that the condition expressed by

```
        .IFZERO     A - B
```

means that the symbols A and B reference the same variable or memory location.

As a very simple example, consider a multi-input summer block for the simple algebraic block-diagram language of Sec. 4-22*b*. We will write a macro to add a maximum of four inputs, $X1$, $X2$, $X3$, and $X4$, to produce an output Z:

```
.MACRO   SUMR                  Z, X1, X2, X3, X4
 LOAD ACCUMULATOR              X1
 CLEAR OVERFLOW FLAG
 ADD INTO ACCUMULATOR          X2
.IFDEF                         X3
 ADD INTO ACCUMULATOR          X3
.ENDCOND
.IFDEF                         X4
 ADD INTO ACCUMULATOR          X4
.ENDCOND
 OTEST
 STORE ACCUMULATOR IN          Z
.ENDMACRO
```

We now see the beauty of the conditional-assembly feature. If our four-input summer is given only *three* inputs, say X1, X2, and X4, with X3 undefined in our program, then the assembler will omit the unneeded ADD INTO ACCUMULATOR X3; this saves memory and execution time. We could similarly omit X4 or both X3 and X4.

4-24. Nested Macro Definitions. We mentioned in Sec. 4-21 that macro definitions may contain macro *calls* (see also Table 4-1). A macro definition which contains another macro *definition*, as in

```
.MACRO     MAC1           Z, X
 LOAD ACCUMULATOR            X
 STORE ACCUMULATOR IN        Z
.MACRO     MAC2           U, V
 XOR INTO ACCUMULATOR     V
 STORE ACCUMULATOR IN     U
.ENDMACRO
 ROTATE ACCUMULATOR LEFT
.ENDMACRO
```

(nested definitions), is a different situation. The assembler regards MAC2 as *undefined* in the part of our program preceding the first call for MAC1, say

```
        MAC1     P, Q
```

This results in the expansion

```
LOAD ACCUMULATOR          Q
STORE ACCUMULATOR IN      P
ROTATE ACCUMULATOR LEFT
```

Note that *no code due to* MAC2 *is generated this time*, but MAC2 is now defined and can be called either alone or through the next call to MAC1. Multiple nesting of definitions is possible. We have here another means of turning assembly of a section of code off and on.

REFERENCES AND BIBLIOGRAPHY

1. Flores, I.. *Computer Programming*, Prentice-Hall, Englewood Cliffs, N J., 1966.
2. ------: *Computer Software*, Prentice-Hall, Englewood Cliffs, N J . 1965.
3. Gear, C. W. *Computer Organization and Programming*, McGraw-Hill, New York, 1969.
4. Maurer, W. D *Programming An Introduction to Computer Languages and Techniques*, Academic Press, New York, 1969
5. *Introduction to Programming*, PDP-8 Handbook Series, Digital Equipment Corporation, Maynard, Mass (current edition)
6. *Programming Languages*, PDP-8 Handbook Series, Digital Equipment Corporation, Maynard, Mass (current edition).
7. *MACRO-15 Manual*, Digital Equipment Corporation, Maynard, Mass. (current edition).
8. *DAP/15 Mod 2 Assembler Manual*, Honeywell Information Systems, Framingham, Mass. (current edition)
9. McIlroy, M. D Macro-instruction Extensions of Compiler Languages, *Comm. ACM*, April 1960.
10. Liebert, T. A.: "The DARE II Simulation System," Ph.D. thesis, University of Arizona, 1970.

CHAPTER 5

INTERFACING THE MINICOMPUTER WITH THE OUTSIDE WORLD

INTRODUCTION AND SURVEY

The exceptional power of the small digital computer is substantially based on its ready interaction with real-world devices—analog-to-digital converters (ADCs), digital-to-analog converters (DACs), transducers, displays, logic controllers, alarm systems—in addition to the usual card readers, line printers, and tape drives. To the outside world, the minicomputer presents a relatively small number (30 to 80) of bus-line terminations. These lines transmit and receive digital data words together with a few command pulses and control levels, which select devices and functions or alert the computer, in turn, to new real-world situations.

This chapter introduces the basic logic and programming principles for such interfaces. With inexpensive, off-the-shelf digital and analog system components widely and readily available, a little knowledge of interfacing principles can produce dramatically effective new systems and also surprising cost savings. A handful of integrated circuits, cards, and connectors, which you can wire-wrap yourself for a total cost of $500, quite easily gets to be a $3,000 subsystem if you purchase it from an instrument manufacturer.

Sections 5-1 through 5-8 deal with *program-controlled input/output and sensing operations*. Sections 5-9 to 5-16 describe minicomputer *interrupt systems*—the basic means for time-sharing the small computers between different time-critical tasks. Sections 5-17 to 5-23 deal with *direct memory access* and *automatic block transfers*, which permit not only remarkable time savings but also more manageable input/output programming. Th

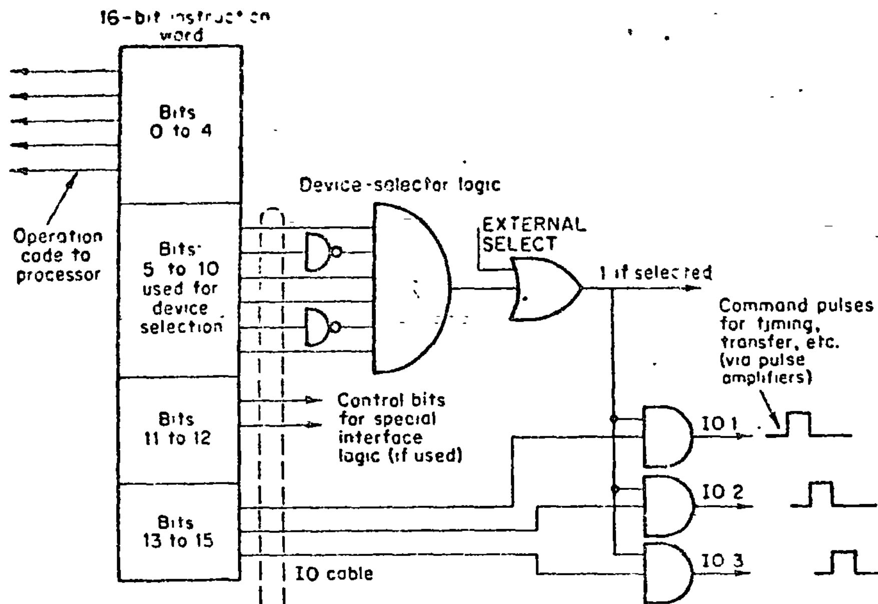


Fig. 5-3. Program-controlled selection (decoding) of device addresses and device functions (*based on Ref 6*)

3. Bits 11 and 12 (control bits, select bits, or subdevice bits) determine levels on two control lines. These can be used to select additional devices or different functions to be performed by a given device.
4. Bits 13, 14, and 15, respectively, produce successive timed command pulses IO1, IO2, and IO3 on three separate control lines. A pulse occurs if the corresponding bit is 1.

With the arrangement of Fig. 5-3, a 16-bit I/O instruction can select one of $2^{11} = 2{,}048$ possible devices and/or device functions through different combinations of device-address bits, control bits, and command pulses. This particular system requires four complete digital-computer memory cycles for each I/O operation, one to fetch the instruction, and one for each IO pulse. Many modifications of our basic programmed-I/O scheme are possible, e.g.,

1. Different numbers of processor-code bits, device-selection bits, control bits, and IO pulses may be used.
2. IO pulses can be simultaneous (on different lines), rather than successive, to save execution time.

Some more incisive modifications will be described in Secs. 5-6, 5-7, and 6-9.

5-3. Programmed Data Transfers. The most common application of the device-selector-gated command pulses is data transfer to and from the

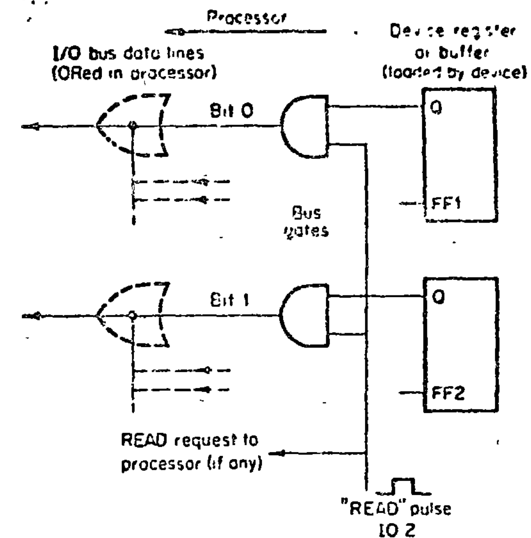


Fig. 5-4a. Programmed transfer of data into the processor.

processor. In our basic programmed-I/O scheme, the IO pulses are synchronized with the processor operation cycle, and thus with the computer's ability to accept or transmit data.

In Fig. 5-4*a*, the correctly timed IO2 pulse gates data from an external device (e.g., an ADC) into a processor register (accumulator) via the I/O-bus data lines.

Figure 5-4*b* illustrates clear-and-strobe data transfer from the I/O data lines into the flip-flops of a device register. Each flip-flop is first cleared by

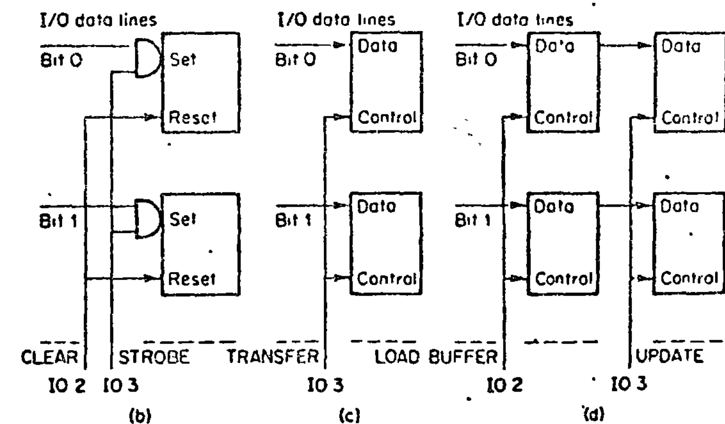


Fig. 5-4b to *d*. Programmed data transfer into device registers. clear-and-strobe (*b*) jam transfer (*c*) and transfer through a device buffer register, as in a double-buffered digital-to-analog converter (*d*).

remainder of the chapter adds a little *hardware know-how* and discusses the elements of *input/output programming*. Additional applications and examples will be given in Chap. 7.

PROGRAMMED I/O OPERATIONS

5-1. The Party-line I/O Bus. Minicomputers usually transmit digital data on parallel 8- to 18-bit buses; i.e., all data bits are transmitted simultaneously in the interest of processing speed. Serial data transmission is usually restricted to communication links.

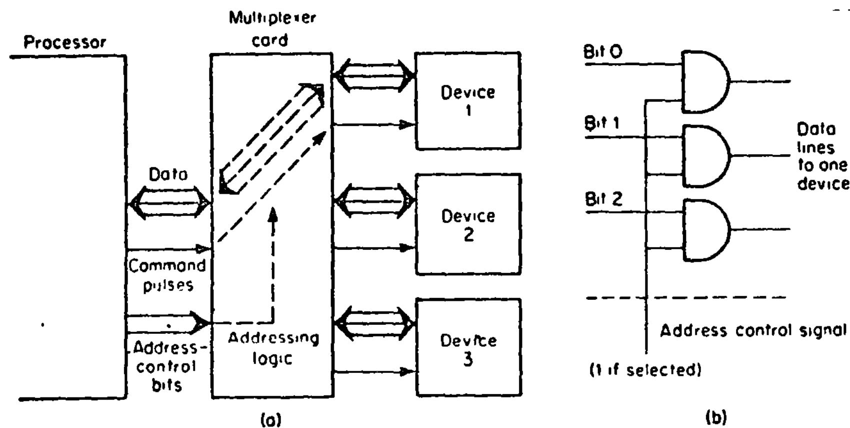


Fig. 5-1. Simple switching (multiplexing) of multiple bus lines with a multiplexer card in the processor cabinet (*a*) and switching circuits for one set of processor output data lines (*b*). In this arrangement, bus-driving circuits are loaded only by one processor-to-device bus at a time, and no device addresses need be transmitted over the bus. But multiple-line switching becomes cumbersome if there are more than a few devices.

The 8- to 18-data-bit lines can be **bidirectional** (this takes extra logic at each device, Fig. 5-19*c*), or we can have separate **input and output buses.** (This takes less logic but more interconnections, Fig. 5-19*b*.) In addition to the data lines, we will need a comparable number of **interface-control-logic lines.**

If the computer must service only a few external devices, processor instructions can select individual buses for each device through multiplexing gates (Fig. 5-1). But the circuits needed to multiplex data and control lines for more than 4 (and perhaps as many as 1,000) devices could compromise the processor design. Thus, most interface systems employ a **party-line I/O bus** of the general type illustrated in Fig. 5-2. Here, *all* "devices" (printers, displays, ADCs, DACs, etc.) intended to receive or transmit data words are wired to a parallel I/O data bus connected to a processor register via suitable logic. Additional party-line wires carry control-logic signals

for selecting a specific device and its function (e.g., transmission or reception) and synchronize data transfers with the digital-computer operating cycle.

5-2. Program-controlled Device Selection and Operation. For a minimum of linkage hardware, interfaces work with programmed digital-computer instructions (input/output instructions, I/O instructions; refer to Sec. 5-17 for

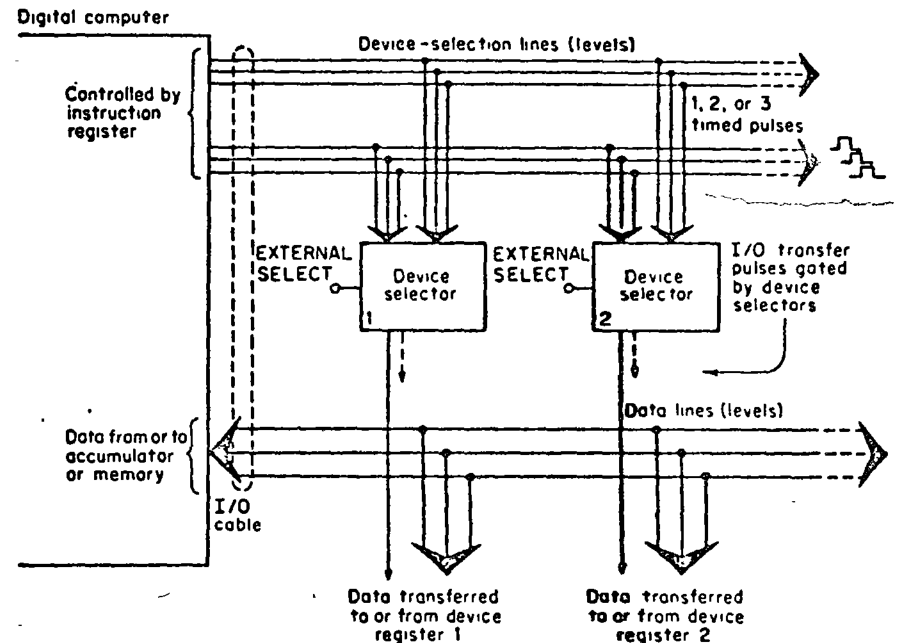


Fig. 5-2. Programmed control of multiple devices by a minicomputer with a party-line I/O bus. An I/O instruction addressed to a specific device is recognized by a device selector, which gates data-transfer or command pulses to the device in question (*based on Ref. 6*).

direct-memory-access operation). **Figure 5-3 shows how the individual bits of an input/output instruction word in a typical processor determine device-selection and control-line signals on a party-line I/O bus:**

1. Bits 0 to 4 tell the processor that an I/O **instruction** is wanted. One of these bits can select a READ or WRITE **operation**, or this decision may be left to a logic input from the device.
2. Bits 5 to 10 (device-address bits) place logic levels (0 or 1) on **device-selection lines** parallel-connected to all devices on the I/O bus. When these lines carry the device-selection code assigned to a specific device, its **device selector** (decoding AND gate) **accepts** (and regenerates) a set of one, two, or three successive command pulses (IO pulses) used to effect data transfers and other operations in the selected device as determined by instruction bits 13 to 15.

IO2; then IO3 strobes the 1s on the data bus into the flip-flop register. Figure 5-4c shows jam transfer of bus data into a device register (see also Sec. 1-6). Jam transfers require only a single transfer pulse and must be employed whenever clearing and strobing operations would disturb device functions. This is true, for instance, with DACs required to switch through successive voltage levels without returning to 0 in between.

Figure 5-4*d* illustrates a double-buffered-register data transfer. Data are first transferred into the buffer register and then into the device register proper. In an analog hybrid computer or display system, for instance, one can load a set of DAC buffer registers in turn and then "update" all DAC registers simultaneously with another I/O instruction or with a clock pulse.

5-4. Device Control Registers. Many peripheral devices have more different functions or operating modes than we can control with a few control bits or IO-pulse bits in a single I/O instruction. Such devices can be designed to accept, store, and execute multibit "device instructions" loaded into a device control register or registers via the I/O-bus data lines (just like into a data register). In general, control registers will require jam transfers or double-buffered transfers (Fig. 5-4*b* and *c*). An important example of a control register is the multiplexer control register for selecting different multiplexer input channels for an ADC. Control registers may permit incrementation; i.e., the control register can be a counter set to a given initial count by an I/O instruction and then incremented by I/O pulses as needed; the variety of possible arrangements is endless. Control registers of many old-fashioned process controllers simply operate *electromechanical relays*.

Typical examples of more complex devices whose status and function are established by computer-controlled registers are process controllers, cathode-ray-tube displays (Sec. 7-9), automatic data channels (Sec. 5-19), analog/hybrid computers (Sec. 7-18) and other digital computers.

5-5. Interfacing with Incremental and Serial Data Representations. Device-selector-gated command pulses (IO pulses) are not used only to transfer data and control-register settings. Command pulses can also set, reset, or complement special flip-flops and increment or decrement counters in device interfaces (Fig. 5-5).

Computer-read digital counters can also accumulate external incremental data (variables proportional to pulse rates) into parallel digital words. Incremental data representation is employed in control and navigation systems based on *digital-differential-analyzer* (*DDA*) *integrators* (Ref. 19).

Digital communication systems, teletypewriter keyboard/printers, and disk or drum storage systems employ serial data representation (Sec. 1-3). Parallel-to-serial and serial-to-parallel conversion is accomplished by either

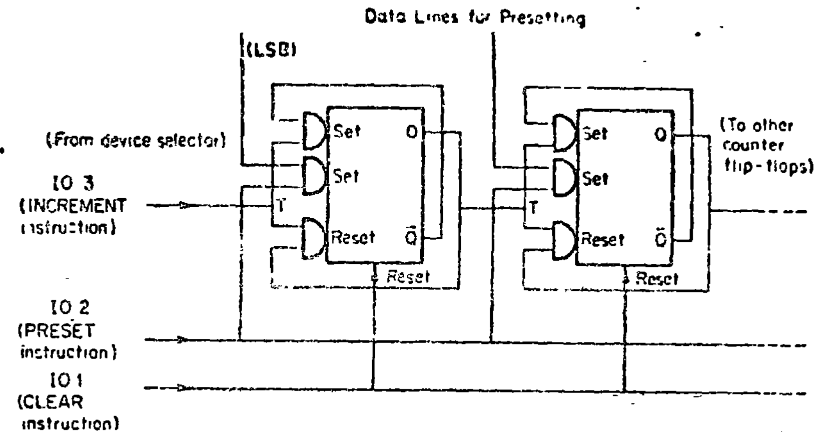


Fig. 5-5. Three different processor instructions employ one of three successive command pulses from the same device selector to clear, strobe, or increment a binary counter. A fourth instruction combines IO1 and IO2 for clear-and-strobe presetting (see also Table 1-5*b*).

of two methods:

1. Serial n-bit data words (usually 8-bit bytes) are shifted in or out of a shift register in the interface by shift pulses from a clock oscillator, which is stopped by a control counter after n, $n + 1$, or $n + 2$ shift pulses. The extra shift pulses transmit or receive start and stop bits marking spaces between serial words in some systems. The shift register is parallel-loaded or read by the digital computer like any other device register.
2. Shift pulses cause interrupt-activated processor instructions for shifting data words bit by bit into or out of an accumulator carry bit.

5-6. Timing Considerations: Synchronous and Asynchronous I/O Operations. Programmed data-transfer operations with processor-timed command pulses require that an external device accept or transmit data levels within the allotted instruction time. More specifically, timing diagrams like Fig. 5-6, supplied in every minicomputer interface manual, show (perhaps obviously) that *a set of data-bit levels must be established when the processor issues the data-transfer command pulse* (*IO pulse*).

We will assume here that our device (say an ADC) is already *prepared* to transfer data in the sense that an ADC conversion has been completed; we can, and should, make sure of this through a sense instruction (Sec. 5-8) or interrupt operation (Sec. 5-9). What we *are* concerned with here is that data levels may not be established *soon enough* or *long enough* to complete a data transfer, allowing for cable-transmission and logic delays and rise times. Cable delays are, at best, about 1.5 nsec/ft; programmed-I/O instruction timing usually allows for up to 50 ft of I/O bus cable.

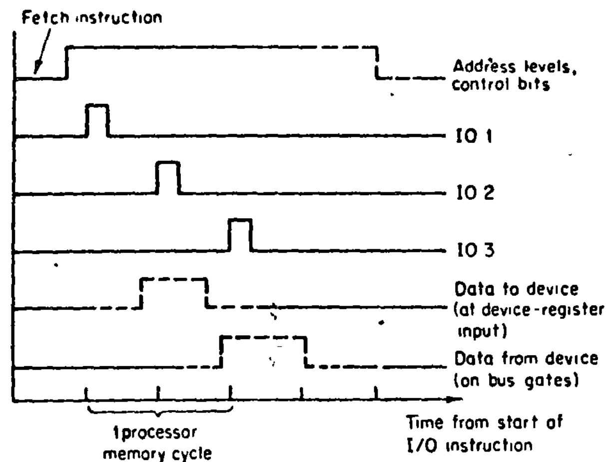


Fig. 5-6. Typical timing for processor-synchronized programmed data transfers. Data levels must be ready for the bus when transfer pulses occur.

When cable and/or logic delays become critical, we can escape the tyranny of the processor clock through **asynchronous ("hand-shaking") I/O operations.** An asynchronous data-transfer instruction addresses an external device with address levels, as in Sec. 5-3. But the processor does not issue an automatically timed IO pulse. Instead, it waits until the device-selector-gate output has set an ADDRESS ACTIVE flip-flop in the addressed device (Fig. 5-7). The resulting voltage step returns to the processor over a special interface line; *only then* will the processor issue the appropriate IO pulse or pulses. The processor may now continue with the next instruction after a

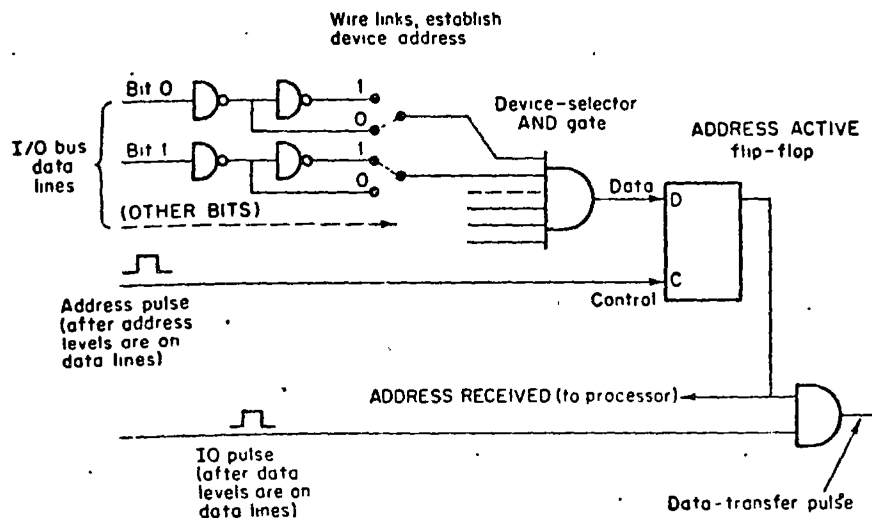


Fig. 5-7. An ADDRESS ACTIVE flip-flop (type D flip-flop, Table 1-5h) indicates reception of the devic[illegible]dress and also stores the device-selector output when the address levels are no longer o[illegible]bus. Note that all ADDRESS ACTIVE flip-flops which are *not* addressed will [illegible] each device address.

decent interval. If we wish to be even more careful, the processor may not continue its operation until the data-transfer pulse actuates another device-response signal signifying that the IO pulse has been received and that the data transfer is complete.

5-7. Minicomputers Using Data Lines for Device Addressing. To save interconnections at the expense of time and interface hardware, some minicomputer I/O systems transmit the device address over the *data lines.* After the address levels are on the bus, an ADDRESS control level or pulse from the processor sets an ADDRESS ACTIVE flip-flop in the addressed device and resets all other ADDRESS ACTIVE flip-flops (Fig. 5-7). The output of the ADDRESS ACTIVE flip-flop substitutes for the simple device-selector output of Fig. 5-3; it can activate non-data-transfer operations immediately, or *it can gate data transfers from and to the bus after the address has been removed from the data lines.* There are two main types of such systems:

1. In the Varian Data Systems 620i and 620f, the successive addressing and data-transfer operations form part of *the same* 16-bit I/O instruction; the address bits come from the processor *instruction* register, as in Sec. 5-2.
2. In the 8-bit Interdata Model 1, we must first load the desired device address into the *accumulator.* Then *a separate addressing instruction* places the address bits from the accumulator on the data line to activate the selected device. Data-transfer operations (to or from the accumulator) follow; note that one addressing operation may do for several data transfers from and/or to the same device.

5-8. Sense-line Operation and Status Registers. The IO pulses implement program-controlled operations at times determined by the digital-computer program. But a device thus addressed might not be ready; an important example is an ADC which has not completed a conversion. In such cases, program-controlled data or control-logic transfers may be preceded by a **sense-line interrogation or flag test.** The device status is indicated by a flag (logic level, usually a flip-flop output). A special instruction addressed to the associated device selector gates one of the command pulses (IO1 in Fig. 5-8) into a sense gate and, if the flag level (sense line) is up, onto the skip bus in the interface cable. The pulse on the skip bus then increments the processor instruction counter, which thus skips the next instruction to produce a program branch. An example would be

```
SKIP IF ADC FLAG IS UP
REPEAT LAST INSTRUCTION
READ ADC
```

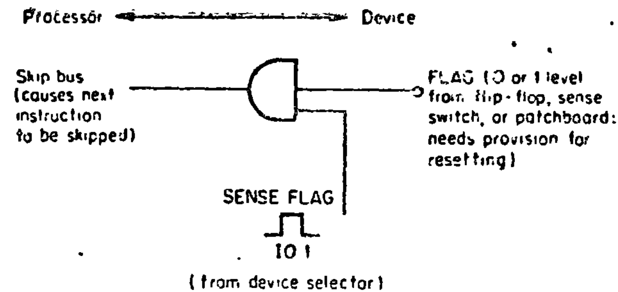


Fig. 5-8. Sense-line operation using a device selector, sensing gate, and common skip bus.

The program will cycle until the device flag is up, at which time the program continues, usually with a data-transfer instruction. The ADC flag must be *reset* by the READ ADC instruction and/or by the START CONVERSION instruction or timing pulse.

Other sense-line systems exist. Many computers do away with device-selector addressing of sense gates by simply accepting flag levels on multiple sense lines, which are interrogated with processor instructions like SKIP IF SENSE LINE 5 IS UP.

Flag logic levels controlled by manually operated sense switches permit a human operator to control program branching during computation.

Instead of interrogating multiple sense lines in turn, we can treat several flag flip-flops (which may or may not be associated with the same peripheral device) as a device status register. The processor can read this register (READ FLAGS or READ STATUS instruction) via the data bus. The resulting "status word" in the processor accumulator can then be logically interpreted by the computer program; in particular, the status word may serve as an indirect address for the next instruction and thus permit multiple branching.

Sense/skip instructions can be profitably combined with data-transfer operations. Thus, the single instruction SKIP AND READ ADC IF ADC FLAG IS UP combines two instructions by using the IO1 pulse to test the flag and the IO3 pulse to transfer data; the flag level itself is employed to gate IO3 off if the ADC is not ready.

I/O systems with *asynchronous data transfer* (Sec. 5-6) do not need a sense/skip loop to wait for device readiness. The device-flag level can simply operate a gate which keeps the ADDRESS RECEIVED signal in Fig. 5-7 from returning to the processor until the device is ready for the data transfer.

Sense lines are inexpensive, but even small digital computers can rarely afford the time for "idle" sense/skip loops such as the simple ADC example shown above. Sense lines are, therefore, used mainly for decisions between device-dependent program branches which do *not* cause idling. Otherwise we require *program interrupts*.

INTERRUPT SYSTEMS

5-9. Simple Interrupt-system Operations. In an interrupt system, a device-flag level (INTERRUPT REQUEST) interrupts the computer program on completion of the current instruction. Processor hardware then causes a subroutine jump (Sec. 4-12):

1. Contents of the incremented program counter and of other selected processor registers (if any) are automatically saved in specific memory locations or in spare registers.
2. The program counter is reset to start a new instruction sequence (interrupt-service subroutine) from a specific memory location ("trap location") associated with the interrupt. The interrupt thus acted upon is *disabled* so that it cannot interrupt its own service routine.

Minicomputer interrupt-service routines must usually first *save the contents of processor registers (such as accumulators) which are needed by the main program, but which are not saved automatically by the hardware.* We might also have to save (and later restore) some peripheral-device control registers. Only then can the actual interrupt service proceed: the service routine can transfer data after an ADC-conversion-completed interrupt, implement emergency-shutdown procedures after a power-supply failure, etc. Either the service routine or the interrupt-system hardware must then *clear the interrupt-causing flag* to prepare it for new interrupts. The service routine ends by *restoring registers and program counter to return to the original program*, like any subroutine (Sec. 4-12). As the service routine completes its job, it must also *reenable the interrupt.*

EXAMPLE: Consider a simple minicomputer which stores only the program counter automatically after an interrupt. The interrupt-service routine is to read an ADC after its conversion-complete interrupt.

Location	Label	Instruction or Word Data (main program)	Comments
.		.	
.		.	
.		.	
1713		current instruction	/ Interrupt occurs here
0000		1714	/ Incremented program / counter (1714) will be / stored here by hard- / ware
0001		JUMP TO SRVICE	/ Trap location, contains / jump to relocatable / service routine
3600	SRVICE	STORE ACCUMULATOR IN SAVAC	

```
3600   SRVICE  STORE ACCUMULATOR IN  SAVAC   / Save accumulator
3601           READ ADC                      / Read ADC into
                                             /   accumulator and
                                             /   clear ADC flag
3602           STORE ACCUMULATOR IN  X       / Store ADC reading
3603           LOAD ACCUMULATOR      SAVAC   / Restore accumulator
3604           INTERRUPT ON                  / Turn interrupt back on
3605           JUMP INDIRECT VIA     0000    / Return jump
                   .
1714           (main program)                / Interrupted program
  .                .                         /   continues
  .                .
  .                .
```

NOTE: Interrupts do not work when the computer is HALTed, so *we cannot test interrupts when stepping a program manually.*

5-10. Multiple Interrupts. Interrupt-system operation would be simple if there were only one possible source of interrupts, but this is practically never true. Even a stand-alone digital computer usually has several interrupts corresponding to peripheral malfunctions (tape unit out of tape, printer out of paper), and flight simulators, space-vehicle controllers, and process-control systems may have *hundreds* of different interrupts.

A practical multiple-interrupt system will have to:

1. "Trap" the program to different memory locations corresponding to specific individual interrupts
2. Assign priorities to simultaneous or successive interrupts
3. Store lower-priority interrupt requests to be serviced after higher-priority routines are completed
4. Permit higher-priority interrupts to interrupt lower-priority service routines as soon as the return address and any automatically saved registers are safely stored

Note that programs and/or hardware must carefully save successive levels of program-counter and register contents, which will have to be recovered as needed. Interrupt-system programming will be further discussed in Sec. 5-16.

More sophisticated systems will be able to *reassign new priorities through programmed instructions* as the needs of a process or program change (see also Secs. 5-12, 5-14, and 5-16).

5-11. Skip-chain Identification of Interrupts. The most primitive multiple-interrupt systems simply OR all interrupt flags onto *a single interrupt line. The interrupt-service routine then employs sense/skip instructions* (Sec. 5-8) *to test successive device flags in order of descending priority.*

Suppose that the simple interrupt system discussed in Sec. 5-9 was connected not only to the ADC requesting service but also to "emergency" interrupts from a fire alarm and from the computer power supply (Sec. 2-15). A skip-chain service routine with appropriate branches for fire alarm, emergency shutdown, and ADC might look like this (only the ADC service routine is actually shown):

```
SRVICE  SKIP IF FIRE-ALARM FLAG LOW          / Fire alarm?
        JUMP TO FIRE                         / Yes, go to service
                                             /   routine
        SKIP IF POWER FLAG LOW               / No: power-supply
                                             /   trouble?
        JUMP TO  LOWPWR                      / Yes, go to service
                                             /   routine
        SKIP IF ADC DONE FLAG LOW            / No: ADC service
                                             /   request?
        JUMP TO  ADC                         / Yes, service it
        JUMP TO  ERROR                       / No; spurious
                                             /   interrupt -print
                                             /   error message
ADC     STORE ACCUMULATOR IN  SAVAC          / ADC service routine
        READ ADC
        STORE ACCUMULATOR IN  X
        LOAD ACCUMULATOR      SAVAC          / Restore accumulator
        INTERRUPT ON                         / Turn interrupts back
                                             /   on
        JUMP INDIRECT VIA     0000           / Return jump
```

The skip-chain system requires only simple electronics and disposes of the priority problem, but the flag-sensing program is time-consuming. (n devices may require $\log_2 n$ successive decisions even if the flag sensing is done by successive binary decisions). A somewhat faster method is to employ a *flag status word* (Sec. 5-8), which can be tested bit by bit or used for indirect addressing of different service routines (Sec. 4-11*a*).

Note also that our primitive ORed-interrupt system must automatically disable *all* interrupts as soon and as long as any interrupt is recognized. We cannot interrupt even low-priority interrupt-service routines.

5-12. Program-controlled Interrupt Masking. It is often useful to enable (arm) or disable (disarm) individual interrupts under program control to meet special conditions. Improved multiple-interrupt systems gate individual interrupt-request lines with mask flip-flops which can be set and reset by programmed instructions. The ordered set of mask flip-flops is usually treated as a control register (interrupt mask register) which is loaded with

appropriate 0s and 1s from an accumulator through a programmed I/O instruction. Groups of interrupts quite often have a common mask flip-flop (see also Sec. 5-14).

A very important application of programmed masking instructions is to give selected portions of main programs (as well as interrupt-service routines) greater or lesser protection from interrupts.

Note that we will have to restore the mask register on returning from any interrupt-service routine which has changed the mask, so program or hardware must keep track of mask changes. We must also still provide programmed instructions to enable and disable the entire interrupt system without changing the mask.

EXAMPLE: *A skip-chain system with mask flip-flops.* Addition of mask flip-flops to our simple skip-chain interrupt system (Fig. 5-9) makes it practical to interrupt lower-priority service routines. *Each such routine must now have its own memory location to save the program counter*, and the mask must be restored before the interrupt is dismissed. The ADC service routine of Sec. 5-11 is modified as follows (all interrupts are initially disabled):

```
ADC  STORE ACCUMULATOR IN   SAVAC
     LOAD ACCUMULATOR       0000      / Save program
     STORE ACCUMULATOR IN   SAVPC     /   counter
     LOAD ACCUMULATOR       MASK      / Save
     STORE ACCUMULATOR IN   SVMSK     /   current mask
     LOAD ACCUMULATOR       MASK 1    / Arm higher-
     LOAD MASK REGISTER               /   priority interrupts
     INTERRUPT ON                     / Enable interrupt system
     READ ADC
     STORE ACCUMULATOR IN   X
     INTERRUPT OFF
     LOAD ACCUMULATOR       SVMSK     / Restore
     LOAD MASK REGISTER               /   previous
     STORE ACCUMULATOR      MASK      /   mask, and
     LOAD ACCUMULATOR       SAVAC     /   restore accumulator
     INTERRUPT ON
     JUMP INDIRECT VIA      SVPC      / Return jump
```

Since most minicomputer mask registers cannot be read by the program, the mask-setting is duplicated in the memory location MASK. Some minicomputers (e.g., PDP-9, PDP-15, Raytheon 706) allow only a restricted set of masks and provide special instructions which simplify mask saving and restoring (see also Sec. 5-15). Machines having two or more accumulators can reserve one of them to store the mask and thus save memory references.

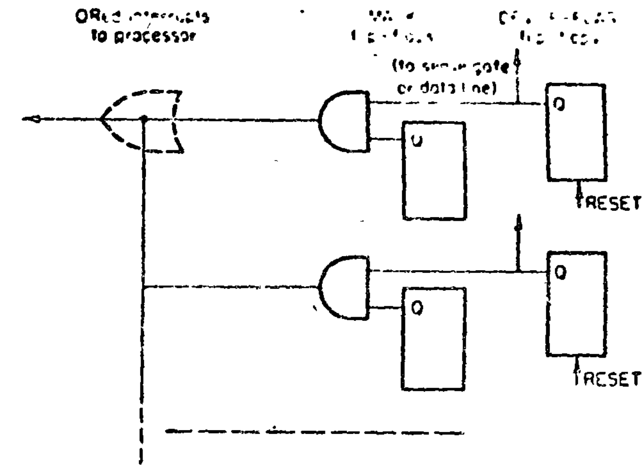


Fig. 5-9. Interrupt masking. The mask flip-flops are treated as a control register (*mask register*), which can be cleared and loaded by I/O instructions.

5-13. Priority-interrupt Systems: Request/Grant Logic. We could replace the skip-chain system of Sec. 5-11 with *hardware* for polling successive interrupt lines in order of descending priority, but this is still relatively slow if there are many interrupts. We prefer the priority-request logic of Figs. 5-10 or 5-11, which can be located in the processor, on special interface cards, and/or on individual device-controller cards.

Refer to Fig. 5-10*a*. If the interrupt is not disabled by the mask flip-flop or by the PRIORITY IN line, a service request (device-flag level) will set the REQUEST flip-flop, which is clocked by periodic processor pulses (I/O SYNC) to fit the processor cycle and to time the priority decision. The resulting timed PRIORITY REQUEST step has *three* jobs:

1. It preenables the "ACTIVE" flip-flop belonging to the same interrupt circuit.
2. It blocks lower-priority interrupts.
3. It informs the processor that an interrupt is wanted.

If the interrupt system is on (and if there are no direct-memory-access requests pending, Sec. 5-17), the processor answers with an INTERRUPT ACKNOWLEDGE pulse just before the current instruction is completed (Fig. 5-13). This sets the preenabled "ACTIVE" flip-flop, which now gates the correct trap address onto a set of bus lines—the interrupt is active. INTERRUPT ACKNOWLEDGE also resets *all* REQUEST flip-flops to ready them for repeated or new priority requests.

Each interrupt has three states: inactive, waiting (device-flag flip-flop set), and active. Waiting interrupts will be serviced as soon as possible. Unless reset by program or hardware, the device flag maintains the "waiting" state

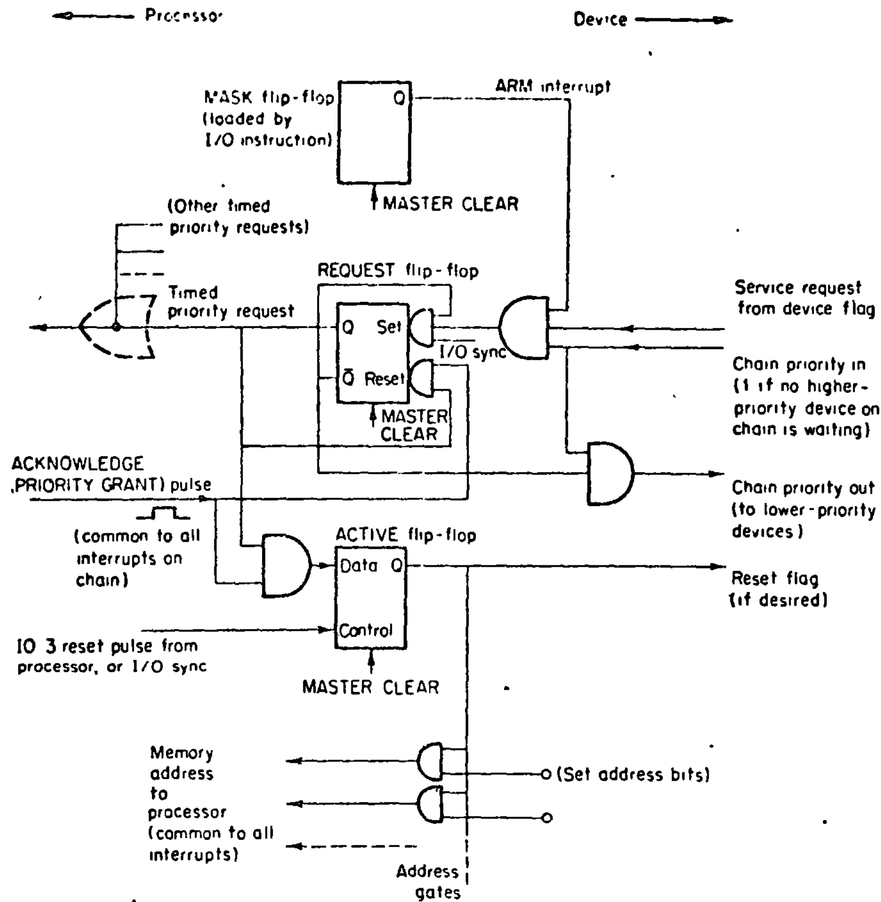


Fig. 5-10*a*. Priority-chain timing/queuing logic for one device (see also the timing diagram of Fig 5-12) The ACKNOWLEDGE line is common to all interrupts on the chain Note how the flip-flops are timed by the processor-supplied I/O SYNC pulses MASTER CLEAR is issued by the processor whenever power is turned on, and through a console pushbutton, to reset flip-flops initially Many different modifications of this circuit exist (see also Fig. 5-11). Similar logic is used for direct-memory-access requests

while higher-priority service routines run and even while its interrupt is disarmed or while the entire interrupt system is turned off

5-14. Priority Propagation and Priority Changes. There are two basic methods for suppressing lower-priority interrupts. The first is the **wired-priority-chain** method illustrated in Fig 5-10. Referring to Fig. 5-10*a*, the PRIORITY IN terminal of the lowest-priority device is wired to the PRIORITY OUT terminal of the device with the next-higher priority, and so on. Thus the timed requests from higher-priority devices block lower-priority requests. The PRIORITY IN terminal of the highest-priority

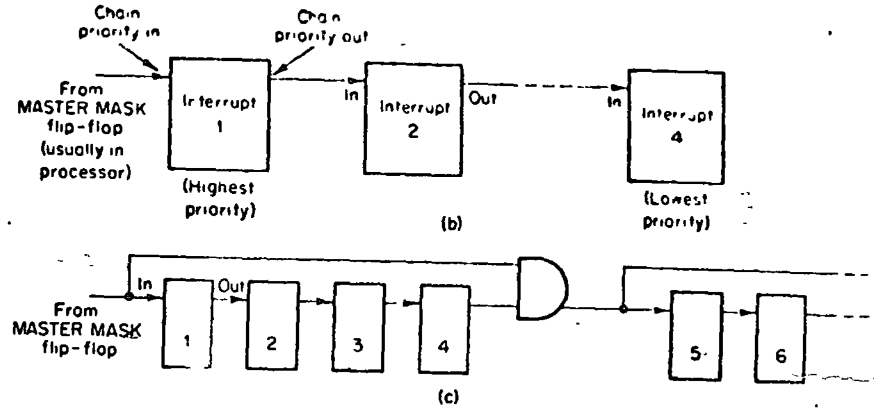


Fig. 5-10*b* and *c*. Wired-chain priority-propagation circuits. Since each subsystem (and its associated wiring) delays the propagated REQUEST flip-flop steps (Fig. 5-10*a*) by 10 to 30 nsec, the simple chain of Fig 5-10*b* should not have more than four to six links, the circuit of Fig. 5-10*c* bypasses priority-inhibiting steps for faster propagation (*based on Ref.* 10).

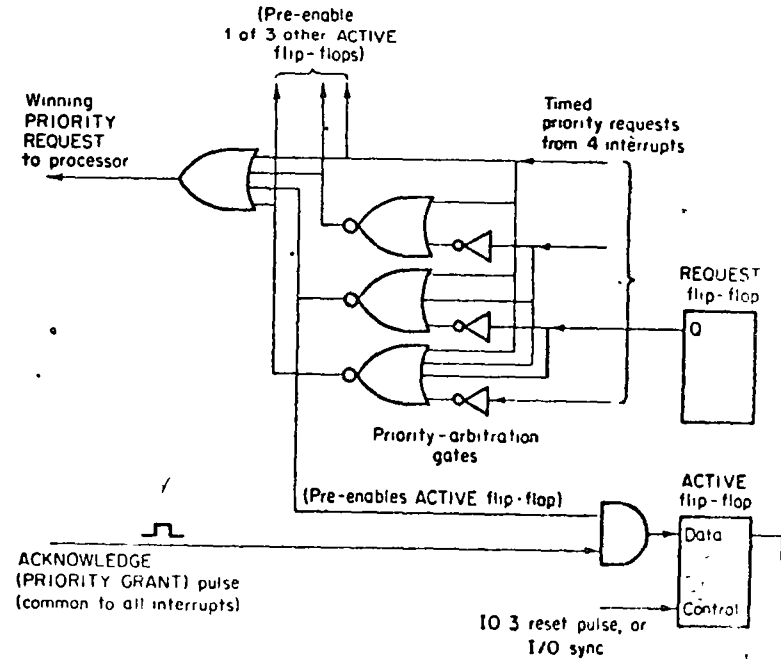


Fig. 5-11. This modified version of the priority-interrupt logic in Fig 5-10*a* has priority-propagation gates at the output rather than at the input of the REQUEST flip-flop Again, many similar circuits exist.

device (usually a power-failure, parity-error, or real-time-clock interrupt in the processor itself) connects to a processor flip-flop ("master-mask" flip-flop), which can thus arm or disarm the entire chain (Fig. 5-10*b* and *c*).

The computer program can load mask-register flip-flops (Fig. 5-10*a*) to *disarm* selected interrupts in such a wired chain, but the relative priorities of all armed interrupts are determined by their positions in the chain. It is possible, though, to assign two or more different priorities to a given device flag: we connect it to two or more separate priority circuits in the chain and arm one of them under program or device control.

Figure 5-11 illustrates the second type of priority-propagation logic, which permits every armed interrupt to set its REQUEST flip-flop. The timed PRIORITY REQUEST steps from different interrupts are combined in a "priority-arbitration" gate circuit, which lets only the highest-priority REQUEST step pass to preenable its "ACTIVE" flip-flop. Some larger digital computers implement dynamic priority reallocation by modifying their priority-arbitration logic under program control, but most minicomputers are content with programmed masking.

The two priority-propagation schemes can be *combined*. Several minicomputer systems (e.g., PDP-9, PDP-15) employ four separate wired-priority chains, each armed or disarmed by a common "master-mask" flip-flop in the processor. Interrupts from the four chains are combined through a priority-arbitration network which, together with the program-controlled "master-mask" flip-flops, establishes the relative priorities of the four chains.

5-15. Complete Priority-interrupt Systems. (a) Program-controlled Address Transfer. The "ACTIVE" flip-flop in Fig. 5-10*a* or 5-11 places the starting address of the correct interrupt-service routine on a set of address lines common to all interrupts. Automatic or "hardware" priority-interrupt systems will then immediately trap to the desired address (Sec. 5-15*b*). But in many small computers (e.g., PDP-8 series, SUPERNOVA), the priority logic is only an add-on card for a basic single-level (ORed) interrupt system. Such systems cannot access different trap addresses directly. With the interrupt system on, *every* PRIORITY REQUEST disables further interrupts and causes the program to trap to *the same* memory location, say 0000, and to store the program counter, just as in Sec. 5-9. The trap location contains a jump to the service routine

```
SRVICE  STORE ACCUMULATOR IN   SAVAC   / Unless we have
                                       /   a spare
                                       /   accumulator
        READ INTERRUPT ADDRESS
        STORE ACCUMULATOR IN   PTR
        JUMP INDIRECT VIA      PTR
```


READ INTERRUPT ADDRESS is an ordinary I/O instruction, which employs a device selector to read the interrupt-address lines into the accumulator (Sec. 5-9). The IO2 pulse from the device selector can serve as the ACKNOWLEDGE pulse in Fig. 5-10*a* or 5-11 (in fact, the "ACTIVE" flip-flop can be omitted in this simple system). The program then transfers the address word to a pointer location PTR in memory, and an indirect jump lands us where we want to be.

Unfortunately, the service routine for each individual device, say for an ADC, must save and restore program counter, mask, *and* accumulator (see also Sec. 5-12):

```
ADC     LOAD ACCUMULATOR       0000
        STORE ACCUMULATOR IN   SAVPC
        LOAD ACCUMULATOR       SAVAC
        STORE ACCUMULATOR IN   SAVAC2
        LOAD ACCUMULATOR       MASK
        STORE ACCUMULATOR IN   SVMSK
        LOAD ACCUMULATOR       MASK 1
        STORE ACCUMULATOR      MASK
        LOAD MASK REGISTER
        INTERRUPT ON
        READ ADC                          / Useful work
        STORE ACCUMULATOR IN   X          /   done only here!
        INTERRUPT OFF
        LOAD ACCUMULATOR       SVMSK
        STORE ACCUMULATOR      MASK
        LOAD MASK REGISTER
        LOAD ACCUMULATOR       SAVAC 2
        INTERRUPT ON
        JUMP INDIRECT VIA      SAVPC
```

Note that most of the time and memory used up by this routine is overhead devoted to storing and saving registers.

(b) A Fully Automatic ("Hardware") Priority-interrupt System. In an automatic or "hardware" priority-interrupt system, the "ACTIVE" flip-flop in Fig. 5-10*a* or 5-11 gates the trap address of the active interrupt into the processor memory address register as soon as the current instruction is completed (Fig. 5-12). This requires special address lines in the input/output bus and a little extra processor logic. This hardware buys improved response time and simplifies programming:

1. **The program traps immediately to a different trap location for each interrupt; there is no need for the program to identify the interrupt.**
2. There is no need to save program counter and registers twice as in Secs. 5-11, 5-12, and 5-15*a*.

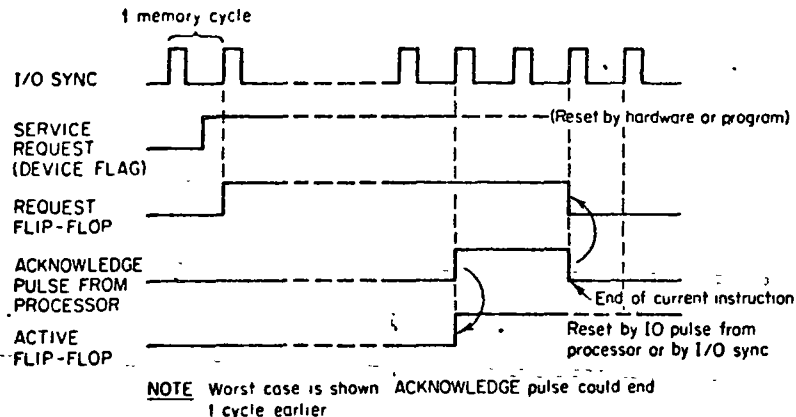


Fig. 5-12. Timing diagram for the priority-interrupt logic of Figs 5-10 and 5-11. The ACKNOWLEDGE pulse remains ON until the trap address is transferred (either immediately over special address lines or by a programmed instruction)

In a typical system, each hardware-designated trap location is loaded with a modified JUMP AND SAVE instruction (Sec. 2-11). Its effective address, say SRVICE, will store the interrupt return address (plus some status bits); this is followed by the interrupt-service routine, which can be relocatable:

```
SRVICE  XXXX                                / Incremented program-
                                            /   counter reading
                                            /   (return address)
                                            /   saved here
        STORE ACCUMULATOR IN    SAVAC       / Save accumulator

        LOAD ACCUMULATOR        MASK        / Save current
        STORE ACCUMULATOR IN    SVMSK       /   mask
        LOAD ACCUMULATOR        MASK 1      / Get
        STORE ACCUMULATOR IN    MASK        /   new
        LOAD MASK REGISTER                  /   mask
        INTERRUPT ON
        READ ADC                            / Actual work begins here
          .
          .
          .
```

Saving (and later restoring) the interrupt mask in this program is the same as in Secs. 5-12 and 5-15*a* and is seen to be quite a cumbersome operation. A little extra processor hardware can simplify this job:

1. We can combine the LOAD MASK REGISTER and INTERRUPTION instructions into a single I/O instruction.

2. We can use only masks disarming *all* interrupts with priorities *below* level 1, 2, 3, ... Such simple masks are easier to store automatically.

In the more sophisticated interrupt systems, the interrupt return-jump instruction is replaced by a special instruction (RETURN FROM INTERRUPT), which automatically restores the program-counter reading and all automatically saved registers. Be sure to consult the interface manual for your own minicomputer to determine which hardware features and software techniques are available.

5-16. Discussion of Interrupt-system Features and Applications. Interrupts are the basic mechanism for sharing a digital computer between different, often time-critical, tasks. The practical effectiveness of a minicomputer interrupt system will depend on:

1. The time needed to service possibly critical situations
2. The total time and program overhead imposed by saving, restoring, and masking operations associated with interrupts
3. The number of priority levels needed versus the number which can be readily implemented
4. Programming flexibility and convenience

The minimum time needed to obtain service will include:

1. The "raw" latency time, i.e., the time needed to complete the longest possible processor instruction (including any indirect addressing); most minicomputers are also designed so that the processor will always execute the instruction following any I/O READ or SENSE/SKIP instruction. We are sure you will be able to tell why! Check your interface manual.
2. The time needed for any necessary saving and/or masking operation.

A look at the interrupt-service programs of Secs. 5-11, 5-12, 5-15*a*, and 5-15*b* will illustrate how successively more sophisticated priority-interrupt systems provide faster service with less overhead. You should, however, take a hard-nosed attitude to establish whether you really need the more advanced features in your specific application.

It is useful at this point to list the principal applications of interrupts. Many interrupts are associated with I/O routines for relatively slow devices such as teletypewriters and tape reader/punches, and thousands of minicomputers service these happily with simple skip-chain systems. Things become more critical in instrumentation and control systems, which must not miss real-time-clock interrupts intended to log time, to read instruments, or to perform control operations. Time-critical jobs require *fast responses*. If there are many time-critical operations or any time-sharing computations,

the computing time wasted in overhead operations becomes interesting. Some real-time systems may have periods of peak loads when it becomes actually impossible to service *all* interrupt requests. At this point, the designer must decide whether to buy an improved system or which interrupt requests are at least temporarily expendable. It is in the latter connection that *dynamic priority allocation* becomes useful: it may, for instance, be expedient *to mask certain interrupts during peak-load periods*. In other situations we might, instead, *lower the relative priority of the main computer program by unmasking additional interrupts during peak real-time loads*.

If two or more interrupt-service routines employ the same library subroutine, we are faced, as in Sec. 4-16, with the problem of *reentrant programming*. Temporary-storage locations used by the common subroutine may be wiped out unless we either duplicate the subroutine program in memory for each interrupt or unless we provide true reentrant subroutines. This is not usually the case for FORTRAN-compiler-supplied library routines. Only a few minicomputer manufacturers and software houses provide reentrant FORTRAN (sometimes called "real-time" FORTRAN). The best way to store saved registers and temporary intermediate results is in a stack (Sec. 4-16); a stack pointer is advanced whenever a new interrupt is recognized and retracted when an interrupt is dismissed. *The best minicomputer interrupt systems have hardware for automatically advancing and retracting such a stack pointer* (Sec 6-10).

If very fast interrupt service is not a paramount consideration, *we can get around reentrant coding by programming interrupt masks which simply prevent interruption of critical service routines*.

In conclusion, remember that the chief purpose of interrupt systems is to initiate computer operations more complicated than simple data transfers. The best method for time-critical reading and writing as such is not through interrupt-service routines with their awkward programming overhead but with a *direct-memory-access system*, which has no such problems at all.

DIRECT MEMORY ACCESS AND AUTOMATIC BLOCK TRANSFERS

5-17. Cycle Stealing. Step-by-step program-controlled data transfers limit data-transmission rates and use valuable processor time for alternate instruction fetches and execution; programming is also tedious. It is often preferable to use additional hardware for interfacing a parallel data bus directly with the digital-computer memory data register and to request and grant 1-cycle pauses in processor operation for direct transfer of data to or from memory (interlace or cycle-stealing operation). In larger digital machines, and optionally in a few minicomputers (PDP-15), a data bus can even access one memory bank without stopping processor interaction with other memory banks at all.

Note that cycle stealing in no way disturbs the program sequence. Even though smaller digital computers must stop computation during memory transfers, the program simply skips a cycle at the end of the current memory cycle (no need to complete the current *instruction*) and later resumes just where it left off. One does not have to save register contents or other information, as with program interrupts.

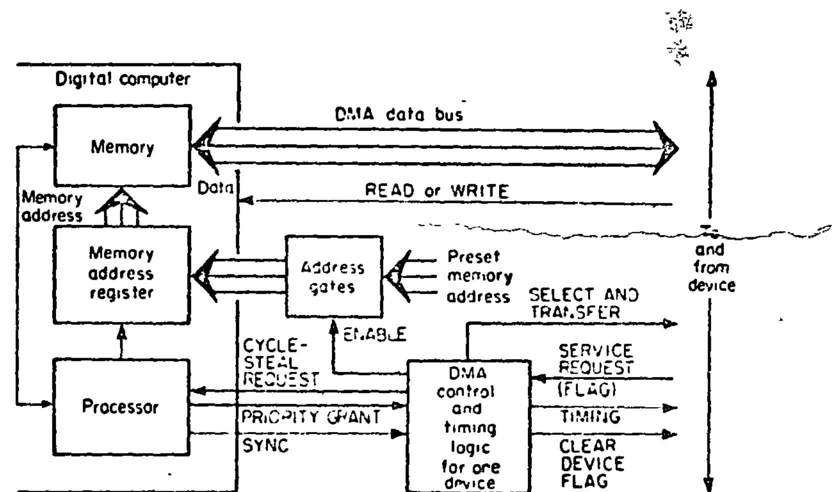


Fig. 5-13. A direct-memory-access (DMA) interface.

5-18. DMA Interface Logic. To make direct memory access (DMA) practical, the interface must be able to:

1. Address desired locations in memory
2. Synchronize cycle stealing with processor operation
3. Initiate transfers by device requests (this includes clock-timed transfers) or by the computer program
4. Deal with priorities and queuing of service requests if two or more devices request data transfers

DMA priority/queuing logic is essentially the same as the priority-interrupt logic of Figs. 5-10 and 5-11; indeed, identical logic cards often serve both purposes. DMA service requests are always given priority over concurrent *interrupt requests*.

Just as in Fig. 5-11, a DMA service request (caused by a device-flag level) produces a cycle-steal request unless it is inhibited by a higher-priority request; the processor answers with an acknowledge (priority-grant) pulse. This signal then sets a processor-clocked "ACTIVE" flip-flop, which strobes a suitable memory address into the processor memory address register and then causes memory and device logic to transfer data from or to the DMA data bus (Fig. 5-13).

In some computer systems (e.g., Digital Equipment Corporation PDP-15), the DMA data lines are identical with the programmed-transfer data lines. This simplifies interconnections at the expense of processor hardware. In other systems, the DMA data lines are also used to transmit the DMA address to the processor before data are transferred. This further reduces the number of bus lines, but complicates hardware and timing.

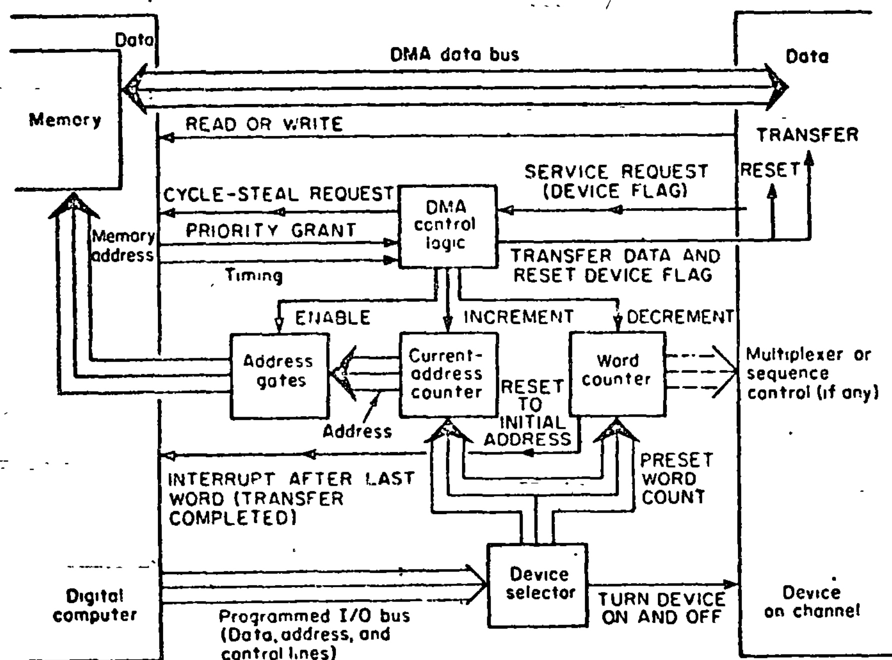


Fig. 5-14. A simple data channel for automatic block transfers.

5-19. Automatic Block Transfers. As we described it, the DMA data transfer is *device*-initiated. A *program-dependent* decision to transfer data, even directly from or to memory, still requires a programmed instruction to cause a DMA service request. This is hardly worth the trouble for a *single-word* transfer. Most DMA transfers, whether device or program initiated, involve not single words but blocks of tens, hundreds, or even thousands of data words.

Figure 5-14 shows how the simple DMA system of Fig. 5-13 may be expanded into an automatic data channel for block transfers. Data for a block can arrive or depart asynchronously, and the DMA controller will steal cycles as needed and permit the program to go on between cycles. A block of words to be transferred will, in general, occupy a corresponding block of adjacent memory registers. Successive memory addresses can be

gated into the memory address registered by a counter, the **current-address counter.** Before any data transfer takes place, a programmed instruction sets the current-address counter to the desired initial address; the desired number of words (block length) is set into a second counter, the **word counter,** which will count down with each data transfer until 0 is reached after the desired number of transfers. As service requests arrive from, say, an analog-to-digital converter or data link, the DMA control logic implements successive cycle-steal requests and gates successive current addresses into the memory address register as the current-address counter counts up (see also Fig. 5-5*a*).

The word counter is similarly decremented once per data word. When a block transfer is completed, the word counter can stop the device from requesting further data transfers. The word-counter carry pulse can also cause an *interrupt* so that a new block of data can be processed. The word counter may, if desired, also serve for sequencing device functions (e.g., for selecting successive ADC multiplexer addresses).

Some computers replace the word counter with a program-loaded final-address register, whose contents are compared with the current-address counter to determine the end of the block.

A DMA system often involves several data channels, each with a DMA control, address gates, a current-address counter, and a word counter, with different priorities assigned to different channels. For efficient handling of randomly timed requests from multiple devices (and to prevent loss of data words), data-channel systems may incorporate buffer registers in the interface or in devices such as ADCs or DACs.

5-20. Advantages of DMA Systems (see Ref. 6). Direct-memory-access systems can transfer data blocks at very high rates (10^6 words/sec is readily possible) without elaborate I/O programming. The processor essentially deals mainly with buffer areas in its own memory, and only a few I/O instructions are needed to initialize or reinitialize transfers.

Automatic data channels are especially suitable for servicing peripherals with high data rates, such as disks, drums, and fast ADCs and DACs. But fast data transfer with minimal program overhead is extremely valuable in many other applications, especially if there are many devices to be serviced. To indicate the remarkable efficiency of cycle-stealing direct memory access with multiple block-transfer data channels, consider the operation of a training-type digital flight simulator, which solves aircraft and engine equations and services an elaborate cockpit mock-up with many controls and instrument displays. During each 160-msec time increment, the interface not only performs 174 analog-to-digital conversions requiring a total conversion time of 7.7 msec but also 430 digital-to-analog conversions, and handles 540 eight-bit bytes of discrete control information. The actual

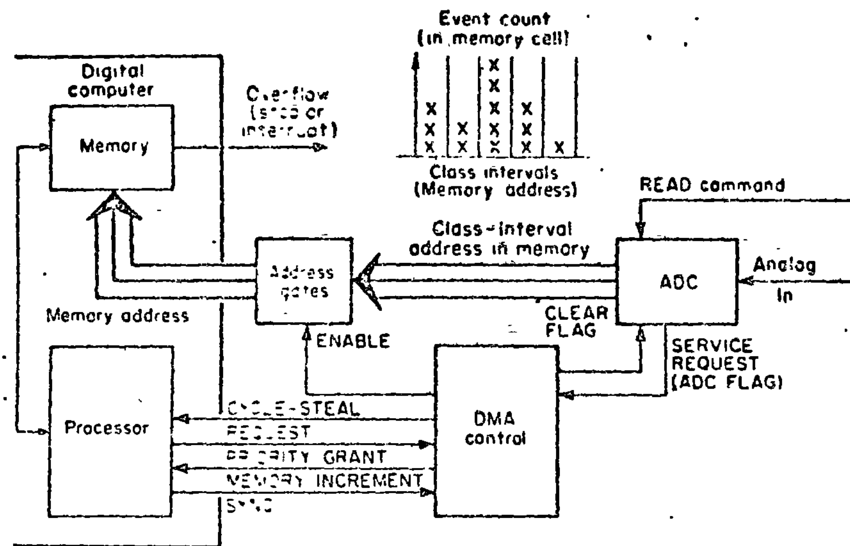


Fig. 5-15a. Memory-increment technique of measuring amplitude distributions (*based on Ref. 6*).

time required to transfer all this information in and out of the data channels is 143 msec per time increment, but because of the fast direct memory transfers, cycle-stealing subtracts only 3.2 msec for each 160 msec of processor time (Ref. 2).

5-21. Memory-increment Technique for Amplitude-distribution Measurements. In many minicomputers, a special pulse input will *increment* the contents of a memory location addressed by the DMA address lines; an interrupt can be generated when one of the memory cells is full. When ADC outputs representing successive samples of a random voltage are applied to the DMA address lines, the memory-increment feature will effectively generate a model of the input-voltage amplitude distribution in the computer

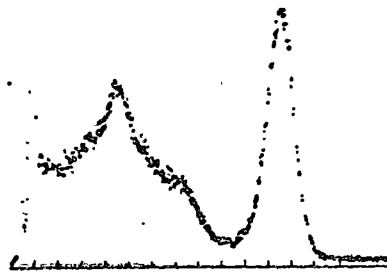


Fig. 5-15b. An amplitude-distribution display obtained by the method of Fig 5-15a. (*Digital Equipment Corporation*)

memory: Each memory address corresponds to a voltage class interval, and the contents of the memory register represent the number of samples falling into that class interval. Data taking is terminated after a preset number of samples or when the first memory register overloads (Fig. 5-15*a*). The empirical amplitude distribution thus created in memory may be displayed or plotted by a display routine (Fig. 5-15*b*), and statistics such as

$$\bar{X} = \frac{1}{n}\sum_{k=1}^{n} X_k \qquad \overline{X^2} = \frac{1}{n}\sum_{k=1}^{n} X_k^2 \qquad \cdots$$

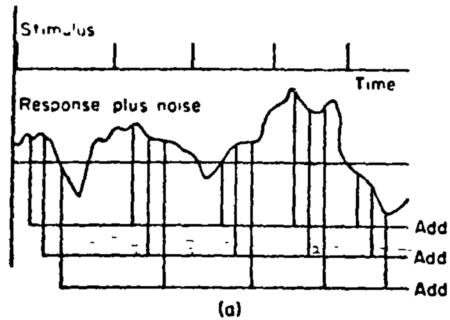
are readily computed after the distribution is complete. This technique has been extensively applied to the analysis of pulse-energy spectra from nuclear-physics experiments.

Joint distributions of two random variables X, Y can be similarly compiled. It is only necessary to apply, say, a 12-bit word X, Y composed of two 6-bit bytes corresponding to two ADC outputs X and Y to the memory address register. Now each addressed memory location will correspond to the region $X_i \le X < X_{i+1}$, $Y_k \le Y < Y_{k+1}$ in XY space.

5-22. Add-to-memory Technique of Signal Averaging. Another command-pulse input to some DMA interfaces will *add* a data word on the I/O-bus data lines to the memory location addressed by the DMA address lines without ever bothering the digital-computer arithmetic unit or the program. This "add-to-memory" feature permits useful linear operations on data obtained from various instruments; the only application well known at this time is in data averaging.

Figure 5-16*a* and *b* illustrates an especially interesting application of data averaging, which has been very fruitful in biological-data reduction (e.g., electroencephalogram analysis). Periodically applied stimuli produce the same system response after each stimulus so that one obtains an analog waveform periodic with the period T of the applied stimuli. To pull the desired function $X(t)$ out of additive zero-mean random noise, one adds $X(t)$, $X(t + T)$, $X(t + 2T)$, . . . during successive periods to enhance the signal, while the noise will tend to average out. Figure 5-16*c* shows the extraction of a signal from additive noise in successive data-averaging runs.

5-23. Implementing Current-address and Word Counters in the Processor Memory. Some minicomputers (in particular, PDP-9, PDP-15, and the PDP-8 series) have, in addition to their regular DMA facilities, a set of fixed core-memory locations to be used as data-channel address and word counters. Ordinary processor instructions (not I/O instructions) load these locations, respectively, with the block starting address and with minus the block count. The data-channel interface card (Fig. 5-17) supplies the address of one of the four to eight address-counter locations available in the processor; the word counter is the location following the address counter.



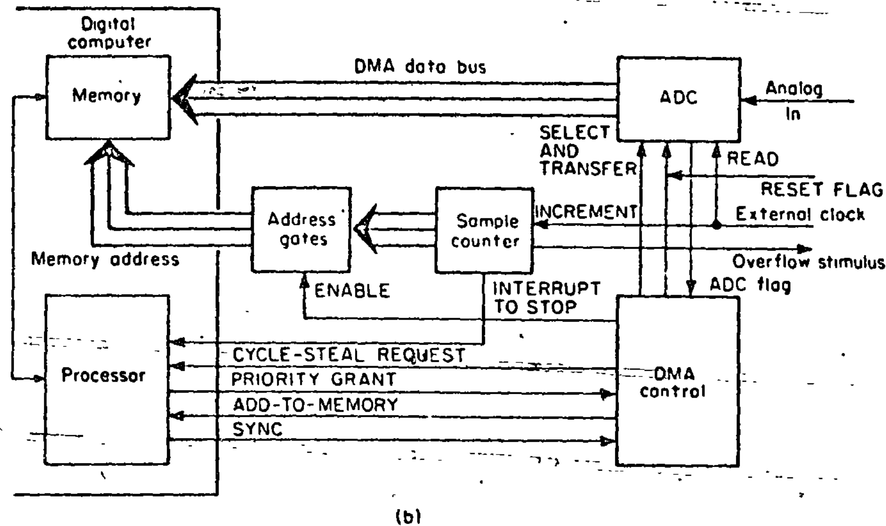
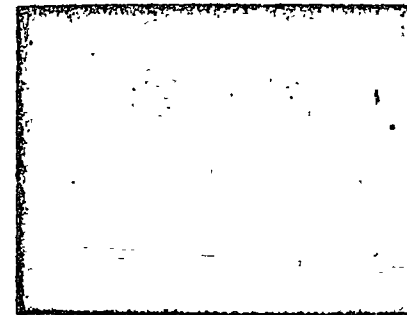


Fig. 5-16a and b. Signal enhancement by periodic averaging (*a*) and add-to-memory technique for signal averaging (*b*) (*based on Ref. 6*).

Now, successive service requests steal not one but three or four cycles since the processor must increment the two counter locations, and they then transfer data to or from successive memory cells indirectly addressed via the address counter. When the word counter reaches 0 (from its negative initial setting), the processor issues a special signal which is used to stop further service requests and usually to interrupt the processor (Fig. 5-17).

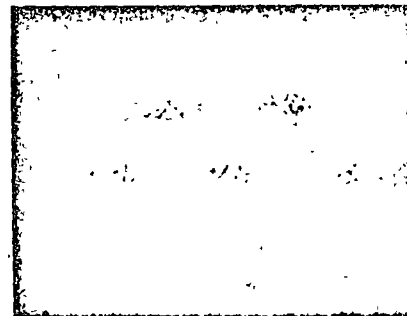
Some memory-implemented data channels will also permit add-to-memory operation (PDP-9, PDP-15). The Honeywell 316/516 machines implement a final-address register in memory rather than a word counter (Sec. 5-19) and permit automatic alternation of data transfers to or from two blocks of memory locations (*swinging buffers*).

Memory-implemented data channels permit automatic block transfers with a minimum of interface hardware since they eliminate the two external counter/registers pl[illegible] the circuits needed to preset them. On the other hand, logic

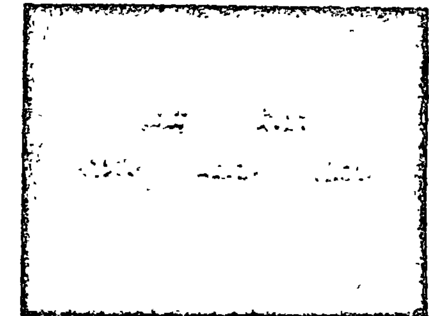


8 repetitions

32 repetitions



128 repetitions



512 repetitions

Fig. 5-16c. A periodically retriggered waveform extracted from additive noise through successive signal-averaging runs. The display is self-scaling, i.e., ordinate words are shifted right by 1 bit after 2, 4, 8, . . . repetitions to divide accumulated sums by the number of repetitions. (*University of Arizona, PDP-9 data taken by H. M. Aus.*)

circuits are becoming more and more inexpensive, and true data channels steal only one-third to one-fourth as much processor time as memory-implemented channels.

SOME INTERFACE-HARDWARE CONSIDERATIONS

5-24. From Ready-made to Do-it-yourself Interfaces. Device controllers for typical peripheral devices have many common features, so several minicomputer manufacturers sell standard interface cards. Typical device-controller cards implement a device selector, bus gates, a register, and/or some device-flag flip-flops and sense gates. The same card or a second card may comprise interrupt logic or a data-channel controller. Some minicomputer main-frames (e.g., Hewlett-Packard, Data General) have

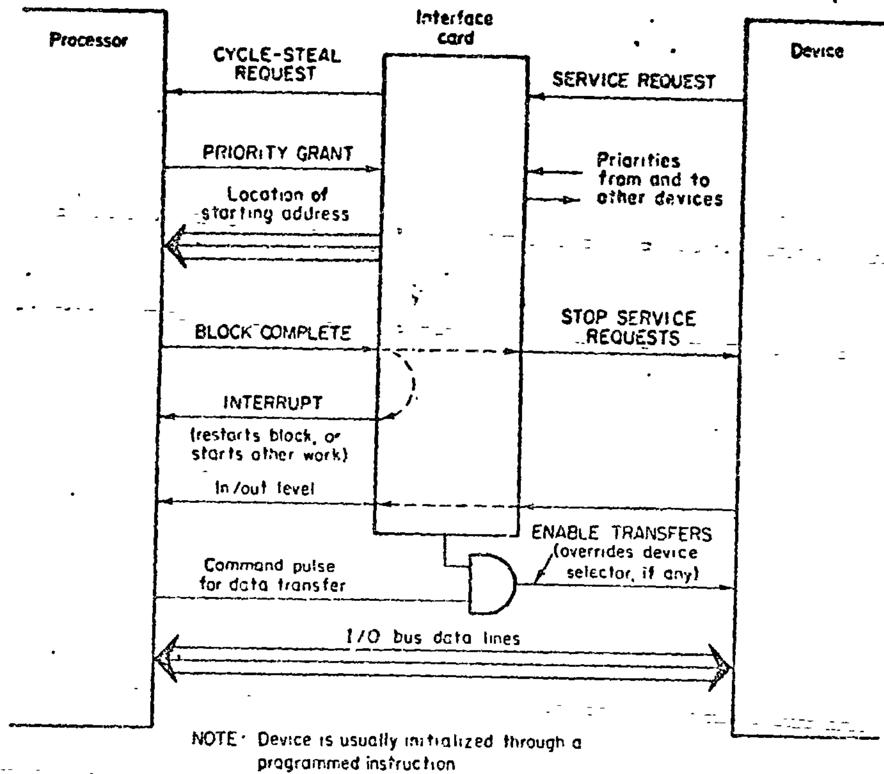


Fig. 5-17. With data-channel address and word counters implemented in the processor memory, data-channel operation requires only a simple interface card containing priority queuing and address logic like that in Fig. 5-10, but each data transfer steals three or four processor cycles, not just one cycle

special slots for interface cards. In other systems, a substantial portion of the interface logic is physically associated with each device.

Interface logic, all the way from simple data-transfer and sensing logic to new and special controllers, displays, and accessory arithmetic units, is remarkably easy to make from commercially available logic cards and/or socket-mounted integrated circuits. Several manufacturers sell logic and related analog/digital circuit cards (ADCs, DACs, electronic switches, amplifiers, power supplies), plus very convenient mounting hardware, enclosures, panel switches, indicators, etc. (Fig. 5-18). Transistor/transistor logic (TTL) interfaces naturally with most minicomputers, but *high-noise-immunity logic* (Motorola HTL, Digital Equipment Corporation K-series cards) should be considered for high-noise environments; these circuits have larger logic-level swings and are intentionally slower to reduce the possibility of random-noise triggering.

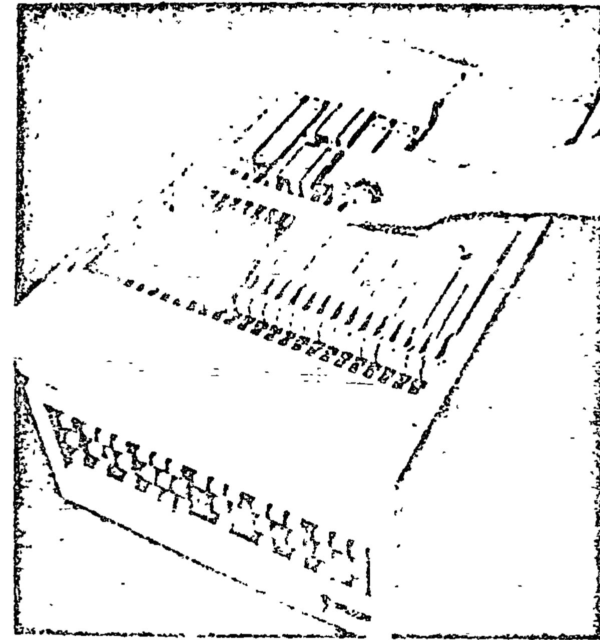
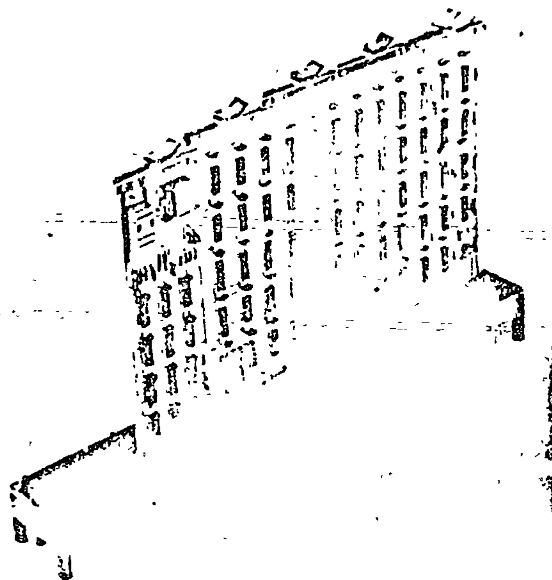


Fig. 5-18a Much of the interface logic in this Hewlett-Packard 2100A system is on manufacturer-supplied interface cards plugged into the processor chassis. *(Hewlett-Packard Corporation.)*

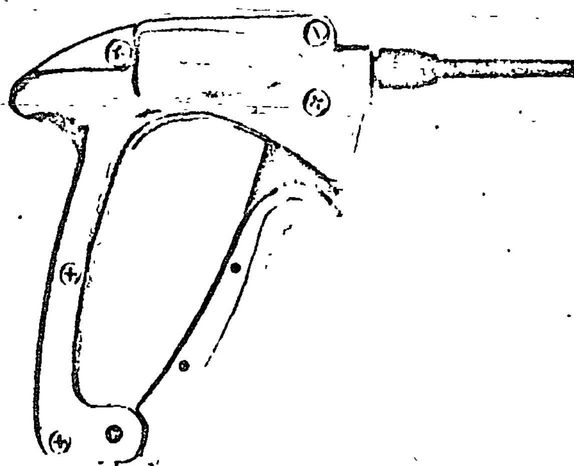
We recommend wire wrapping with a simple "squeeze gun" (Fig. 5-18c), rather than soldering, for convenient and reliable connections except for fast emitter-coupled logic (ECL) and radio-frequency circuits. A very little knowledge of digital-logic design goes a long way (Table 1-5), but if you have problems, we suggest that you hire a graduate student from the nearest electrical engineering department part time.

5-25. I/O-bus Lines and Signals. Party-line I/O buses (Sec. 5-1) are usually "daisy-chained" from device to device through male/female connector pairs, with a suitable line termination plugged into the last female connector.

Figure 5-19 shows some typical bus circuits. Most minicomputer TTL interfaces employ open-collector integrated circuits (gates or amplifiers) as line drivers (Fig. 5-19*a*). Off-the-shelf ICs not specifically designed as line drivers may require testing for voltage levels and rise times. Ordinary gates used as line receivers may also have to be tested for safe logic-level thresholds. It is surely best to employ special drivers and receivers (perhaps even push-pull drivers and receivers, Sec. 5-26), but this may not be necessary for short buses. If in doubt, use special cards supplied by the computer manufacturer.



(b)



(c)

Fig. 5-18b and c. Logic cards, receptacles, and a Gardner-Denver wire-wrap tool for assembling homemade minicomputer interface systems. (*Digital Equipment Corporation.*)

With wiring delays (at least 1.5 nsec/ft) of the same order as logic rise times, transmission-line reflections must be considered (Fig. 5-20). If the power to each device on a daisy-chained bus can be turned off separately, the interface designer must r e sure that this will not affect proper operation of other devices (see also gs. 5-19 and 5-20).

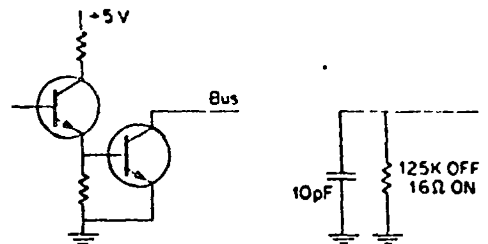


Fig. 5-19a. Output stages of an open-collector, inverting TTL bus driver and its equivalent source impedance.

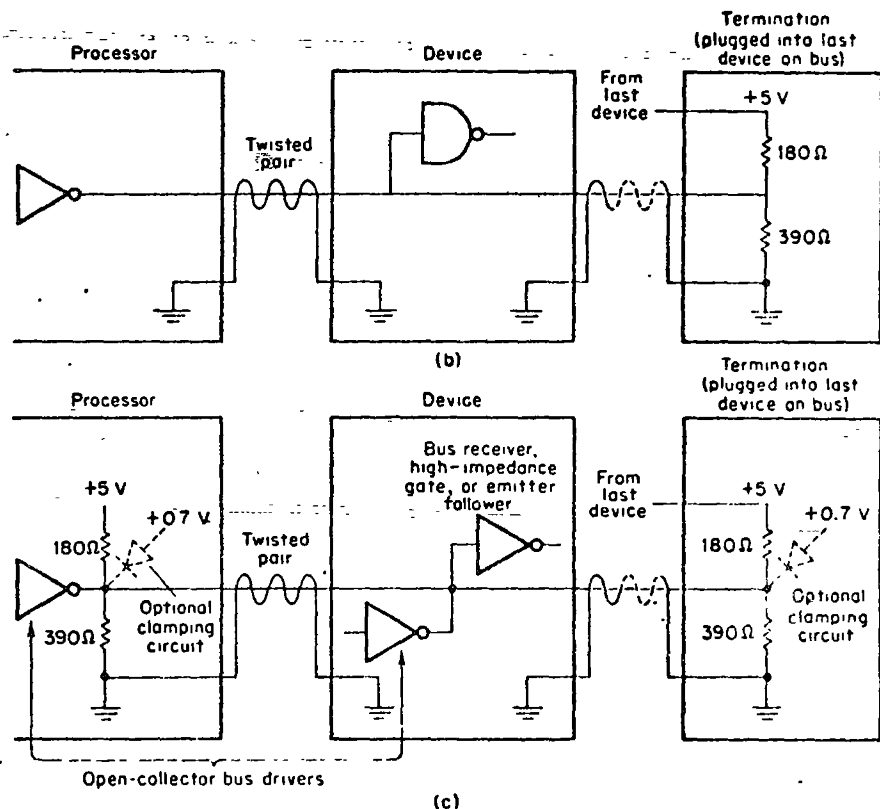


Fig. 5-19b and c. A *unidirectional processor-to-device bus* (b) and a *bidirectional bus* (c) each without differential line receivers. *Unidirectional device-to-processor* bus lines would also be terminated at both ends as in Fig. 5-19c.

Use lines whose characteristic impedance Z_0 is at least 90 ohms (93-ohm coax or 100-ohm No. 26 or 28 twisted pair, about 30 turns/ft), with ground return. Flat cable, with signal conductors separated by ground returns, and possibly with a shielding backplane, is very convenient, especially for short (below 1 to 3 ft) cable runs. A diode or Schottky-diode reverse termination at the output end will limit negative overshoot; follow the

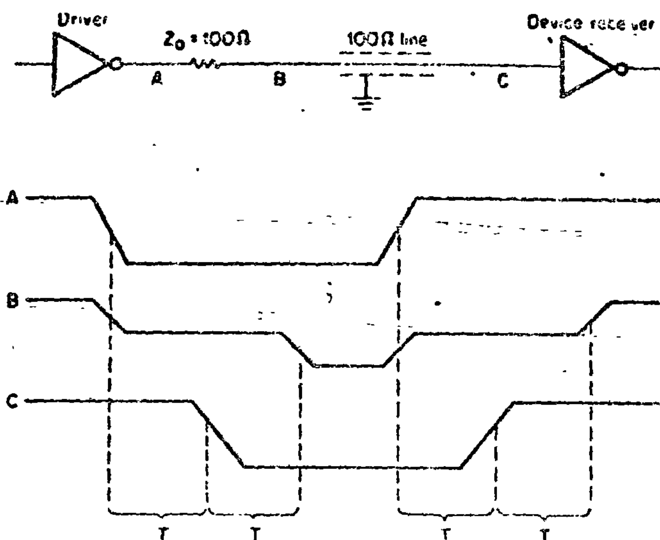


Fig. 5-20. A series-terminated transmission line. The effect on the driving circuit does not change when the receiver power supply is turned off. *T* is the line-propagation delay.

computer manufacturer's recommendations. Do *not* use flip-flops as line drivers.

I/O-bus logic levels are *not* usually the same as standard logic levels but will depend on the bus system and loading. Note in particular that many minicomputer I/O buses are driven by amplifiers or gates which **invert logic** levels (e.g., Fig. 5-19*a*), i.e., bus signals corresponding to logical 1 may be LOW voltage ("ACTIVE LOW" signals). **Check your interface manual,** which will also specify the load each bus line can drive under various conditions.

5-26. Noise, Interconnections, and Ground Systems. Digital-computer interfaces are very often connected to sensitive and accurate analog instrumentation and computing circuits. Digital-system noise, especially high-frequency spikes and pulse ringing, can cause very objectionable noise in analog circuits via ground currents and radiation. This can be true even though the digital circuits themselves work well within their noise-immunity limits. Transistor/transistor logic (TTL) and diode/transistor logic (DTL), with their harsh ground-current transients and relatively high output impedances, are bad offenders in this respect. Emitter-coupled logic (ECL) has near-constant ground current, low logic levels, and low output impedances, and it is a good choice for critical wide-band analog/digital circuits. Digitally controlled analog switching circuits (digital-to-analog converters, sample-hold circuits, multiplexers) should be designed to minimize impedances common to digital and analog signals.

Ground-system noise and common ground impedances can cause serious

problems. A good earth ground is not always easy to come by, and the power-line "industrial" ground should be used for ac return only. To minimize common ground impedances within a cabinet or subsystem, it is best to select a **single common ground point** and to return all signal, power, and chassis grounds separately to this point, which is also connected to earth ground.

Unfortunately, this simple technique may not work when we must ground widely separated subsystems interconnected by signal lines (e.g., a digital

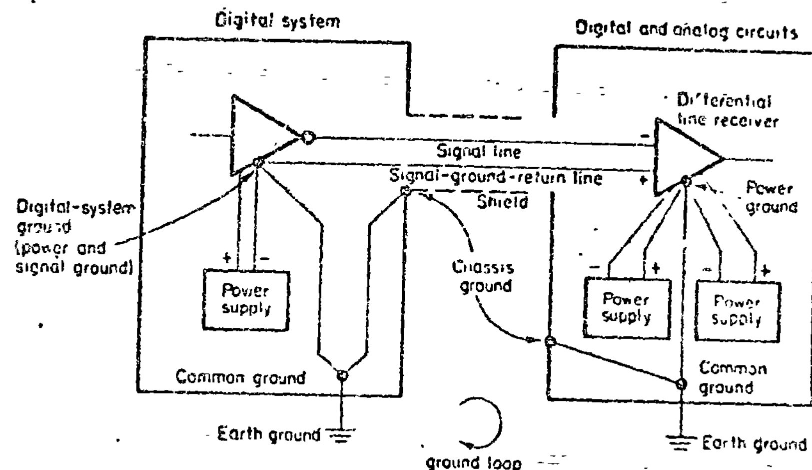


Fig. 5-21. Interconnection and grounding of two typical subsystems. Differential line receivers cancel common-mode ground-loop and other disturbances. An even better method is to use push-pull line receivers and line drivers without any ground return between subsystems. Radiation from digital circuits is still a problem (see also Table 5-2). Earth-ground connections, required for electrical safety, may need shielding. Power-supply ac inputs may need radio-frequency filters, and power transformers may need electrostatic shields.

computer and an analog subsystem 40 ft away). We will then give each subsystem a common ground point and try to keep all power-supply loops within each subsystem. But if each subsystem has an earth ground (often required for electrical safety), then we have a **ground loop** enclosed by the ground connections and each signal wire. Such inductive loops will pick up and/or radiate noise (Fig. 5-21). The best way to fix this is to use differential (push-pull) signal transmission or at least differential signal receivers, which will cancel ground-loop noise and other disturbances common to both differential inputs (Fig. 5-21; see also Table 5-1).

If electrical safety is no problem, we may omit one earth ground and interconnect the subsystem grounds through a ground return line in the signal cable. Signal-cable shields must *never* carry ground currents; they ordinarily connect to chassis ground at the source end. Even so, we may still have a ground loop formed through leakage resistances and capacitances.

etc. (even DMA block transfers are usually terminated and reinitiated by interrupts)

2. These interrupt-service programs must also manage assignment and reassignment of buffer areas so as to avoid interference between data-processing and I/O operations.
3. Numerical-character I/O for numerical computations is necessarily associated with formatting routines relating numerical input or output data to binary fixed-point or floating-point number representations used in the computer. Character packing/unpacking (Sec. 4-11) is in a similar category. Formatting and packing/unpacking are not themselves I/O operations but are often combined with I/O.
4. In addition to this, input/output programs often include error routines which advise the programmer of incorrect I/O requests (e.g., calls to devices which do not exist or are assigned to other jobs), parity errors, etc.

As a result of these requirements, input/output programming always involves an ugly amount of tedious (but important) detail, which has little to do with the computer application as such. To relieve the computer user (who would like to concentrate on his applications programs), computer manufacturers furnish "canned" I/O and formatting routines and special system programs which make it easy to call I/O-related routines from user programs.

5-28. Use of FORTRAN Formatting and I/O. Most readers will be familiar with FORTRAN formatting and I/O statements like

```
12    FORMAT    (E 10.4)
      READ      (2, 12)   X
```

where E 10.4 calls for a floating-point conversion, 2 is the number of the peripheral device to be read, and 12 refers to the associated FORMAT statement. *Unformatted* READ or WRITE statements like

```
WRITE (7) X, Y, Z
```

call for input or output of the listed quantities in *binary* form (e.g., for ADCs and DACs).

Minicomputer FORTRAN compilers recognize such statements and automatically generate the appropriate formatting and I/O routines without further effort on the part of the programmer. With programs written in *assembly language*, the easiest way to produce formatted numerical input/output is still to link the assembly-language program to a short FORTRAN program for FORTRAN I/O (Sec. 4-20). The time consumed by the (unseen, but quite formidable) FORTRAN-generated formatting and I/O routines will rarely bother you with slow keyboard printer I/O, although you might notice the time lost with alphanumeric cathode-ray-tube displays supervising real-time computations.

5-29. Interpreter-program Formatting and I/O. Interpreter systems (e.g., BASIC and FOCAL, Sec. 3-8) include formatting and I/O commands used much like the corresponding FORTRAN statements, plus special commands for simple cathode-ray-tube and plotter graphics output. Several minicomputer manufacturers supply special versions of interpreter systems such as BASIC with greatly enhanced I/O capabilities for "conversational" programming of instruments, test systems, and process controllers (Sec. 7-3). Such systems are extra convenient, but their range of applications may be restricted. Interpreters do not generate relocatable code which can be combined with other programs, and execution may be slow.

5-30. I/O at the Assembly-language Level: Device Drivers. At the assembly-language level, a complete I/O operation is called as a subroutine known as a device driver or device handler. The most important drivers implement block transfers, and any one device driver can involve all or most of the I/O-related jobs listed in Sec. 5-27, including formatting. The same peripheral device (a paper-tape punch, say) could have two, three, or more associated drivers for different jobs or formats (e.g., binary and ASCII output).

Most device drivers involve interrupt service and can, therefore, be separated into two sections. The initiator subroutine reserves buffer space in memory through pseudo instructions like

```
BUFFR  .BLOCK  SIZE  / Saves SIZE locations
                     / starting at BUFFR
```

initializes buffer pointers, and prepares a peripheral device by clearing flags, setting control registers, and checking status.

The continuator section of the device driver typically initiates data transfer (e.g., START ADC CONVERSION or ENABLE DATA CHANNEL) and returns to the main program to wait for an interrupt. The continuator section of the device driver also comprises the interrupt-service routine(s) needed for data transfer and/or its termination (as in DMA block transfers, Sec. 5-19). Both initiator and continuator routines will, in general, have exits to the main computer program and to *error printout or display subroutines* (in case we have called for a nonexistent, illegal, or unready device; if there is no buffer space left; etc.).

Frequently needed device drivers (e.g., for reading and printing a line of text, reading and writing ASCII and binary tape records, etc.) are practically always furnished by the computer manufacturer as library routines. Complete device drivers for specific systems, including some DMA drivers, will be found in each minicomputer manual. They can often be used as

TABLE 5-2. Typical Minicomputer Peripheral Devices and I/O Instructions (see also Secs 3-10, 3-14, and 7-9)

A. PAPER-TAPE READER, PAPER-TAPE PUNCH, KEYBOARD, TELEPRINTER

Each of these electromechanical character-handling devices has an 8-bit device register (character buffer), which can be read or loaded by a processor I/O instruction. But each such data transfer must wait until completion of a relatively slow electromechanical reading, punching, or printing operation is signaled by a device flag (flip-flop) via a sense or interrupt line. Typical instructions are:

1. CLEAR DEVICE FLAG/OPERATOR ON NEXT CHARACTER clears the flag and permits the device register to receive or transmit a new character through electromechanical operations (shift 8 bits into device register from keyboard *if a key is struck*, read and advance tape, punch, print)
2. READ (LOAD) DEVICE REGISTER a fast, all-electrical data transfer to or from a processor accumulator
3. SKIP ON DEVICE FLAG used to implement a skip loop as in Sec 5-8 for noninterrupt operation
4. *Two or all three of these instructions can be combined into a single I/O instruction* employing two or three IO pulses (Sec 5-3).

Special control characters control special teleprinter operations (tab, line feed, form feed, etc.).

B. MAGNETIC-TAPE TRANSPORTS (see also Sec 3-10)

Tape-transport-controller logic "repacks" 8- to 18-bit computer words into 7-bit or 9-bit tape words, adding extra parity bits as needed. Binary or character formats can be selected through processor instructions

Data transfers to and from magnetic tape are usually direct-memory-access block transfers, so the controller has a *current-address counter* and a *word counter* (Sec 5-19), which can be preset by processor instructions. Since tape keeps moving once a transfer is initiated and may transfer as many as 50,000 bytes/sec, the tape-transport controller employs double buffering (Fig. 5-4*d*) between tape and memory. Only one transport at a time transfers data, but others can wind or rewind at the same time.

The controller has a *control register* (Sec 5-4) whose control bits are set by processor instructions which implement (and possibly combine) functions like:

Select one of 4 to 16 transports	Write (read) forward
Select binary or character format	Write (read) backward
Select tape speed	Rewind
Select record density	Backspace
Write end of file	Advance to end of file

The current-address counter and the status of the control register can also be *read* by processor instructions. In addition, tape-transport flag flip-flops can be sensed or cause interrupts to detect conditions such as:

Transport not ready (no power, no tape)	End (or beginning) of tape
Transport busy	Parity or check-sum error
Word-counter overflow	

Magnetic-tape driver routines are usually supplied as part of system programs (Secs 3-12 and 5-31) and can be quite complicated since hardware and program will, respectively, check lateral and longitudinal parity (Sec. 3-10). The controller can "retry" reading or writing when a parity error occurs.

C. FIXED-HEAD DISKS AND DRUMS

A disk or drum rotates continuously. Each computer word (plus a parity bit and word-delimiting gi[illegible]its) is recorded *serially* along numbered tracks, with one read/write head to each track [illegible] memory location on a disk system (*disk address*) is specified by its *disk*

TABLE 5-2. Typical Minicomputer Peripheral Devices and I/O Instructions (see also Secs 3-10, 3-14, and 7-9) (*Continued*)

number, track number, and *segment address*. Segment addresses are prerecorded on an *address track*; each segment address is read before the corresponding words pass under the read/write heads

Data transfers (typically 50,000 to 180,000 words/sec) are DMA block transfers via a *disk buffer register*, which buffers a *shift register* implementing the parallel/serial or serial/parallel conversion. The controller has a *current-address counter* and a *word counter*, which can be preset by processor instructions (Sec 5-19). DMA word-transfer request pulses synchronized with the disk rotation are derived from another prerecorded track. All prerecorded tracks are duplicated in case one gets damaged.

A typical disk may have 128 data tracks, with 2,048 word locations and an end-of-track gap on each track. Word sequences can be written or read consecutively along a track, switching to the next track when the end-of-track gap is reached ("spiral" writing or reading). If a slower data-transfer rate is desirable, the controller can also read every other word, or every fourth word, along each track, so that words are effectively interleaved.

To initiate a data transfer, programmed instructions set a *disk control register* to WRITE or READ and preset the *current-address register*, the *word counter*, and the *disk address register*. The latter (which may be a two-word register) specifies the disk, track and segment address for the first word of a data block. When the segment address matches that read on the address track, the disk controller initiates the block transfer, which stops when the word counter runs out; this will also cause a program interrupt (Sec 5-19)

Besides WRITE and READ, disk systems have a CHECK (COMPARE, SEARCH) mode which *compares* a word set into the disk buffer register with the word currently read from the disk into the shift register and which sets an interrupt flag when the words agree. This mode is used for checking purposes, and can also find words on the disk.

Other interrupt flags detect *parity errors, missing prerecorded-track bits*, illegal addresses, etc. Front-panel *write-lock switches* can protect selected tracks from overwriting, e.g., to protect system programs

D. REAL-TIME CLOCKS

Timing pulses needed to relate computer operations to real time are derived either from the 60-Hz line frequency (50 Hz in Europe) or from a 10-Hz to 1-MHz crystal oscillator (*clock oscillator*); counter flip-flops yield submultiples of the clock frequency, as desired. Timing pulses can be started by an external gate signal or by a programmed instruction setting a control-register bit (Sec 5-4), or the clock may run free.

Timing pulses are counted by a *clock counter*. If the clock counter is initially reset to 0 by a programmed I/O instruction, the count will be proportional to *elapsed time* and can be read by the computer program when desired. *To mark a preset time interval*, we preset the clock-counter recognition gate (NAND gate, see also Table 1-5*a*), which detects when the counter reaches 0 after *N* clock pulses. The gate output then interrupts the processor (*clock interrupt*). The ensuing interrupt-service routine performs whatever timed operation is wanted and can reset the clock counter to *repeat periodic cycles*.

In many minicomputers, the clock counter is not a flip-flop register but an *incrementable memory location* (memory-increment technique, Sec 5-21).

E. DIGITAL-TO-ANALOG CONVERTERS

The most commonly used *digital-to-analog converters (DACs)* are resistance networks whose output voltage or current is determined by an analog input (*reference voltage*) and a digit number set into the DAC flip-flop register. Each register bit controls one of the electronic switches (*bit switches*), which together determine the correct output. As a typical example, the *ladder-network DAC* of Fig 5-22*a* is designed to convert 2s-complement-coded binary numbers (Tables 1-1 and 1-2) into positive and negative analog output. Converters for many other codes (e.g., BCD, Table 1-4) exist (Ref 18)

If two or more DACs must be updated simultaneously (as in cathode-ray-tube displays, Se

TABLE 5-2. Typical Minicomputer Peripheral Devices and I/O Instructions (see also Secs. 3-10, 3-14, and 7-9) *(Continued)*

7-9, or analog hybrid computers), the *double-buffering* scheme of Fig. 5-4d permits individual loading of DAC buffer registers. All DACs can then be updated simultaneously through a programmed instruction producing a common transfer pulse. If one must service more DACs than there are readily available I/O addresses, DAC addresses can be entered as data words into a control register (*DAC address register*).

F. ANALOG-TO-DIGITAL CONVERTERS

Analog-to digital converters (*ADCs*) are discussed in detail in Ref. 18. There are two principal types. Many digital voltmeters employ *analog-to-time conversion*; i.e., the START CONVERSION command causes a binary or decimal counter to count clock pulses while an analog sweep voltage varies between a reference voltage level and a voltage level proportional to the unknown input (or, in the more accurate *integrating converters*, to a time average of the input). See Ref. 18. The count then terminates, and the CONVERSION COMPLETE flag level goes up, the counter can now be read by the digital computer. 8- to 14-bit precision is possible, with conversion times between 0.5 and 500 msec.

Faster binary ADCs employ the *feedback principle* illustrated in Fig. 5-22b. After the START CONVERSION signal, *digital logic tries to set a DAC register so that the DAC output approximates the unknown analog input as closely as possible*. A *voltage comparator* (basically a high-gain amplifier) produces logic 1 output if the comparison DAC output is too large, and 0 otherwise. The most commonly used ADC feedback logic successively tries the sign bit (with all other DAC bits at 0), then the most significant bit, etc. (*successive-approximation ADC*). This requires n voltage comparisons for n bits. The CONVERSION COMPLETE flag goes up, and the ADC output can be read from the DAC register when the comparison voltage equals the unknown input within one-half of the voltage step determined by the least significant bit. Up to 15-bit precision is possible, but that may require 30 to 50 μsec; 1 to 20 μsec is typical for 8- to 12-bit precision.

The ADC flag is reset (cleared) by the READ ADC instruction and/or by the START CONVERSION instruction or timing pulse.

An ADC often serves multiple analog channels through an analog-switching scheme (relay or electronic *analog multiplexer*). The multiplexer address is determined by a control register (*multiplexer address register*), which may be set by the digital computer or can be incremented to scan successive input channels.

Because of the finite ADC conversion time, accurately timed sampling of time-variable analog signals may require an analog *sample-hold circuit*, which holds the desired analog voltage sample long enough for conversion (Ref. 18).

NOTE: Other important and interesting peripheral-interface systems discussed in this book are *cathode-ray-tube displays* (Sec. 7-8) and *communication-system interfaces* (Sec. 7-14).

models for new or modified drivers. References 10 and 11 give detailed instructions for writing drivers fitted to Hewlett-Packard computers and operating systems.

5-31. Input/Output Control Systems (IOCS). (a) Subroutine Calls for Device Drivers. The subroutine calling sequence for a device-driver initiator (which may also pass parameters to its continuator) must transfer device number and function, buffer location(s) and size(s), and pointers to error routines. To simplify and standardize calls for I/O operations and to simplify assignment of devices to different tasks (Sec. 5-30), the better minicomputer systems incorporate their device-driver libraries into a special system program, the input/output control system (IOCS), which is usually furnished as part of a monitor or executive system (Sec. 3-12).

With IOCS, the device drivers in the IOCS library are never called *directly*. Instead, user programs requesting I/O call on a single master subroutine, IOCS, whose four-word to eight-word calling sequence specifies device, functions, formatting (if any), buffer(s), and error exit:

```
JUMP AND SAVE    IOCS
(device/function/format/code)
(address of error routine)
(buffer starting address)
(buffer size)
```

The proper codes to be used will be found in your minicomputer manual. The IOCS subroutine passes the information in the calling sequence to the appropriate device driver, calls its initiator routine, and returns control to the calling program. Interrupts which occur during or on termination of the transfer are processed entirely by the system; *no interrupt-handling subroutines are required in the user's program* (Ref. 10).

With such a system, the user program requires only a single .EXTERNAL reference (Sec. 4-19) to IOCS for any and all drivers. Some operating systems, however, save core space by requiring a list of the devices or drivers actually used, e.g.,

```
.EXTERNAL    IOCS    3, 4, 7
```

so that the loader need only load the device drivers which are actually used.

NOTE: To make IOCS practical, the assembler used must permit *external linkage* to IOCS, or else the IOCS subroutine and the associated device drivers must be supplied by the assembler itself; just as some FORTRAN compilers supply their own I/O programming system.

(b) Buffer-status Management. In connection with appropriate device handlers, an input/output control system also maintains *status words* which can specify not only device status but also the buffer(s) currently kept busy by program or devices. Special IOCS-subroutine calling sequences or IOCS macros can read these status words for use by the user program. We may also have special IOCS subroutines or macros (e.g., .WAIT *m, n*) which *stop the user program until a required buffer is free* (this is analogous to, but *not* the same as, a skip loop waiting for device readiness). More elaborate device handlers may do such buffer management automatically.

(c) IOCS Macros (see also Sec. 4-21). As a further convenience, the larger minicomputer input/output control systems replace each IOCS subroutine call and its elaborate calling sequence by an IOCS macro, such as

```
.WRITE    a, b, c, d
.READ     a, b, c, d
```

where the arguments *a, b, c, d* specify device, function, format, buffer, and

error exit. Assembly-language I/O programming then approaches the simplicity of FORTRAN I/O, with various optional tradeoffs between programming flexibility and simplicity. It is, for instance, possible to standardize buffer sizes so that they need not be included as parameters.

(d) Device-independent IOCS (see also Sec. 3-12). Good input/output control systems permit **device-independent programming**; i.e., user programs refer to each peripheral device by the **logical device number** to which a physical device (identified by a **physical device number**, name, or code) is attached through program statements. Device-independent IOCS will, of course, let us reassign multiple similar peripherals such as tape drivers, but it does more. Through simple device-number reassignment, an otherwise unchanged program can, for instance, either process real data from a communications interface or be tested with analogous data from cards or tape; or a compiler can accept source programs from paper tape, magnetic tape, or a disk.

Device-independent IOCS subroutines or macros do not call device drivers directly but reference a **device-assignment table** which assigns a *physical* device number to each *logical* device number. The user sets up his own "normal" device assignments for system programs (monitor, assembler, compiler, debugging program, etc.) when the computer system is first put together; most computers have a special conversational program ("system generator") for this purpose. The "normal" device assignments are reestablished whenever a system program is first loaded, but the user can change device assignments through special commands like

```
.ASSIGN TTYO 3 / Teleprinter becomes
               / logical device 3
```

The user can also request printout or display of the current device assignment table.

5-32. Discussion. A convenient input/output control system permits the user to concentrate on his applications program without having to bother with the massive detail involved in I/O, buffer management, formatting, and packing/unpacking operations. IOCS is an indispensable part of modern operating systems which make it simple to store, retrieve, load, combine, and execute modular programs. The elaborate device drivers and multiple subroutine calls of an IOCS system do exact a price in computing time. In the most time-critical applications, users may have to write simplified device drivers and insert them into their own programs.

REFERENCES AND BIBLIOGRAPHY

Interface Design

(Refer also to the interface manuals of various digital-computer manufacturers.)

1. Berger, F. R: Characteristics of Priority Interrupts, *Datamation*, June 1965.
2. Andelman, S. J.: Real-time I/O Techniques, *Comput. Des.*, May 1966.
3. Klerer, M., and G. A. Korn: *Digital Computer User's Handbook*, McGraw-Hill, New York, 1967.
4. Schmidt, W. E.: Methods for Priority Interrupts and Their Implication for Hybrid Programs, *Simulation*, July 1967.
5. Marston, G. P.: A Medium-scale Hybrid Interface, *Simulation*, May 1968.
6. Korn, G. A.: Digital-computer Interface Systems, *Simulation*, December 1968.
7. ——— et al.: A New Graphic-display/Plotter for Small Digital Computers, *Proc. SJCC*, 1969.
8. Wilkins, J.: "The PDP-9/LOCUST Interface," M.S. thesis, University of Arizona, 1969.
9. Van Gelder, M. K., and A. W. England: A Primer on Priority-interrupt Systems, *Control Eng.*, March 1969.
10. *A Pocket Guide to Interfacing HP Computers*, Hewlett-Packard Corporation, Cupertino, Calif., 1969.
11. *Driver Manual*, Hewlett-Packard Corporation, Palo Alto, Calif., 1969.
12. Flores, I.: *Computer Organization*, chaps. 2 and 3, Prentice-Hall, Englewood Cliffs, N.J., 1969.
13. CAMAC, a Modular Instrumentation System for Data Handling, *Rept. EUR* 4100e, EURATOM, Ispra, Italy, March 1969.
14. Meng, J. D.: A Serial Input/Output Scheme for Small Computers, *Comput. Des.*, March 1970.
15. Holland, E.: Minicomputer I/O and Peripherals, *IEEE Comput. Group News*, July–August 1970.
16. Coury, F. F.: A Systems Approach to Minicomputer I/O, *Proc. SJCC*, 1970.
17. Babiloni, L., et al.: On-line Tradeoff, Hardware versus Software, *Control Eng.*, October 1970.
18. Korn, G. A., and T. M. Korn: *Electronic Analog and Hybrid Computers* (revised edition), McGraw-Hill, New York, 1972.
19. Huskey, H. D., and G. A. Korn: *Computer Handbook*, McGraw-Hill, New York, 1962.

Digital-circuit Noise

20. Jones, J. P.: *Causes and Cures of Noise in Digital Systems*, Computer Design Publication Company, West Concord, Mass., 1964.
21. Korn, G. A.: Reduction of Digital Noise in Hybrid Analog-digital Computers, *Simulation*, March 1966.
22. Printed-circuit Shielding in *AFAL-TR-66-371*, Bunker-Ramo Corporation, April 1967.
23. Walker, R. M., and R. A. Aldrich (Fairchild Semiconductor): Standard IC's for Digital Data Communication, *Electron. Eng.*, March 1968.
24. Saenz, R. G., and E. M. Fulcher (RCA): An Approach to Logic Circuit Noise, *Comput. Des.*, April 1969.
25. *Balun Applications*, Pulse Engineering, Inc., Santa Clara, Calif., 1969.
26. Heniford, B.: Noise in 54/74 TTL Systems, *Appl. Bull.* CA-108, Texas Instruments, Inc., Dallas, 1968.
27. Garrett, L. S.: Integrated-circuit Digital-logic Families, Part III, *IEEE Spectrum*, December 1970.

Shielding, Interconnections, and Grounding

28. Stewart, E. L. (Martin/Baltimore): Grounds, Grounds, and More Grounds, *Simulation*, August 1965.
29. ———: Noise Reduction in Interconnect Lines, *Simulation*, September 1965.
30. Sargent, R. S., and T. R. Kuchlewski: Reduction of Noise in Low-impedance (Ground) Lines, *Analog Rept.* 16, Martin Co., Baltimore, 1963.
31. Marsh, J. M., and V. P. Scott: Analog-facility Grounding Techniques, *WAPD-T-1701*, Bettis At. Power Lab., Westinghouse Electric Company, Pittsburgh, 1964.
32. Marimon, R. L. (NOTS/Pasadena): Ground Rules for Low-frequency High-gain Amplifiers, *Electron. Des.*, April 12, 1963.

33. Morrison, R. (Dynamics Instrument Company): Shielding Signal Circuits, *ISA J.*, March 1966.
34. EID Staff Grounding Low-level Instrumentation Systems, *EID*, January 1966.
35. Ginn, D. (Data Control Systems, Inc.): Data-system Grounding Techniques, *Telemetry*, July 1966.
36. Burdi, A. J. (Interstate Electronics Corp.): Keep Sampled-data Systems Accurate, *Electron. Des.*, May 24, 1966.
37. McCullough, W. (Hewlett-Packard): How to Reduce Common-mode Effects, *EEE*, February 1967.
38. Bowers, W. J. (Scientific Data Systems): Minimizing Common-mode Noise, *Control Eng.*, July 1967.
39. *Dynamic Bridge Differential Amplifier 770-440*, Redcor Corporation, Canoga Park, Calif., 1967.
40. Klipec, B. (J. Moore and Company): Controlling Noise in Instrument Circuits, *Control Eng.*, March 1968.
41. Bailey, S. J.: Using Telephone Lines for Critical Signal Transmission, *Control Eng.*, March 1969.
42. Buntenbach, R. W : On the Design of Ground Circuits, *STAR* N68-3573, Lawrence Radiat. Lab., Livermore, Calif., 1968
43. Budzilovich, P. N.: Electrical Noise: Its Nature, Causes, Solutions, *Control Eng.*, May 1969.
44. Morrison, B.: *Grounding and Shielding Techniques in Instrumentation*, Wiley, New York, 1967.

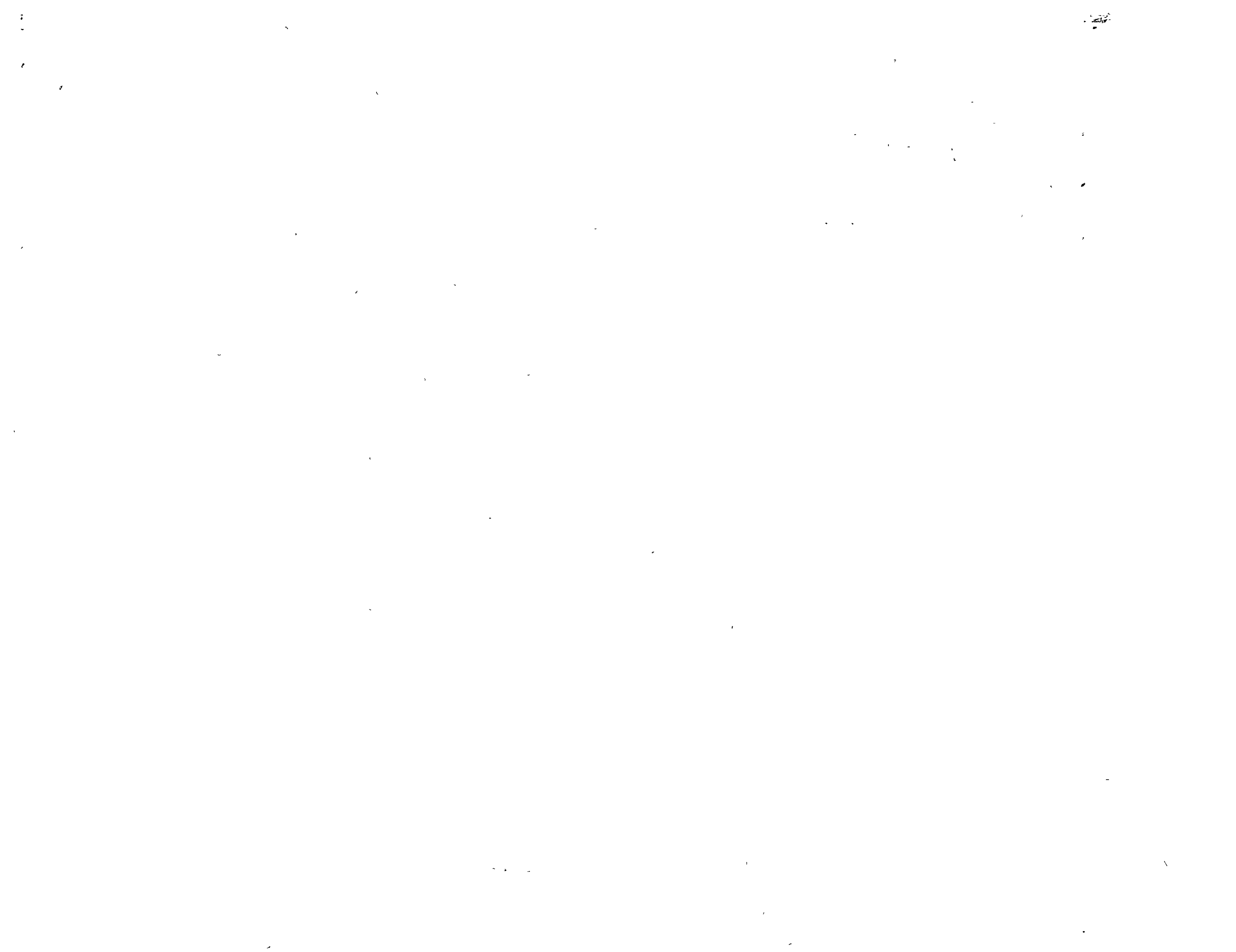
CHAPTER 6

DEVELOPMENT AND EVALUATION OF IMPROVED MINICOMPUTER ARCHITECTURES

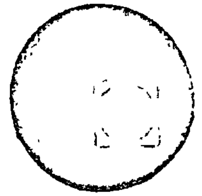
INTRODUCTION AND SURVEY

The progress of small-computer design, pushed forward by inexpensive logic circuits and fierce competition, is made more deliberate by the need to spread grievous software costs over as many machines as possible. Hardware and software, in any case, must be designed and evaluated together. Their development is the system-designer's response to widely divergent markets, whose requirements are not always well defined for him. The enormous versatility of the quantity-produced miniprocessor is the key to its success, but consider just how wildly these applications differ:

1. *Logic-sequencing and timing operations* (as in simple process control). Sequences may be complex, but individual operations are simple and need not be very fast.
2. *Simple logic or character-string operations, but with a requirement for fast service and/or large volume* (communications handling, front-ending larger computers, supervision of time-critical operations or measurements). Such applications favor powerful interrupt systems, direct memory access, and fast processing—but simple instruction sets will do.
3. *More complicated on-line numerical computations* (data processing, digital filtering, simulation). Here one needs more advanced arithmetic—even accessory array or floating-point-arithmetic processors; or one tries to pass the load to a larger computer.



centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



APLICACION DE MINICOMPUTADORAS

EXPOSICION DEL DR. ADOLFO GUZMAN

diciembre de 1975.

PALACIO DE MINERIA
Tacuba 5, primer piso. México 1, D. F.
TELEFONOS: 513-27-95
512-31-23 521-73-35

5

loaders

The purpose of this chapter is to discuss various loader schemes and to present the design of a direct-linking loader.

As illustrated in the previous chapter, the user's *source program* decks are usually converted to *object program* decks (machine language) by assemblers and compilers. The *loader* is a program which accepts the object program decks, prepares these programs for execution by the computer, and initiates the execution (see Fig. 5.1).

In particular, the loader must perform four functions:

1. Allocate space in memory for the programs *(allocation)*
2. Resolve symbolic references between object decks *(linking)*
3. Adjust all address dependent locations, such as address constants, to correspond to the allocated space *(relocation)*
4. Physically place the machine instructions and data into memory *(loading)*

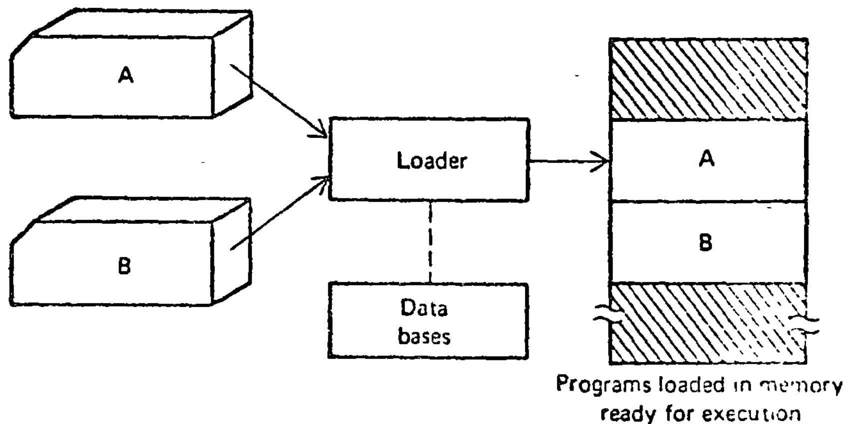


FIGURE 5.1 General loading scheme

Most of the examples used in this section are based upon the IBM System/370 assembler and loader. Several of the alternative loader schemes discussed are based upon computers with a fixed-word, direct-address instruction format, such as the IBM 7094, IBM 1130, UNIVAC 1108 and GE 635.

For simplicity of presentation, we assume card deck inputs – these of course may actually be card images on tape or any secondary storage.

5.1 LOADER SCHEMES

In this section we discuss various schemes for accomplishing the four functions of a loader. It is desirable to introduce the term *segment*, which is a unit of information that is treated as an entity, be it a program or data. Usually a segment corresponds to a single source or object deck. It is possible to produce multiple program or data segments in a single source deck by means of the assembly CSECT (Control Section) pseudo-op, the FORTRAN COMMON statement, or the PL/I EXTERNAL STATIC data attribute.

5.1.1 "Compile-and-Go" Loaders

One method of performing the loader functions is to have the assembler run in one part of memory and place the assembled machine instructions and data, as they are assembled, directly into their assigned memory locations (Fig. 5.2). When the assembly is completed, the assembler causes a transfer to the starting instruction of the program. This is a simple solution, involving no extra procedures. It is used by the WATFOR FORTRAN compiler and several other language processors.

Such a loading scheme is commonly called "compile-and-go" or "assemble-and-go." It is relatively easy to implement. The assembler simply places the code into core, and the "loader" consists of one instruction that transfers to the starting instruction of the newly assembled program.

However, there are several apparent disadvantages. First, a portion of memory is wasted because the core occupied by the assembler is unavailable to the object program. Second, it is necessary to retranslate (assemble) the user's program deck every time it is run. Third, it is very difficult to handle multiple segments, especially if the source programs are in different languages (e.g., one subroutine in assembly language and another subroutine in FORTRAN or PL/I). This last disadvantage makes it very difficult to produce orderly modular programs as discussed in the design of assemblers.

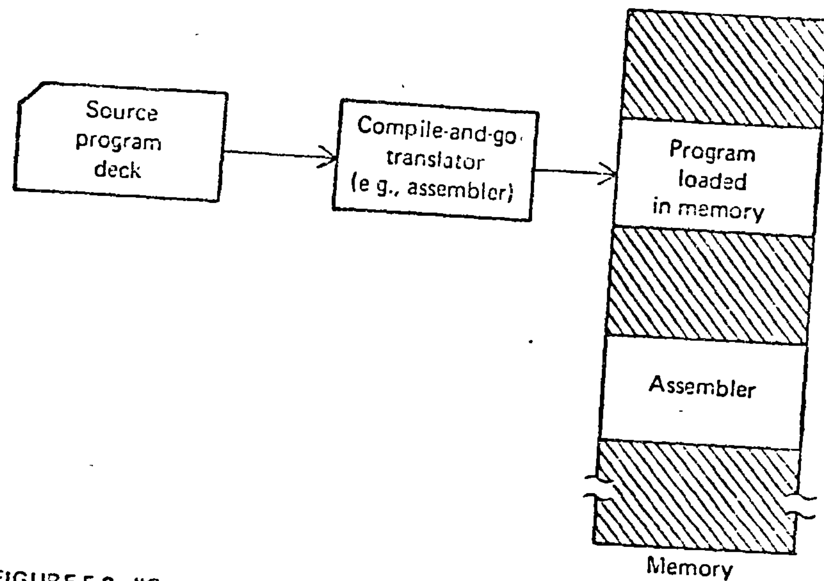


FIGURE 5.2 "Compile-and-go" loader scheme

5.1.2 General Loader Scheme

Outputting the instructions and data as they are assembled circumvents the problem of wasting core for the assembler. Such an output could be saved and loaded whenever the code was to be executed. The assembled programs could be loaded into the same area in core that the assembler occupied (since the translation will have been completed). This output form, which may be on cards containing a coded form of the instructions, is called an *object deck*.

The use of an object deck as intermediate data to avoid one disadvantage of the preceding "compile-and-go" scheme requires the addition of a new program to the system, a loader (Fig. 5.3). The *loader* accepts the assembled machine instructions, data, and other information present in the object format, and places machine instructions and data in core in an executable computer form. The loader is assumed to be smaller than the assembler, so that more memory is available to the user. A further advantage is that reassembly is no longer necessary to run the program at a later date.

Finally, if all the source program translators (assemblers and compilers) produce compatible object program deck formats and use compatible linkage conventions, it is possible to write subroutines in several different languages since the object decks to be processed by the loader will all be in the same "language" (machine language).

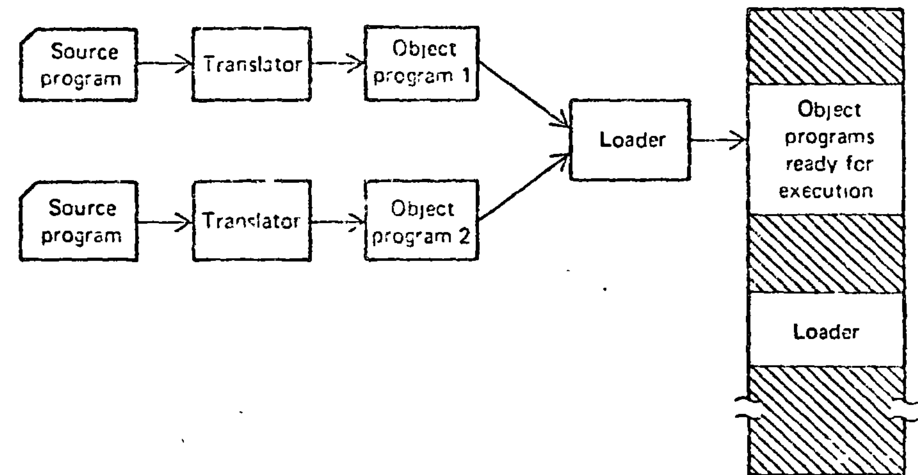


FIGURE 5.3 General loader scheme

5.1.3 Absolute Loaders

The simplest type of loader scheme, which fits the general model of Figure 5.3, is called an absolute loader. In this scheme the assembler outputs the machine language translation of the source program in almost the same form as in the "assemble-and-go" scheme, except that the data is punched on cards (object deck) instead of being placed directly in memory. The loader in turn simply accepts the machine language text and places it into core at the location prescribed by the assembler. This scheme makes more core available to the user since the assembler is not in memory at load time.

Absolute loaders are simple to implement but they do have several disadvantages. First, the programmer must specify to the assembler the address in core where the program is to be loaded. Furthermore, if there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in his other subroutines to perform subroutine linkage.

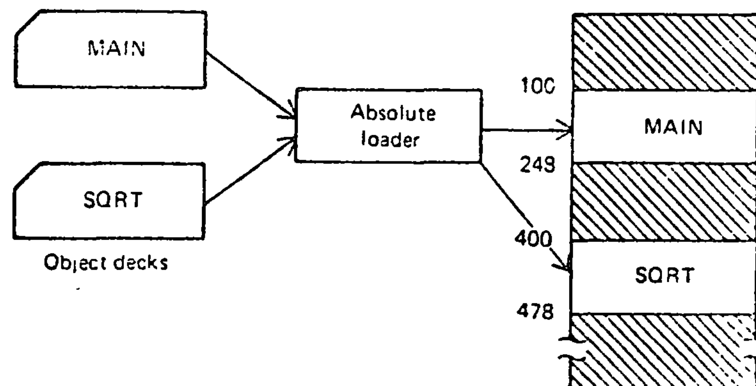
Figure 5.4 illustrates the operation of an absolute assembler and an absolute loader. The programmer must be careful not to assign two subroutines to the same or overlapping locations.

The MAIN program is assigned to locations 100-247 and the SQRT subroutine is assigned locations 400-477. If changes were made to MAIN that increased its length to more than 300 bytes, the end of MAIN (at 100 + 300 = 400) would overlap the start of SQRT (at 400) It would then be necessary to assign SQRT

```
         MAIN program

MAIN     START   100
         BALR    12,0
         USING   MAIN+2,12
           .
           .
         L       15,ASQRT }  Call
         BALR    14,15    }  SQRT
           .
           .
ASQRT    DC      F'400'  }  Address
         END             }  of SQRT

         SQRT subroutine

SQRT     START   400
         USING   *,15
           .            }  Compute
           .            }  square root
         BR      14     }  Return
         END

      SOURCE DECK INPUT
   TO ABSOLUTE ASSEMBLER
```

```
            MAIN program
   Location        Instruction

   100        BALR     12,0
   102
    .           .
    .           .
   120        L        15,142(0,12)
   124        BALR     14,15
   126
    .           .
    .           .
   244        F'400'
   248

            SQRT subroutine

   400
    .           .
    .           .
   476        BCR      15,14
   478          .

        OBJECT DECK OUTPUT
     FROM ABSOLUTE ASSEMBLER
```

Part (a)



Part (b)

FIGURE 5.4 Absolute loader example

to a new location by changing its START pseudo-op card and reassembling it. Furthermore, it would also be necessary to modify all other subroutines that referred to the address of SQRT. In situations where dozens of subroutines are being used, this manual "shuffling" can get very complex, tedious, and wasteful of core.

The four loader functions are accomplished as follows in an absolute loading scheme:

1. Allocation – by programmer
2. Linking – by programmer
3. Relocation – by assembler
4. Loading – by loader

5.1.4 Subroutine Linkages

In this section we briefly discuss, from a programmer's point of view, the special mechanism for calling another subroutine in an assembly language program.

The problem of subroutine linkage is this: a main program A wishes to transfer to subprogram B. The programmer, in program A, could write a transfer instruction (e.g., BAL 14,B) to subprogram B. However, the assembler does not know the value of this symbol reference and will declare it as an error (undefined symbol) unless a special mechanism has been provided.

This mechanism is typically implemented with a relocating or a direct-linking loader. The assembler pseudo-op EXTRN followed by a list of symbols indicates that these symbols are defined in other programs but referenced in the present program. Correspondingly, if a symbol is defined in one program and referenced in others, we insert it into a symbol list following the pseudo-op ENTRY. In turn, the assembler will inform the loader that these symbols may be referenced by other programs. For example, the following sequence of instructions may be a simple calling sequence to another program:

```
MAIN     START
         EXTRN   SUBROUT
         ---
         ---
         L       15,=A(SUBROUT) }  CALL SUBROUT
         BALR    14,15          }
           .
           .
         END
```

The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable referenced but not defined in this program. The load

instruction loads the address of that variable into register 15. The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next instruction in register 14. In most assemblies we may simply use a CALL SUBROUT macro, which is translated by the assembler into a calling sequence as shown above.

Now we may see the reason for programming conventions, since it is necessary for both the caller and called subprograms to cooperate. On a 360 we observe the convention that register 15 is used as a linkage and base register. Note that the caller in the above program has loaded register 15 with the beginning address of the subroutine being called. Thus the called subroutine does not have to load a base register. Register 14 contains the return address of the caller. The programmer must not use register 14 within a subprogram unless he saves its contents and restores them before he returns. A typical sequence for subroutines is:

```
SUBROUT     START
            USING     *,15
              :
            BR        14
            END
```

No BALR instruction is necessary at the beginning since register 15 was already loaded with the address of the start of this program. The BR 14 instruction is an unconditional branch to the address contained in register 14, which is the return address of the calling program. In Appendix B we present in more detail the methods that are typically used on a 360 and discuss further the calling sequences. The following discussion introduces the basic mechanism used.

ASSEMBLER LINKAGE PSEUDO-OPS

Subroutine and Entry Naming (START and ENTRY Pseudo-ops)

```
A         START     defines subroutine A
          ENTRY     B1,B2,B3,... defines locations B1,...,
                                 Bn as additional subroutine
                                 entry points
B1        -----
B2        -----
```

The uses of multiple entry points are:

1. Common coding
   Example. SIN and COS involve basically the same computations and could employ different entry points of the same routine.
2. Collecting together related routines for convenience.
3. Better or convenient access to common data base

SUBROUTINE REFERENCE (EXTRN PSEUDO-OP)

Assembler symbols are either internal or external. External means that their value is not known to the assembler but will be provided by the loader – the action of the loader will be discussed in the following section.

```
EXTRN E1,E2,etc.
```

defines E1, E2, etc. as external symbols; to be used in address constants

```
Example   CALL BETA becomes:
                    EXTRN     BETA
                      :
                    L         15,ABETA
                    BALR      14,15
                      .
               ABETA DC       A(BETA)
```


5.1.5 Relocating Loaders

To avoid possible reassembling of all subroutines when a single subroutine is changed, and to perform the tasks of allocation and linking for the programmer, the general class of *relocating loaders* was introduced. An example of a relocating loader scheme is that of the Binary Symbolic Subroutine (BSS) loader such as was used in the IBM 7094, IBM 1130, GE 635, and UNIVAC 1108. The BSS loader allows many procedure segments, yet only one data segment (common segment). The assembler assembles each procedure segment independently and passes on to the loader the text and information as to relocation and intersegment references.

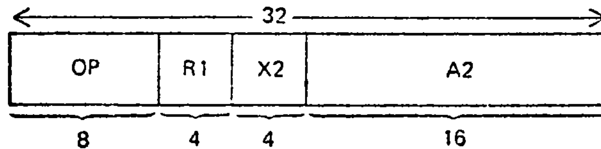
The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references. In addition, there is information (relocation information) as to locations in this program that need to be changed if it is to be loaded in an arbitrary place in core, i.e., the locations which are dependent on the core allocation.

For each source program the assembler outputs a text (machine translation of the program) prefixed by a *transfer vector* that consists of addresses containing names of the subroutines referenced by the source program. For example, if a Square Root Routine (SQRT) was referenced and was the first subroutine called, the first location in the transfer vector would contain the symbolic name SQRT

The statement calling SQRT would be translated into a transfer instruction indicating a branch to the location of the transfer vector associated with SQRT.

The assembler would also provide the loader with additional information, such as the length of the entire program and the length of the transfer vector portion. After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector. It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector. Thus, the execution of the call SQRT statement would result in a branch to the first location in the transfer vector, which would contain a transfer instruction to the location of SQRT.

The BSS loader scheme is often used on computers with a fixed-length direct-address instruction format. For example, *if* the format of the 360 RX instruction were:



where A2 was the 16-bit absolute address of the operand, this would be a *direct address* instruction format. Such a format works if there are less than 2^{16} = 65,536 bytes of storage, as was the case with most of the early computers and is still true for many of the current "minicomputers" and "midicomputers."

Since it is necessary to relocate the address portion of every instruction, computers with a direct-address instruction format have a much more severe relocation problem than the 360. In the absence of 360-type base registers, this problem is often solved by the use of "*relocation bits*." The assembler associates a bit with each instruction or address field. If this bit equals one, then the corresponding address field must be relocated; otherwise the field is not relocated. These relocation indicators, surprisingly enough, are known as relocation bits and are included in the object deck.

Figure 5.5 illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader. The function of the program is not important: it supposedly calls the SQRT subroutine to get the square root of 9. If the result is not 3, it transfers to a subroutine called ERR. Since this is a direct-address computer, there is no base register field in the object code and no need for a USING pseudo-op in the source program. The EXTRN pseudo-op identifies the symbols SQRT and ERR as the names of other subroutines; since the locations of these symbols are not defined in this subroutine, they are called *external symbols*. For each external symbol the assembler generates a four-

byte fullword at the beginning of the program, containing the EBCDIC characters for the symbol (for simplicity, we are assuming that symbols are not more than four characters long). These extra words are called transfer vectors. Every reference to an external symbol is assigned the address of the corresponding transfer vector word. In addition, for every halfword (two bytes) in the program, the assembler produces a separate relocation bit. For example, the assembled instruction ST 14,36 is assigned relocation bits 01 since the first halfword contains the op-code, register field, and index field which should not be relocated but the second halfword contains the relative address 36, which must be relocated.

Program length = 48 bytes
Transfer vector = 8 bytes

Source program				*Rel. addr.*	*Relocation*	*Object code*	
MAIN	START						
	EXTRN	SQRT		0	00	'SQRT'	
	EXTRN	ERR		4	00	'ERR*b*'	
	ST	14,SAVE	Save return address	8	01	ST	14,36
	L	1,=F'9'	Load test value	12	01	L	1,40
	BAL	14,SQRT	Call SQRT	16	01	BAL	14,0
	C	1,=F'3'	Compare answer	20	01	C	1,44
	BNE	ERR	Transfer to ERR	24	01	BC	7,4
	L	14,SAVE	Get return address	28	01	L	14,36
	BR	14	Return to caller	32	0	BCR	15,14
				34	0	(Skipped for alignment)	
SAVE	DS	F	Temp. loc.	36	00	(Temp location)	
	END			40	00	9	
				44	00	3	

FIGURE 5.5 Assembly of program for "direct-address" 360

Figure 5.6 illustrates the contents of memory after the programs have been loaded by the BSS loader. Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400. Using the program length information, the loader placed the subroutines SQRT and ERR at the next available locations which were 448 and 526, respectively. Finally, the transfer vector words were changed to contain

Absolute address	Relative address	Contents	
⋮	⋮		
400	0	BC 15,448	Length = 48 bytes
404	4	BC 15,526	
408	8	ST 14,436	
412	12	L 1,440	
416	16	BAL 14,400	
420	20	C 1,444	
424	24	BC 7,404	
428	28	L 4,436	
432	32	BCR 15,14	
436	36	(Temp loc)	
440	40	9	
444	44	3	
⋮		SQRT	Length = 78 bytes
526		ERR	
⋮			

FIGURE 5.6 BSS loading of programs for "direct access" 360

branch instructions to the corresponding subroutines. Thus, the four functions of the loader (allocation, linking, relocation, and loading) were all performed automatically by the BSS loader.

It should be noted that the relocation bits are used to solve the problem of relocation, the transfer vector is used to solve the problem of linking; and the program length information is used to solve the problem of allocation.

There are several disadvantages to the BSS loader scheme. First, the transfer vector linkage is only useful for transfers, and is not well-suited for loading or storing external data (data located in another procedure segment). Second, the

transfer vector increases the size of the object program in memory. Finally, the BSS loader, as described, processes procedure segments but does not facilitate access to data segments that can be shared. This last shortcoming is overcome in many BSS loaders by allowing one common data segment, often called *COMMON*. This facility is usually implemented by extending the relocation bits scheme to use two bits per halfword address field: if the bits are 01, the halfword is relocated relative to the procedure segment, and if they are 10, it is relocated relative to the address of the single common data segment. If the bits are 00 or 11, the halfword is not relocated.

5.1.6 Direct-Linking Loaders

A direct-linking loader is a general relocatable loader, and is perhaps the most popular loading scheme presently used. Throughout our discussion of compilers in later sections, we will assume that the system employs such a loader.

The direct-linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments. This provides flexible intersegment referencing and accessing ability, while at the same time allowing independent translations of programs.

A design of a direct-linking loader will be given later in this chapter. In this section we present a general format for the assembler output with such a loading scheme, patterned after those used in the IBM 370. While the formats themselves are somewhat arbitrary, the information that the assembler must give to the loader is not. The assembler (translator) must give the loader the following information with each procedure or data segment:

1. The length of segment
2. A list of all the symbols in the segment that may be referenced by other segments and their relative location within the segment
3. A list of all symbols not defined in the segment but referenced in the segment
4. Information as to where address constants are located in the segment and a description of how to revise their values
5. The machine code translation of the source program and the relative addresses assigned

A simple example using a direct-linking loading scheme is presented in Figure 5.7. A source program (left-hand column) is translated by an assembler to produce the object code depicted in the right column. Mnemonic machine codes have again been used in the translation shown.

Card no.	Program			Rel. loc.	Translation	
1.	JOHN	START				
2.		ENTRY	RESULT			
3.		EXTRN	SUM			
4.		BALR	12,0	0	BALR	12,0
5.		USING	*,12			
6.		ST	14,SAVE	2	ST	14,54(0,12)
7.		L	1,POINTER	6	L	1,46(0,12)
8.		L	15,ASUM	10	L	15,58(0,12)
9.		BALR	14,15	14	BALR	14,15
10.		ST	1,RESULT	16	ST	1,50(0,12)
11.		L	14,SAVE	20	L	14,54(0,12)
12.		BR	14	24	BCR	15,14
				26	--	
13.	TABLE	DC	F'1,7,9,10,3'	28	1	
				32	7	
				36	9	
				40	10	
				44	3	
14	POINTER	DC	A(TABLE)	48	28	
15.	RESULT	DS	F	52	--	
16	SAVE	DS	F	56	--	
17.	ASUM	DC	A(SUM)	60	?	
18.		END		64		

FIGURE 5.7 Assembly source program and its translation

Card number 14 of Figure 5.7 contains a Define Constant (DC) pseudo-operation which instructs the assembler to create a constant with the value of the address of TABLE, and cause this constant to be placed in the location labelled POINTER. At this point the assembler does not know the final absolute address of TABLE since it has no idea where the program is going to be loaded. It knows, however, that the address is the 28th byte from the beginning of this program. The assembler will put a 28 in POINTER and inform the loader that the content of location POINTER is incorrect if this program is loaded anywhere except absolute location 0. For instance, if this program were loaded in location 2000, the loader would have to change the contents of POINTER to be a 2028.

Card number 17 of Figure 5.7 is another DC pseudo-op, which instructs the assembler to create a constant with the value of the address of the subroutine SUM and cause this constant to be placed in the location labelled ASUM. Since the assembler has no idea where the procedure SUM will be loaded, it cannot generate this constant. Thus, the assembler must provide information to the loader that will cause it to put the final absolute address of SUM at the

designated location (ASUM) when the programs are loaded.

We have named the program JOHN. Hence, JOHN is a symbol that may be referenced externally or "called" by other programs. We also have stated that the symbol RESULT may be referenced by other programs. These facts must be passed on to the loader.

The design of the direct-linking loading scheme we present is similar to the standard IBM 370 scheme. The assembler produces four types of cards in the object deck: ESD, TXT, RLD, and END. External Symbol Dictionary (ESD) cards contain information about all symbols that are defined in this program but that may be referenced elsewhere, and all symbols referenced in this program but defined elsewhere. The text (TXT) cards contain the actual object code translated version of the source program. The Relocation and Linkage Directory (RLD) cards contain information about those locations in the program whose contents depend on the address at which the program is placed. For such locations the assembler must supply information enabling the loader to correct their contents. The END card indicates the end of the object deck and specifies the starting address for execution if the assembled routine is the "main" program. Figure 5.8 depicts the information that would appear for the preceding program on the ESD, TXT, RLD, and END cards.

The reference numbers in Figure 5.8 do not actually appear on the cards. They are for the benefit of the reader, and each denotes the card number of the original program that resulted in the object deck card; e.g., the first RLD card resulted from card number 14 in the original program.

As shown in Figure 5.8, three ESD cards are needed for the program JOHN. The first card contains the name of the program JOHN, which may be referenced externally. The "type" mnemonic we have used is SD, which means the symbol is a Segment Definition. The relative address of JOHN is 0, and the length is of the program that JOHN denotes, 64. On the next ESD card appears the symbol RESULT, which is a Local Definition (LD); its relative address is 52. The final ESD card specifies that the symbol SUM is an External Reference (ER). We will see in a later section how the ER symbols are actually used in conjunction with the RLD cards.

The TXT cards contain the actual assembled program. The format and use of these cards are similar to those for the absolute loader.

The RLD cards contain the following information:

1. The location of each constant that needs to be changed due to relocation
2. By what it has to be changed
3. The operation to be performed

The first RLD card of our example contains a 48, denoting the relative loca-

ESD cards

Reference no.	*Symbol*	*Type*	*Relative location*	*Length*
1	JOHN	SD	0	64
2	RESULT	LD	52	—
3	SUM	ER	—	—

TXT cards

Reference no.	*Relative location*	*Object code*	
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	
13	32	7	
13	36	9	
13	40	10	
13	44	3	
14	48	28	
17	60	0	

RLD cards

Reference no.	*Symbol*	*Flag*	*Length*	*Relative location*
14	JOHN	+	4	48
17	SUM	+	4	60

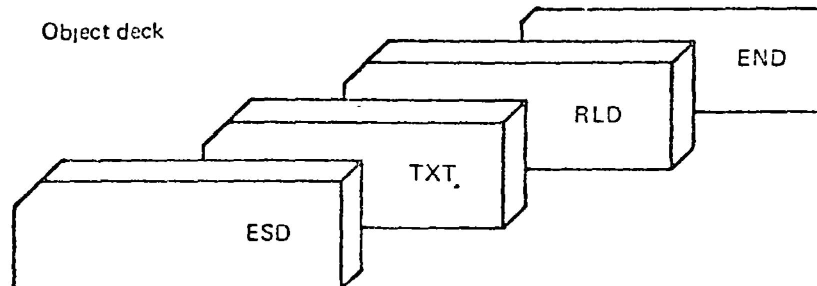


FIGURE 5.8 Example object deck for a direct-linking loader

tion of a constant that must be changed; a plus sign denoting that something must be added to the constant; and the symbol field indicating that the value of external symbol JOHN must be added to relative location 48. The relative value of JOHN is 0. When the program is loaded, the loader will determine its absolute value.

The second RLD card of our example contains a 60, denoting the relative location of a constant that must be changed. The symbol field indicates that the value of the external symbol SUM must be added to relative location 60. Although the assembler does not know the absolute address of SUM, the loader will later be able to fill in the correct value.

The process of adjusting the address constant of an internal symbol, such as TABLE, is normally called relocation; while the process of supplying the contents of an address constant for an external symbol, such as SUM, is normally referred to as linking. Significantly, the RLD card mechanism is used for both cases, which explains why they are called relocation *and* linkage directory cards. The reader may wish to compare this technique with the mechanisms used in the BSS relocating loader described earlier in this chapter.

5.1.7 Other Loader Schemes — Binders, Linking Loaders, Overlays, Dynamic Binders

There are numerous variations to the previously presented loader schemes.

One disadvantage of the direct-linking loader, as presented, is that it is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program. Since there may be tens and often hundreds of subroutines involved, especially when we include utility routines such as SQRT, etc., this loading process can be extremely time-consuming. Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space. These problems can be solved by dividing the loading process into two separate programs: a binder and a module loader.

A *binder* is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This output file is in a format ready to be loaded and is typically called a *load module*. The *module loader* merely has to physically load the module into core. The binder essentially performs the functions of allocation, relocation, and linking; the module loader merely performs the function of loading.

There are two major classes of binders. The simplest type produces a load module that looks very much like a single absolute loader deck. This means that the specific core allocation of the program is performed at the time that the

subroutines are bound together. Since this kind of module looks like an actual "snapshot" or "image" of a section of core, it is called a *core image module* and the corresponding binder is called a *core image builder*. A more sophisticated binder, called a *linkage editor*, can keep track of the relocation information so that the resulting load module, as an ensemble, can be further relocated and thereby loaded anywhere in core. In this case the module loader must perform additional allocation and relocation as well as loading, but it does not have to worry about the complex problems of linking.

In both cases, a program that is to be used repeatedly need only be bound once and then can be loaded whenever required. The core image builder binder is relatively simple and fast. The linkage editor binder is somewhat more complex but allows a more flexible allocation and loading scheme.

DYNAMIC LOADING

In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time. If the total amount of core required by all these subroutines exceeds the amount available, as is common with large programs or small computers, there is trouble! There are several hardward techniques, such as paging and segmentation, that attempt to solve this problem; these are discussed in Chapter 9. In this section we will present conventional dynamic loading schemes based upon the use of a binder prior to loading.

Usually the subroutines of a program are needed at different times: for example, pass 1 and pass 2 of an assembler are mutually exclusive. By explicitly recognizing which subroutines call other subroutines it is possible to produce an *overlay structure* that identifies mutually exclusive subroutines. Figure 5.9a illustrates a program consisting of five subprograms (A,B,C,D and E) that require 100K bytes of core. The arrows indicate that subprogram A only calls B, D and E; subprogram B only calls C and E; subprogram D only calls E; and subprograms C and E do not call any other routines. Figure 5.9b highlights the interdependencies between the procedures. Note that procedures B and D are never in use at the same time; neither are C and E. If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure. This happens to be 70K for the example in Figure 5.9b – procedures A, B, and C. Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.

In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed. We will not go into their specific details, but there are many binders capable of processing and allocating an overlay structure. The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the *overlay supervisor* or

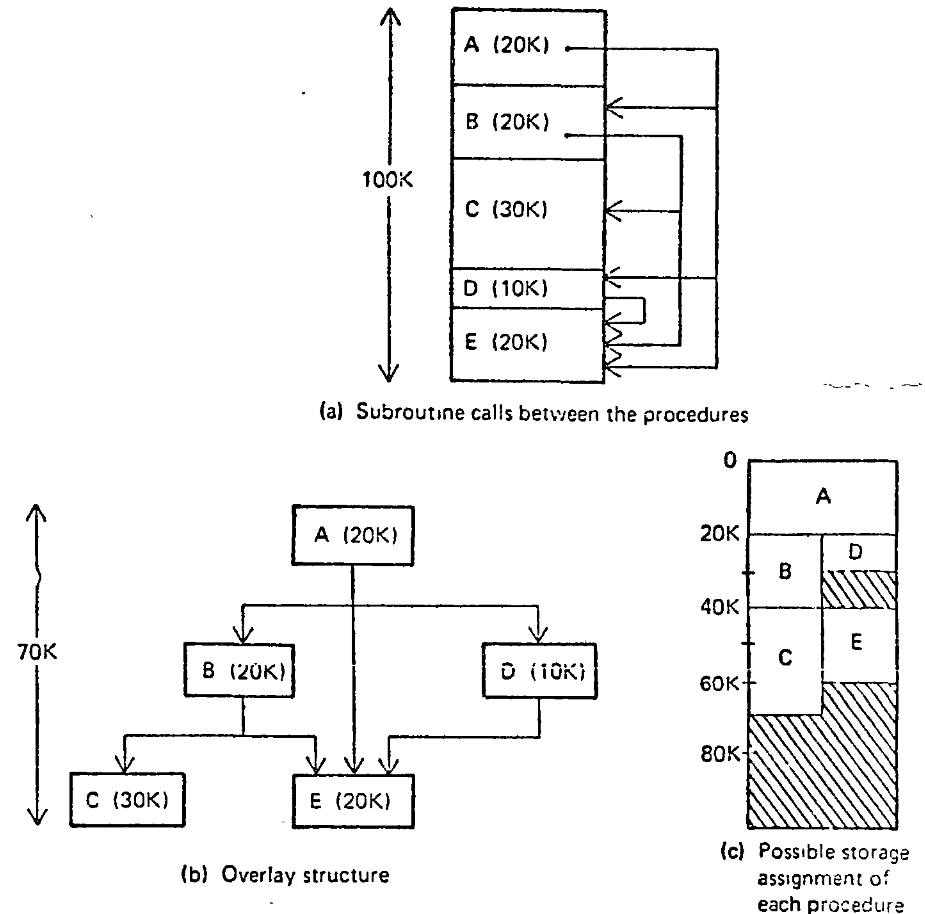


FIGURE 5.9 Other loading schemes

simply the *flipper*. This overall scheme is called *dynamic loading* or *load-on-call* (LOCAL).

DYNAMIC LINKING

A major disadvantage of all of the previous loading schemes is that if a subroutine is referenced but never executed (e.g., if the programmer had placed a call statement in his program but this statement was never executed because of a condition that branched around it), the loader would still incur the overhead of linking the subroutine.

Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called. It is not possible to write programs as follows:

```
     :
READ      SUBNAME, ARGUMENT
ANSWER  = SUBNAME(ARGUMENT)
PRINT     ANSWER
     :
```

where the name of the subroutine (e.g., SQRT, SINE, etc.) is an input parameter, SUBNAME, just like the other data.

A very general type of loading scheme that we will discuss in Chapter 9 is called dynamic linking. This is a mechanism by which loading and linking of external references are postponed until execution time. That is, the assembler produces text, binding, and relocation information from a source language deck. The loader loads only the main program. If the main program should execute a transfer instruction to an external address, or should reference an external variable (that is, a variable that has not been defined in this procedure segment), the loader is called. Only then is the segment containing the external reference loaded.

An advantage here is that no overhead is incurred unless the procedure to be called or referenced is actually used. A further advantage is that the system can be dynamically reconfigured. The major drawback to using this type of loading scheme is the considerable overhead and complexity incurred, due to the fact that we have postponed most of the binding process until execution time.

5.2 DESIGN OF AN ABSOLUTE LOADER

We introduce the general topic of loader design by presenting a design of an absolute loader.

With an absolute loading scheme the programmer and the assembler perform the tasks of allocation, relocation, and linking. Therefore, it is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.

There are two types of information that the object deck must communicate from the assembler to the loader. First, it must convey the machine instructions that the assembler has created along with the assigned core locations. Second, it must convey the entry point of the program, which is where the loader is to transfer control when all instructions are loaded. Assuming that this information is transmitted on cards, a possible format is shown in Figure 5.10.

Note that in the card format shown the instructions are stored on the card as one core byte per column. For each of the 256 possible contents of an eight-bit

byte there is a corresponding punched card code (e.g., hexadecimal 00 is a column punched with five holes, 12-0-1-8-9, whereas a hexadecimal F1 is a column with a single punch in row 1). Thus, when a card is read, it is stored in core as 80 contiguous bytes.

Text cards (for instructions and data)

Card column	*Contents*
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Address at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

Transfer cards (to hold entry point to program)

Card column	*Contents*
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Address of entry point
6-72	Empty
73-80	Card sequence number

FIGURE 5.10 Card formats for an absolute loader

The algorithm for an absolute loader is quite simple. The object deck for this loader consists of a series of text cards terminated by a transfer card. Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached. At this point the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card. A flowchart for this process is illustrated in Figure 5.11.

5.3 DESIGN OF A DIRECT-LINKING LOADER

In this section a design of an IBM 360-type direct-linking loader is presented. Certain obscure features (primarily related to the IBM PL/I implementation and overlay structures) have been omitted, and where alternative formats are possible, only the simplest is given.

The design steps followed will parallel those taken in the design of an assembler (Chapter 3). Note that because the direct-linking loader needs to know

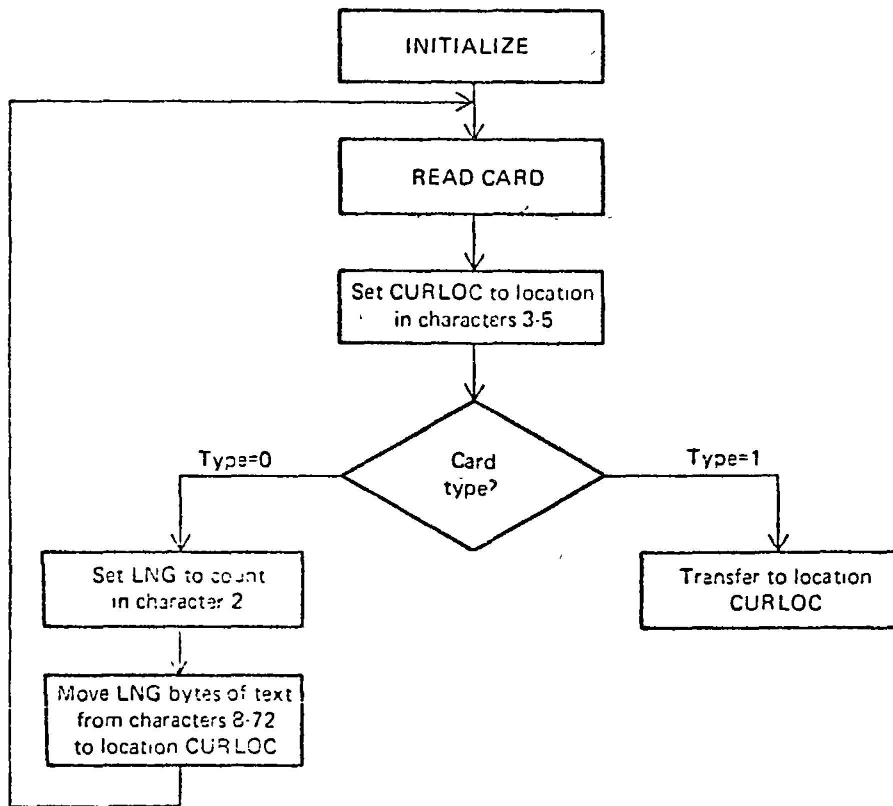


FIGURE 5.11 Absolute loader

the absolute (load time) values of some external symbols before it can perform the modifications on address constants, it requires two passes.

5.3.1 Specification of Problem

The organization of the IBM 360 facilitates the tasks to be performed by its relocating loader. On the IBM 7094, a direct access machine, it was necessary to relocate the address portion on almost all instructions. In the 360, instruction relocation is accomplished by the use of the base register, which is set by neither the assembler nor the loader. Therefore, the 360 relocating loader can treat instructions *exactly* like nonrelocatable data (full word constants, characters, etc.). However, address constants must still be relocated.

For example, the following instructions:


```
TEST     START
         USING    *,15
         L        1,DATA
           :
DATA     DC       F'5'
         END
```

might be assembled as:

Rel. loc.	*Instruction/data*
0	L 1,960(0,15).
:	:
96	5

Regardless of where the program is loaded, the L instruction will be unchanged as long as DATA remains 96 bytes from the beginning of the program. The contents of the base register (15) obviously will be different, depending upon the program's load location.

On the other hand, consider modifying the above example:

```
              :
DATA        DC     F'5'
DATALOC     DC     A(DATA)
            END
```

DATALOC must contain the absolute address of DATA. The assembler knows only that DATA is 96 bytes from the beginning of the program, so the loader must add to this the load address of the program in finding the actual absolute address to be contained in DATALOC.

Let us clarify the scope of the address constant problem. An address constant may be (1) absolute; (2) simple relocatable; or (3) complex relocatable.

For example, the address constant A(LOC1-LOC2) will be:

1. *Absolute* if LOC1 and LOC2 are two relocatable symbols defined internal to program – the assembler can calculate the actual value, their difference.
2. *Simple relocatable* if LOC1 is a relocatable symbol within this procedure and LOC2 is an absolute number (e.g., LOC2 EQU 5). The assembler can calculate the difference between relative location of LOC1 and value of LOC2, but the loader must perform relocation by adding the program load address.
3. *Complex relocatable* if LOC1 and LOC2 are entries to some other program. The assembler can do nothing, and the value must be calculated by the loader.

The 360 direct-linking loader processes programs generated by the assembler, FORTRAN compiler, or PL/I compiler. Recall that neither the original source program nor the assembler symbol table is available to the loader. Therefore, the object deck must contain all information needed for relocation and linking.

There are four sections to the object deck (and four corresponding card formats):

1. External Symbol Dictionary cards (ESD)
2. Instructions and data cards, called "text" of program (TXT)
3. Relocation and Linkage Directory cards (RLD)
4. End card (END)

The ESD cards contain the information necessary to build the external symbol dictionary or symbol table. External symbols are symbols that can be referred beyond the subroutine level. The normal labels in the source program are used only by the assembler, and information about them is not included in the object deck.

EXAMPLE

Assume program B has a table called NAMES; it can be accessed by program A as follows.

```
A          START
           EXTRN    NAMES
             :
           L        1,ADDRNAME get address of NAME table
             :
ADDRNAME   DC       A(NAMES)
           END
-----------------------------
B          START
           ENTRY    NAMES
             :
NAMES      DC       ----
           END
```

There are three types of external symbols, as illustrated in the above:

1. *Segment Definition* (SD) – name on START or CSECT card.
2. *Local Definition* (LD) – specified on ENTRY card. There must be a label in same program with same name.
3. *External Reference* (ER) – specified on EXTRN card. There must be a corresponding ENTRY START or CSECT card in another program with same name.

Each SD and ER symbol is assigned a unique number (e.g., 1,2,3, . . .) by the assembler. This number is called the symbol's *identifier*, or *ID*, and is used in conjunction with the RLD cards.

The TXT cards contain blocks of data and the relative address at which the data is to be placed. Once the loader has decided where to load the program, it merely adds the *Program Load Address* (PLA) to the relative address and moves the data into the resulting location. The data on the TXT card may be instructions, nonrelocated data, or initial values of address constants.

EXAMPLE

```
Relative address          Instruction
      A                   START
                          EXTRN    NAMES
                          USING    *,15
                            :
40                        L        1,ALPHA
44                        BCR      15,14
46                        ----------------    (Skipped by assembler)
48    ALPHA               DC       F'5'
52    ALLOC               DC       A(ALPHA)
56    ADDRNAME            DC       A(NAMES)
                            :
```

The TXT card produced is:

Relative address = 40
Data portion = *58 10 F0 48 07FE XX XX 00 00 00 05 00 00 00 48 00 00 00 00*
Length of data portion = 20 bytes

The RLD cards contain the following information.

1. The location and length of each address constant that needs to be changed for relocation or linking
2. The external symbol by which the address constant should be modified (added or subtracted)
3. The operation to be performed (add or subtract)

Rather than using the actual external symbol's name on the RLD card, as implied in section 5.1 6 and Figure 5.8, the external symbol's identifier, or ID, is used There are various reasons for this, the major one probably being that the ID is only a single byte long, compared to the eight bytes occupied by the

symbol name, so that a considerable amount of space is saved on the RLD cards. Unfortunately, this space-saving technique causes increased loader complexity as will be shown later.

The preceding program segment used as an example for TXT cards would result in the following RLD cards

ID	*Flag*	*Length*	*Rel. loc.*
01	+	4	52
02	+	4	56

if we assume that A's assigned ID is 01 and NAMES' assigned ID is 02. This RLD information tells the loader to add the absolute load address of A to the contents of relative location 52 and then add the absolute load address of NAMES to the contents of relative location 56.

The END card specifies the end of the object deck. If the assembler END card has a symbol in the operand field, it specifies a start of execution point for the entire program (all subroutines). This address is recorded on the END card.

There is a final card required to specify the end of a collection of object decks. The 360 loaders usually use either a loader terminate (LDT) or End of File (EOF) card.

Subroutine A	ESD TXT RLD END
Subroutine B	ESD TXT RLD END
Subroutine C	ESD TXT RLD END
	EOF or LDT

The simple programs PG1 and PG2 in Figure 5.12 illustrate a wide range of relocation and linking situations. Figures 5.13 and 5.14 display the ESD, TXT, and RLD cards produced by the assembler for PG1 and PG2, respectively. Finally, Figure 5.15 depicts the contents of main storage after the programs have been allocated space, relocated, linked, and loaded. The reader should examine these figures carefully and validate the correctness and reasons for each value.

A few specific points in these examples should be noted. Both PG1 and PG2 contain an address constant of the form A(PG1ENT2-PG1ENT1-3). It should be

Source card reference	*Relative address*	*Sample program (source deck)*			
1	0	PG1	START		(a) Procedure PG1
2			ENTRY	PG1ENT1,PG1ENT2	
3			EXTRN	PG2ENT1,PG2	
4	20	PG1ENT1	—		
5	30	PG1ENT2	—		
6	40		DC	A(PG1ENT1)	
7	44		DC	A(PG1ENT2+15)	
8	48		DC	A(PG1ENT2-PG1ENT1-3)	
9	52		DC	A(PG2)	
10	56		DC	A(PG2ENT1+PG2-PG1ENT1+4)	
11			END		
12	0	PG2	START		(b) Procedure PG2
13			ENTRY	PG2ENT1	
14			EXTRN	PG1ENT1,PG1ENT2	
15	16	PG2ENT1	—		
16	24		DC	A(PG1ENT1)	
17	28		DC	A(PG1ENT2+15)	
18	32		DC	A(PG1ENT2-PG1ENT1-3)	
19			END		

FIGURE 5.12 Sample procedures PG1 and PG2

reassuring to note in Figure 5.15 that both instances of this address constant (location 148, 196) have the same value – 7. Since both PG1ENT2 and PG1ENT1 are symbols internal to PG1, the assembler, processing PG1, can compute the entire expression and determine the value of 7. We see in Figure 5.13 that the TXT card for location 48-51 contains the 7 and there are no associated RLD cards for this address constant. On the other hand, these symbols are external to PG2; thus, the assembler, while processing PG2, has no means of evaluating the address constant. This is illustrated in Figure 5.14. The TXT card for relative locations 32-35 contains a -3, the only part of the address constant that can be calculated by the assembler. The last two RLD cards tell the loader to add the valud of ID 03, which is PG1ENT2, to locations 32-35 and then subtract the value of ID 02, which is PG1ENT1. When processed by the loader, this address constant in PG2 will indeed have the same value as the one in PG1.

Since the direct-linking loader may encounter external references in an object deck which cannot be evaluated until a later object deck is processed, this type of loader requires two passes. Their functions are very similar to those of the two passes of an assembler. The major function of pass 1 of a direct-linking loader is to allocate and assign each program a location in core and create a

ESD cards					
Source card reference	Name	Type	ID	Relative address	Length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2	ER	02	—	—
3	PG2ENT1	ER	03	—	—

TXT cards (only the interesting ones, i.e. those involving address constants)			
Source card reference	Relative address	Contents	Comments
6	40-43	20	
7	44-47	45	= 30 + 15
8	48-51	7	= 30-20-3
9	52-55	0	unknown to PG1
10	56-59	-16	= -20 + 4

RLD cards				
Source card reference	ESD ID	Length (bytes)	Flag + or −	Relative address
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	−	56

FIGURE 5.13 Object deck program PG1

symbol table filling in the values of the external symbols. The major function of pass 2 is to load the actual program text and perform the relocation modification of any address constants needing to be altered.

The first pass allocates and assigns storage locations to all segments and stores the values of all external symbols in a symbol table. These external symbols appear as local definitions on the ESD cards of another assembled program. For every external reference symbol there must be a corresponding internal symbol in some other program. The loader inserts the absolute address of all of these external symbols in the symbol table. In the second pass, the loader places the text into the assigned locations and performs the relocation task, modifying relocatable constants. Figure 5.16 depicts the interplay between the passes of a loader in a direct-linking loading scheme.

ESD cards					
Source card reference	Name	Type	ID	ADDR	Length
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

TXT cards (only the interesting ones)		
Source card reference	Relative address	Contents
16	24-27	0
17	28-31	15
18	32-25	-3

RLD cards				
Source card reference	ESD ID	Length flag (bytes)	Flag + or −	Relative address
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	−	32

FIGURE 5.14 Object deck program PG2

5.3.2 Specification of Data Structures

The next step in our design procedure is to identify the data bases required by each pass of the loader.

Pass 1 data bases:

1. Input object decks.
2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system, that specifies the address to load the first segment.
3. A Program Load Address (PLA) counter, used to keep track of each segment's assigned location.
4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.
6. A printed listing, the *load map*, that specifies each external symbol and its assigned value.

Assume

PG1 loaded at location 104
PG2 loaded at location 168

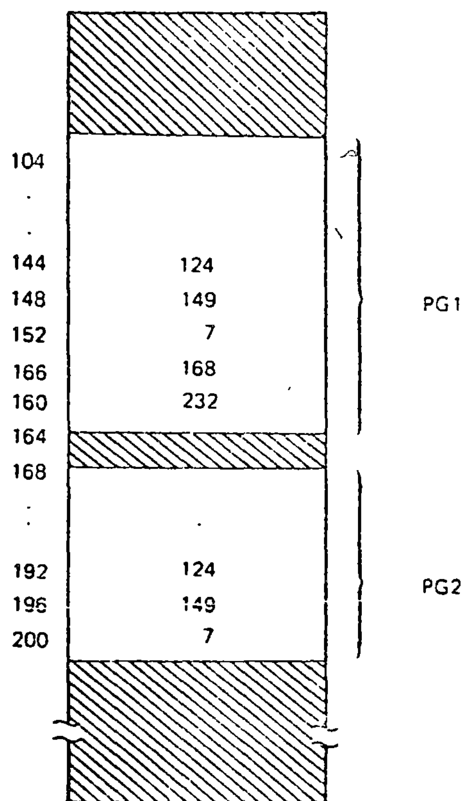


FIGURE 5.15 Main storage after loading programs PG1 and PG2

Pass 2 data bases.

1. Copy of object programs inputted to pass 1
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address counter (PLA)
4. The Global External Symbol Table (GEST), prepared by pass 1, containing each external symbol and its corresponding absolute address value
5. An array, the Local External Symbol Array (LESA), which is used to establish a correspondence between the ESD ID numbers, used on ESD and RLD cards, and the corresponding external symbol's absolute address value

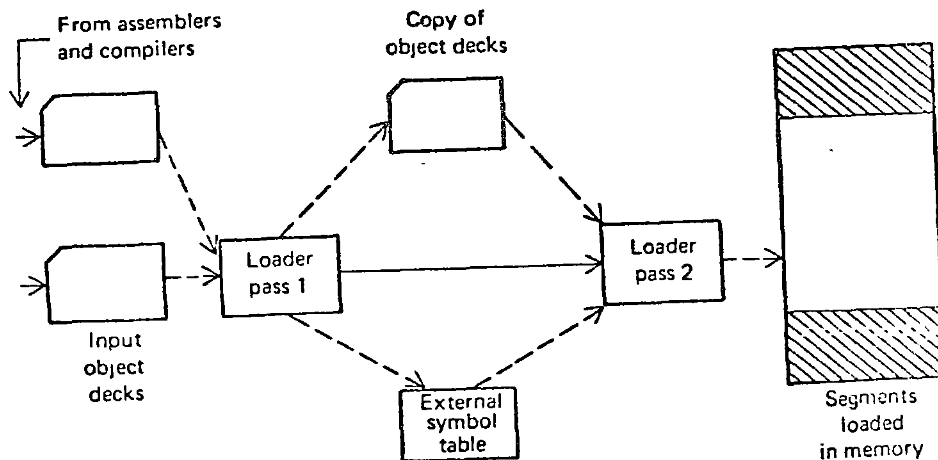


FIGURE 5.16 Two pass direct-linking loader scheme

5.3.3 Format of Data Bases

The third step in our design procedure is to specify the format and content of each of the data bases. The major data bases are depicted in Figure 5.17.

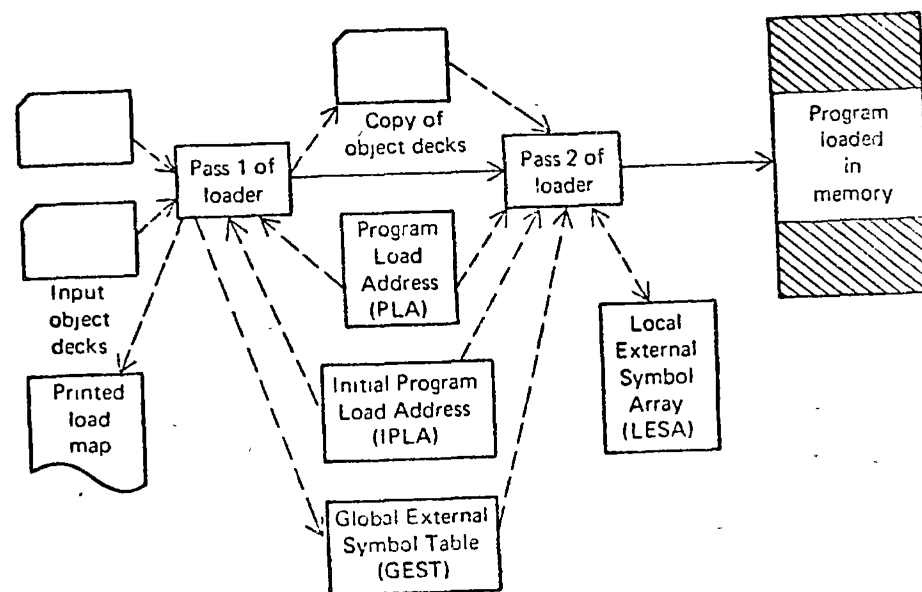


FIGURE 5.17 Use of data bases by loader passes

OBJECT DECK

The object deck has been discussed several times: Figures 5.18, 5.19, 5.20 and 5.21, depict in detail the actual card deck format that is used by various IBM 370 or 360 direct-linking loaders. The specific card format is not crucial, but these figures present a good example of the various techniques used to encode information.

GLOBAL EXTERNAL SYMBOL TABLE

The Global External Symbol Table (GEST) is used to store the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card. When these symbols are encountered during pass 1, they are assigned an absolute core address; this address is stored, along with the symbol, in the GEST as illustrated in Figure 5.22.

The reader may wish to review the discussion on symbol tables and searching/sorting techniques presented in Chapter 3 in conjunction with the design of an

Columns	Contents
1	Hexadecimal byte X'02' (card punch 12-2-9)
2-4	Characters ESD
5-14	Blank
15-16	ESD identifier (ID) for program name (SD) or external symbol (ER) (IDs unique within program) or blank for entry (LD), see box below
17-24	Name, padded with blanks
25	ESD type code (TYPE)
26-28	Relative address or blank (ADDR), see box below
29	Blank
30-32	Length of program otherwise blank (LENGTH)
33-72	Blank
73-80	Card sequence number

TYPE	*Hexadecimal code*
SD	01
LD	02
ER	03

ESD forms and conventions

	Type	*ID*	*ADDR*	*Length*
Program name (segment definition)	SD	01	Zero	Length of program
Entry (local definition)	LD	—	Relative address	—
External reference	ER	Unique number	—	—

(The unique ID numbers are usually assigned sequentially.)

FIGURE 5.18 ESD card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters TXT
5	Blank
6-8	Relative address of first data byte (ADDR)
9-10	Blank
11-12	Byte Count (BC) = number of bytes of information in cc. 17-72
13-16	Blank
17-72	From 1 to 56 data bytes (instructions and "data" look the same)
73-80	Card sequence number

FIGURE 5.19 TXT card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters RLD
5-18	Blank
19-20	ID corresponding to a number assigned to SD or ER on ESD card
21	Flag byte (see box below)
22-24	Relative address of first byte of address constant (ADDR)
25-72	Blank
73-80	Card sequence number

Flag byte conventions

bits	
0-3	Not used
4-5	Length (in bytes) of address constant 00 = one byte 01 = two bytes 10 = three bytes 11 = four bytes
6	0 means add ESD address to address constant 1 means subtract ESD address from address constant

FIGURE 5.20 RLD card format

Columns	*Contents*
1	Hexadecimal byte X'02'
2-4	Characters END
5	Blank
6-8	Start of execution entry (ADDR), if other than beginning of program (specified on assembly END card)
9-72	Blank
73-80	Card sequence number

FIGURE 5.21 END card format

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1bbbbb"	100
"PG1ENT1b"	120
"PG1ENT2b"	130
"PG2bbbbb"	164
"PG2ENT1b"	180

(12 bytes per entry)

Notation This sample GEST content is based upon the example in Figures 5.12 and 5.15.

FIGURE 5.22 Global External Symbol Table (GEST) format

assembler. The GEST has the same general use and characteristics as the assembler's Symbol Table.

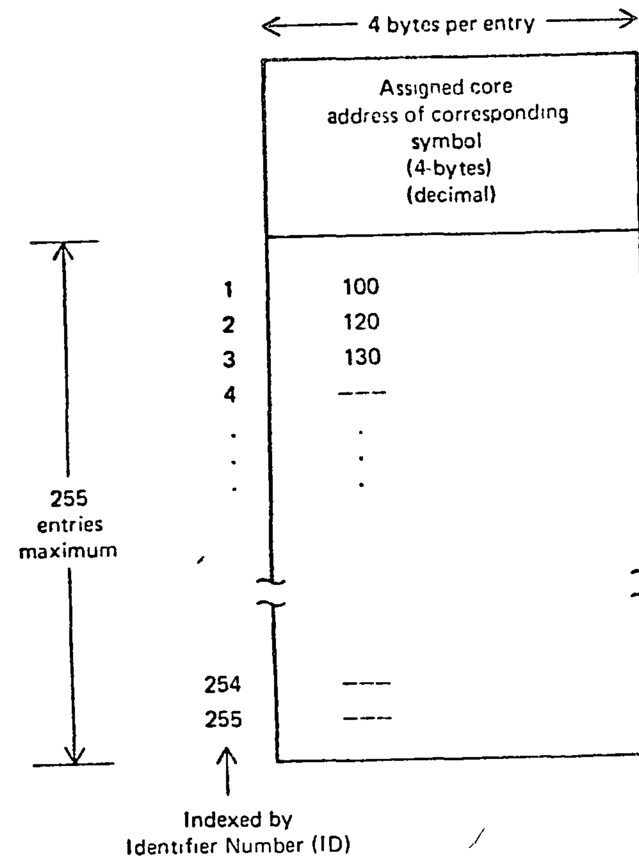
LOCAL EXTERNAL SYMBOL ARRAY

As mentioned earlier, the external symbol to be used for relocation or linking is identified on the RLD cards by means of an ID number rather than the symbol's name. The ID number must match an SD or ER entry on the ESD cards. This technique both saves space on the RLD cards and speeds processing by eliminating many searches of the Global External Symbol Table.

It is necessary to establish a correspondence between an ID number on an RLD card and the absolute core address value. The ESD cards contain the ID numbers and the symbols they correspond to, while the information relating these symbols to absolute core address values may be found in the GEST. In pass 2 of the loader the GEST and ESD information for each individual object deck is merged to produce the local external symbol array that directly relates ID number and value. In principle it is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment. Figure 5.23 depicts the format of the Local External Symbol Array (LESA). Note than unlike the case with the GEST, it is not necessary to *search* the LESA, given an ID number, the corresponding value is written as LESA(ID) and can be immediately obtained.

5.3.4 Algorithm

The following two flowcharts (Figs. 5.24 and 5.25) describe an algorithm for a direct-linking loader for an IBM System/360-type computer. While they illustrate



Notation This sample LESA contents is based upon the PG1 segment as presented in Figures 5.12, 5.13, and 5.15.

FIGURE 5.23 Local External Symbol Array (LESA) format

most of the logical processes involved, these flowcharts are still a simplification of the operations performed in a complex loader. In particular, many "special" features, such as COMMON segments, library processing, dynamic loading and dynamic linking, are not explicitly included (many of them are discussed in the problems section and in later chapters).

Pass 1 – allocate segments and define symbols The purpose of the first pass is to assign a location to each segment, and thus to define the values of all external symbols. Since we wish to minimize the amount of core storage required for the total program, we will assign each segment the next available loca[illegible]

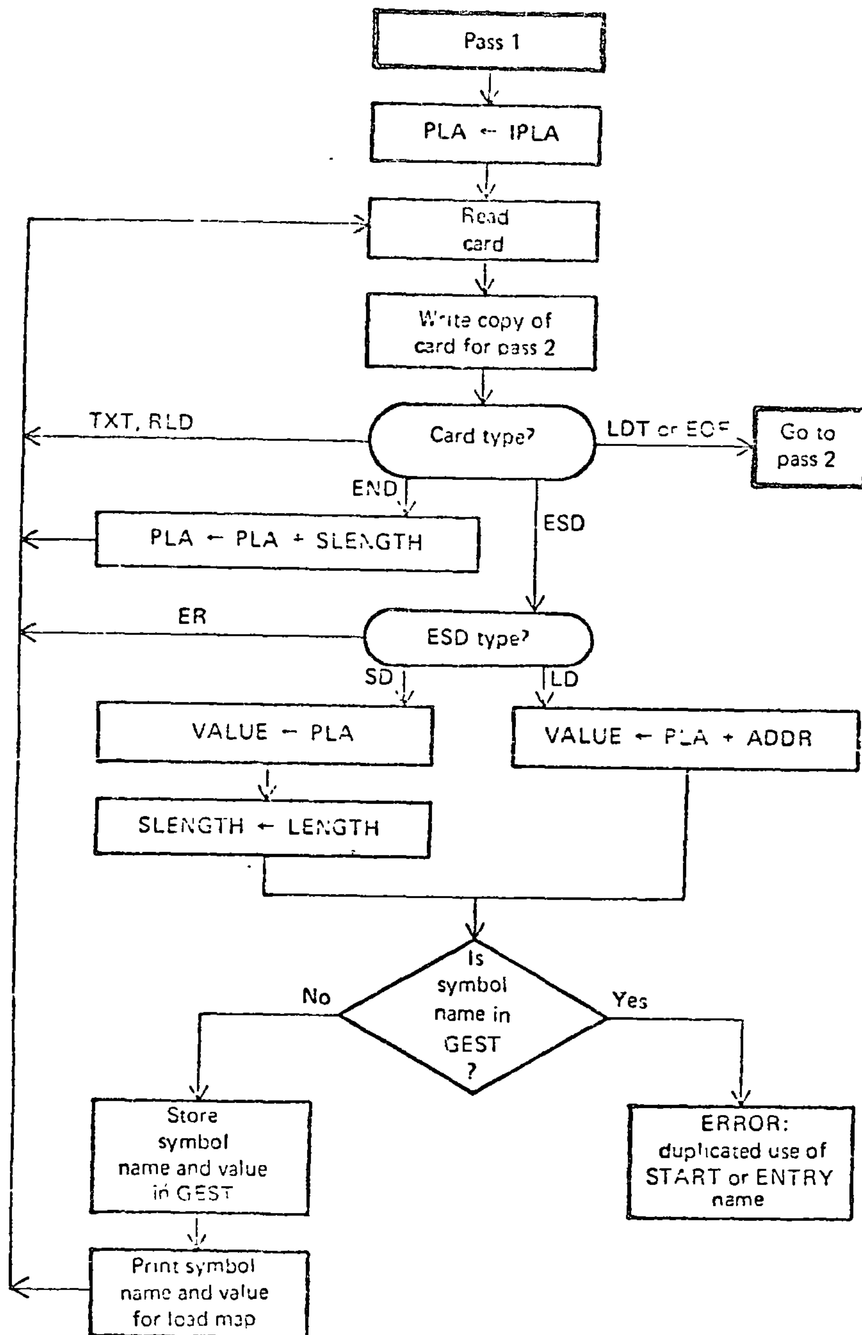


FIGURE 5.24 Detailed pass 1 flowchart

preceding segment. It is necessary for the loader to know where it can load the first segment. This address, the Initial Program Load Address (IPLA), is normally determined by the operating system. In some systems the programmer may specify the IPLA; in either case we will assume that the IPLA is a parameter supplied to the loader.

Initially, the Program Load Address (PLA) is set to the Initial Program Load Address (IPLA). An object card is then read and a copy written for use by pass 2. The card can be one of five types, ESD, TXT, RLD, END, or LDT/EOF. If it is a TXT or RLD card, there is no processing required during pass 1 so the next card is read. An ESD card is processed in different ways depending upon the type of external symbol, SD, LD or ER. If a segment definition ESD card is read, the length field, LENGTH, from the card is temporarily saved in the variable, SLENGTH. The value, VALUE, to be assigned to this symbol is set to the current value of the PLA. The symbol and its assigned value are then stored in the GEST; if the symbol already existed in the GEST, there must have been a previous SD or LD ESD with the same name – this is an error. The symbol and its value are printed as part of the load map. A similar process is used for LD symbols; the value to be assigned is set to the current PLA plus the relative address, ADDR, indicated on the ESD card. The ER symbols do not require any processing during pass 1. When an END card is encountered, the program load address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment. When the LDT or EOF card is finally read, pass 1 is completed and control transfers to pass 2.

Pass 2 – load text and relocate/link address constants After all the segments have been assigned locations and the external symbols have been defined by pass 1, it is possible to complete the loading by loading the text and adjusting (relocation or linking) address constants. At the end of pass 2, the loader will transfer control to the loaded program. The following simple rule is often used to determine where to commence execution:

1. If an address is specified on the END card, that address is used as the execution start address.
2. Otherwise, execution will commence at the beginning of the first segment

At the beginning of pass 2 the program load address is initialized as in pass 1, and the execution start address (EXADDR) is set to IPLA. The cards are read one by one from the object deck file left by pass 1. Each of the five types of cards is processed differently, as follows.

ESD CARD

Each of the ESD card types is processed differently.

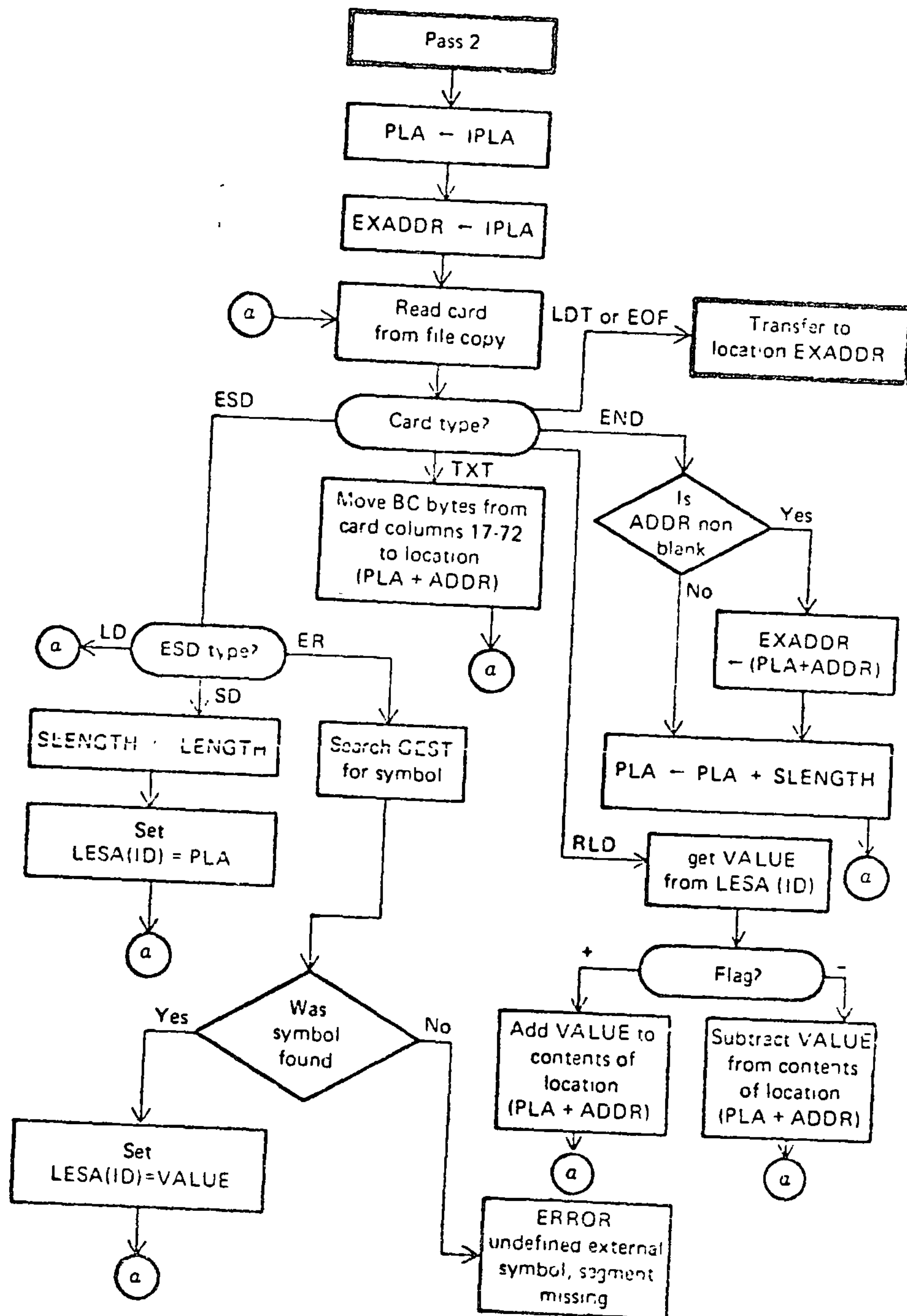


FIGURE 5.25 Detailed pass 2 flowchart

SD-type ESD The LENGTH of the segment is temporarily saved in the variable SLENGTH. The appropriate entry in the local external symbol array, LESA(ID), is set to the current value of the Program Load Address.

LD-type ESD The LD-type ESD does not require any processing during pass 2.

ER-type ESD The Global External Symbol Table (GEST) is searched for a match with the ER symbol. If it is not found, the corresponding segment or entry must be missing – this is an error. If the symbol is found in the GEST, its value is extracted and the corresponding Local External Symbol Array entry, LESA(ID), is set equal to it.

TXT CARD
When a TXT card is read, the text is copied from the card to the appropriate relocated core location (PLA + ADDR).

RLD CARD
The value to be used for relocation and linking is extracted from the local external symbol array as specified by the ID field, i.e., LESA(ID). Depending upon the flag setting (plus or minus) the value is either added to or subtracted from the address constant. The actual relocated address of the address constant is computed as the sum of the PLA and the ADDR field specified on the RLD card.

END CARD
If an execution start address is specified on the END card, it is saved in the variable EXADDR after being relocated by the PLA. The Program Load Address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment.

LDT/EOF CARD
The loader transfers control to the loaded program at the address specified by current contents of the execution address variable (EXADDR).

5.4 SUMMARY

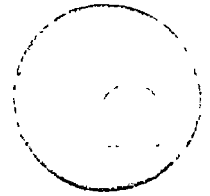
The four basic functions of a loader are allocation, linking, relocation, and loading. The various types of loaders (e.g., "compile-and-go," absolute, relocating, direct-linking, dynamic loading, and dynamic linking) differ primarily in the manner in which the four basic functions are accomplished.

A typical direct-linking loader requires two passes. The first pass allocates space for the segments and defines the values of the external symbols. The

second pass actually loads the text and uses the global external symbol table, produced by pass 1, to relocate and link the address constants.

Although their purposes are quite different, the design of the direct-linking loader has many similarities to the design of an assembler. In particular, the use of a symbol table is important in both cases.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



APLICACION DE MINICOMPUTADORAS

EXPOSICION DEL ING. RAYMUNDO SEGOVIA

diciembre 1975.

PALACIO DE MINERIA
Tacuba 5, primer piso. México 1, D. F.
TELEFONOS: 513-27-95
512-31-23 521-73-35

Third Generation Computer Systems

PETER J. DENNING

Princeton University, Princeton, New Jersey 08540*

The common features of third generation operating systems are surveyed from a general view, with emphasis on the common abstractions that constitute at least the basis for a "theory" of operating systems. Properties of specific systems are not discussed except where examples are useful. The technical aspects of issues and concepts are stressed, the nontechnical aspects mentioned only briefly. A perfunctory knowledge of third generation systems is presumed.

Key words and phrases: multiprogramming systems, operating systems, supervisory systems, time-sharing systems, programming, storage allocation, memory allocation, processes, concurrency, parallelism, resource allocation, protection

CR categories: 1.3, 4.0, 4.30, 6.20

NTRODUCTION

It has been the custom to divide the era of electronic computing into "generations" whose approximate dates are:

First: 1940–1950;
Second: 1950–1964;
Third: 1964-present; and
Late Third: 1968-present.

The principal properties of the generations are summarized in Table I. The term "generation" came into wide use after 1964, the year in which third generation machines were announced. Although the term was originally used to suggest differences in hardware technology, it has come to be applied to the entire hardware/software system rather than to the hardware alone [R3, R4].

A Definition of "Operating System"

As will be discussed in detail below, the term "process" is used to denote a program in execution. A computer system may be defined in terms of the various supervisory and control functions it provides for the processes created by its users: 1) creating and removing processes; 2) controlling the progress of processes—i.e., ensuring that each logically enabled process makes progress at a positive rate and that no process can indefinitely block the progress of others; 3) acting on exceptional conditions arising during the execution of a process—e.g., arithmetic or machine errors, interrupts, addressing snags, illegal and privileged instructions, or protection violations; 4) allocating hardware resources among processes; 5) providing access to software resources—e.g., files, editors, compilers, assemblers, subroutine libraries, and programming systems; 6) providing protection, access control, and security for information; and 7) providing interprocess communications where required. These functions must be provided by the system because they cannot be handled adequately by the processes themselves. The computer system software that assists the hardware in implementing these functions is known as the *operating system*.

Two points about operating systems should be noted. First, users seldom (if ever) perform a computation without assistance from the operating system; thus, they often

* Department of Electrical Engineering. Work reported herein was supported in part by NASA Grant NGR-31-001-170 and by NSF Grant GY-6586.

CONTENTS

Introduction 175–182
A Definition of "Operating System"
Types of Systems
Common Properties of Systems
Concurrency
Automatic Resource Allocation
Sharing
Multiplexing
Remote Conversational Access
Nondeterminacy
Long-Term Storage
Abstractions Common to Systems
Programming 182–193
Systems Programming Languages
Procedures
Storage Allocation 185–193
Motivations for Automatic Memory Management
Virtual Memory
Motivations for Name Management
File Systems
Segmentation
Concurrent Processes 193–201
Determinacy
Deadlocks
Mutual Exclusion
Synchronization
Resource Allocation 201–206
A Resource Allocation Model
System Balance
Choosing a Policy
Protection 206–210
Conclusions 210–211
Annotated Bibliography 212–216

Copyright © 1971, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

come to regard the entire hardware/software system, rather than the hardware alone, as "the machine." Secondly, in extensible systems, such as MULTICS [AS, C5, S1] or the RC-4000 [B4, B5], a user may redefine or add to all but a small nucleus of operating system programs; thus, an operating system need not be fixed or immutable, and each user may be presented with a different "machine."

Types of Systems

An enormous variety of systems are regarded as members of the third generation. The range includes general-purpose programming systems, real-time control systems, time-sharing systems, information service and teleprocessing systems, and computer networks; and noninteractive, large batch-processing systems, such as the Chippewa Operating System on the CDC 6600 or OS on the IBM 360 series. It also includes a wide range of interactive systems, of which there are five increasingly sophisticated categories [D10]: 1) *dedicated information systems*, such as airline and other ticket reservation systems, in which the users may perform a limited number of "transactions" on a given data base; 2) *dedicated interactive systems*, such as Joss (Johnniac Open Shop System) or QUIKTRAN, in which the user may program transactions within a given language; 3) *general-purpose interactive systems*, in which users may write programs in any one of a given set of languages (most time-sharing service bureaus offer systems of this type); 4) *extensible systems*, such as MIT's CTSS (Compatible Time Sharing System) [C4, C8], in which users are not restricted to the programming languages and programming systems provided by the system, but may develop their own and make them available to other users; and 5) *coherent systems*, such as MIT's MULTICS (MULTiplexed Information and Computing Service) [C5], in which one may construct new programs or programming systems from modules by various authors in various languages, without having to know the internal operation of any module.

The views programmers and designers

TABLE I. A SUMMARY OF CHARACTERISTICS OF THE GENERATIONS OF COMPUTERS

Characteristic	*Generations*			
	First	*Second*	*Third*	*Late third*
Electronics:				
Components	vacuum tubes	transistors	integrated circuits	same as third
Time/operation	0.1-1.0 msec	1-10 μsec	0.1-1.0 μsec	same as third
Main memory:				
Components	electrostatic tubes and delay lines	magnetic drum and magnetic core	magnetic core and other magnetic media	semiconductor registers (cache)
Time/access	1 msec	1-10 μsec	0.1-10 μsec	0.1 μsec
Auxiliary memory	paper tape, cards, delay lines	magnetic tape, disks, drums, paper cards	same as second, plus extended core and mass core	same as third
Programming languages and capabilities	binary code and symbolic code	high-level languages, subroutines, recursion	same as second, plus data structures	same as third, plus extensible languages and concurrent programming
Ability of user to participate in debugging and running his program	yes (hands-on)	no	yes (interactive and conversational programs)	same as third
Hardware services and primitives	arithmetic units	floating-point arithmetic, interrupt facilities, microprogramming, special-purpose I/O equipment	same as second, plus microprogramming and read-only storage, paging and relocation hardware, generalized interrupt systems, increased use of parallelism, instruction lookahead and pipelining, datatype control	
Software and other services	none	subroutine libraries, batch monitors, special-purpose I/O facilities	same as second, plus multiaccessing and multiprogramming, time-sharing and remote access, central file systems, automatic resource allocation, relocation and linking, one-level store and virtual memory, segmentation and paging, context editors, programming systems, sharing and protection of information	

take of systems are almost as varied as the systems themselves. Some are viewed as large, powerful batch-processing facilities offering a wide [illegible] of languages, programming system[illegible] services (e.g., IBM's OS/360 and CD[illegible] [illegible]ippewa 6600). Some are viewed as [illegible] environments for certain programm[illegible] languages (e.g., ALGOL on the Burroughs B6500). Some are viewed as extensions of some language or machine (e.g., the virtual machines presented to users in IBM's CP/67 or M44/44X). Others are viewed as information management systems (e.g., SABRE Airline Reservations System). Still others are viewed as extensible systems or information utilities (e.g., MIT's CTSS and MULTICS, A/S Regnecentralen's RC-4000, IBM's 360/TSS, and RCA's Spectra 70/46).

Common Properties of Systems

Despite the diversity of system types and views about them, these systems have a great deal in common. They exhibit common characteristics because they were designed with common general objectives, especially to provide programmers with: 1) an efficient environment for program development, debugging, and execution; 2) a wide range of problem-solving facilities; and 3) low-cost computing through the sharing of resources and information. The characteristics we shall describe below are properties of the class of third generation systems,

but no particular system need exhibit all of them.

Concurrency

A first common characteristic of third generation systems is concurrency—i.e., the existence or potential existence of several simultaneous (parallel) activities or *processes*. The term "process" was introduced in the early 1960s as an abstraction of a processor's activity, so that the concept of "program in execution" could be meaningful at any instant in time, regardless of whether or not a processor was actually executing instructions from a specific program at that instant. The term "process" is now used in a more general sense to denote any computation activity existing in a computer system. Thus, the idea of *parallel processes* can be interpreted to mean that more than one program can be observed between their starting and finishing points at any given time. Processes may or may not be progressing simultaneously; at a given time, for example, we may observe one process running on the central processor, a second suspended awaiting its turn, and a third running on an input/output channel. Processes interact in two ways: indirectly, by competing for the same resources; and directly, by sharing information. When processes are logically independent they interact indirectly, and control of concurrency is normally delegated to the underlying system; but when they are interacting directly, control of concurrency must be expressed explicitly in their implementations.

Since an operating system is often regarded as a control program that regulates and coordinates various concurrent activities, the need for ways of describing concurrent activity at a programming-language level is felt most acutely by systems programmers. There are at least three reasons for this: 1) the demand for rapid response time and efficient equipment utilization has led to various forms of resource sharing, and has created a need for dynamic specification of the resource requirements of a program; 2) widespread use of concurrent activity between the central machine and its peripheral devices already exists; and 3) the desire to share information and to communicate among executing programs has led to the development of message transmission facilities and software mechanisms for stopping a program while it awaits a signal from another.

Automatic Resource Allocation

A second common characteristic of third generation systems is the existence of an automatic resource allocation mechanism with a wide variety of resources. The reasons for central system control of resource allocation include the following. 1) Programmers tend to be more productive when they do not have to be concerned with resource allocation problems in addition to the logical structures of their algorithms. Moreover, programming languages shield programmers from details of machine operation. 2) Due to the unpredictable nature of demands in a system supporting concurrent activity, a programmer is at a disadvantage in making efficient allocation decisions on his own. 3) A centralized resource allocator is able to monitor the entire system and control resource usage to satisfy objectives both of good service and system efficiency.

Several systems have incorporated a very general view of the nature of resources themselves. As we shall see, this view allows one to treat many aspects of system operation as resource allocation problems, and with good results. According to this view, the set of objects that a process may use, and on which its progress depends, is called the *resources* required by the process. The system provides a variety of resource *types*, and there is generally a limited number of *units* of each type available. Examples of resource types include: processors, memories, peripheral devices, data files, procedures, and messages. Examples of resource units include: a processor, a page of memory, a disk or drum track, a file, a procedure, or a message. Some resource types have immediate realizations in terms of hardware (e.g., processor, memory, peripherals), whereas others are realized in software only (e.g., files, procedures, messages). Some resource types are "reusable," a unit of that type being reassignable after its release; whereas other

types are "consumable," meaning they cease to exist after use (e.g., a message or the activation record of a called procedure). All resources have internal "states" that convey information to their users; if the state of a resource is modified during use, a unit of that type is said to be "subject to exclusive control" by, at most, one process at a time. A unit resource is "preemptible" only if the system may intervene and release it without losing input, output, or progress of the process involved. A non-preemptible resource may be released only by the process to which it is assigned.

Sharing

A third common characteristic of third generation systems is sharing, the simultaneous use of resources by more than one process. The term "sharing" has two related meanings in today's parlance. First, it denotes the fact that resource types can be shared regardless of whether or not individual units of that type can be shared. In fact, most units of physical resource types are subject to exclusive control by the processes to which they are assigned, and some form of multiplexing must be used to implement the sharing, but more will be said about this shortly. Secondly, the term "sharing" refers to the sharing of information, potentially or actually. The desire to share information is motivated by three objectives:

1) *Building on the work of others:* the ability to use, in one's own programs, subprograms or program modules constructed by others.
2) *Shared-data problems:* the sharing by many users of a central data base, which is required by certain types of problems (e.g., ticket reservations).
3) *Removing redundancy:* as many system software resources (compilers or input/output routines, for example) may be required simultaneously by several active processes, and as giving each process its own copy would tend to clutter the already scarce memory with many copies of the same thing, it is desirable that the system provide one shared copy of the routine for all to use.

Achieving these objectives places new requirements on the system and its compilers. Point 1 requires that compilers provide linkage information with compiled modules, and that there be loaders to establish intermodular linkages at the proper time. Point 2 requires some sort of "lockout" mechanism* so that no process could attempt to write or read a part of the data base that is being modified by another process. A program module may satisfy point 3 by being "serially reusable": i.e., the compiler would have to have inserted a "prologue" into it which would initialize the values of any internal variables that might have been changed by a previous use of the module; or a lockout mechanism may be required to prevent a second process from using the module before a first has finished.

To obtain sharable (not merely reusable) subprograms, the compiled code must be partitioned into "procedure" and "data." The procedure part is read-only, and contains all the instructions of the program. The data part contains all the variables and operands of the program and may be read or written. Each process using such code may be linked to a single, common copy of the procedure part, but must be provided with its own private copy of the data part. The techniques for compiling code in this manner were originally devised for the ALGOL programming language to implement recursive procedures (i.e., procedures that can call themselves). Since a recursive subroutine can be called (activated) many times before it executes a single return, and since each call must be provided with parameter and internal variable values different from other calls, it is necessary to associate private working storage (an activation record) with each instance of an activated procedure. (More will be said about this in the next section.) At any given time, one may inspect an ALGOL program in execution and find many instances of procedure activations, thus, there is an immediate analogy between this notion and the notion of parallel processes discussed above. For this reason, many of

* Details of lockout mechanisms will be discussed in connection with the mutual exclusion problem in the fourth section, "Concurrent Processes."

the techniques for handling multiple activations of a procedure in a single program (recursion) were found to be immediately applicable to multiple activations of a procedure among distinct programs (sharing). Because the names and memory requirements of all procedures that might be used by a process may not be known in advance of execution (as is the case with the procedures in ALGOL programs), the mechanism for linking procedures and activation records among processes can be more complicated and may require special hardware to be efficient [D1].

Multiplexing

A fourth common characteristic of third generation systems is multiplexing, a technique in which time is divided into disjoint intervals, and a unit of resource is assigned to, at most, one process during each interval [S1]. As mentioned above, multiplexing is necessitated by the desire to share a given resource type when its individual units must be controlled exclusively by processes to which they are assigned. Multiplexing has assumed particular importance as a means, not only of maintaining high load factors on resources, but also of reducing resource-usage costs by distributing them among many users. The time intervals between the instants of reassignment of the multiplexed resource may be defined naturally (by the alternation of a process between periods of demand and nondemand) or artificially (by means of "time slicing" and preemption). The latter method is used primarily in time-sharing and other systems in which response-time deadlines must be satisfied. Although multiplexing is not sharing, it can be used to give the appearance of sharing; if, for example, a processor is switched equally and cyclically among n programs with sufficient speed, a human observer could not distinguish that system from one in which each program has its own processor of $1/n$ speed.

Let us digress briefly to describe some specific examples of commonly used techniques for sharing and multiplexing.

1) Under *multiprogramming*, (parts of) several programs are placed in main memory at once; this not only makes better use of main memory, but it maintains a supply of executable programs to which the processor may switch should the program it is processing become stopped. Moreover, a particular user's program need not reside continuously in memory during the entire length of its run time; indeed, it may be swapped repeatedly for some other program requiring use of the memory--one that is residing in auxiliary memory waiting its turn. Swapping takes place without any overt action by the programs in memory; to the user its only observable effect is that the machine may appear to be operating more slowly than if he had it entirely to himself. Implementing multiprogramming requires two fundamental alterations in system organization from second generation systems. (a) since programs cannot be allowed to reference regions of memory other than those to which they have been granted access, and since the exact region of memory in which a program may be loaded cannot be predicted in advance, programming must be location-independent; and (b) special mechanisms are required to preempt the processor from one program and switch it to another without interfering with the correct operation of any program.
2) Under *multiaccessing*, many users are permitted to access the system (or parts of it) simultaneously. A simple multiaccess system is the batch-processing system with several remote-job-entry terminals. An advanced multiaccess system is the time-sharing system ("time multiplexing" would be more accurate), in which many users at many consoles use system resources and services conversationally. It should be noted that multiaccessing and multiprogramming are independent concepts; many CDC 6600 batch-processing installations are examples of multiprogrammed systems with one job-entry terminal, whereas MIT's CTSS [C4] is an example of a monoprogrammed time-sharing system that uses swapping to implement memory sharing.
3) *Multitasking* refers to the capability of a system to support more than one active process. Multitasking must necessarily

exist in multiprogramming systems, but it may or may not exist in multiaccess systems. It may also exist in batch-processing systems under the auspices of certain programming languages, such as PL/I.

4) Under *multiprocessing*, the system supports several processors so that several active processes may be in execution concurrently. If one regards channels and other input/output controllers as special-purpose processors, then every third generation system (and many second generation systems as well) is a multiprocessor system. However, the term "multiprocessor" is normally applied only to systems having more than one central processor; examples of such systems are the ILLIAC IV, MULTICS, the Bell System's ESS (Electronic Switching System), and the Michigan Time Sharing System on the IBM 360/67.

Remote Conversational Access

A fifth common characteristic of third generation systems is remote conversational access, in which many users are allowed to interact with their processes. Interactive, or on-line, computing is now known to make rapid program development and debugging possible [D10, W8]. It is required for shared-data-base problems—e.g., ticket reservation systems.

Nondeterminacy

A sixth common characteristic of third generation systems is nondeterminacy, i.e., the unpredictable nature of the order in which events will occur. Nondeterminacy is a necessary consequence of concurrency, sharing, and multiplexing. The order in which resources are assigned, released, accessed, or shared by processes is unpredictable, and the mechanisms for handling concurrency, sharing, and multiplexing must be designed to allow for this. In contrast to global, system-wide nondeterminacy, there is often a need for local forms of determinacy at the programming level. When a collection of processes is cooperating toward a common goal and sharing information, it is usually desirable that the result of their computation depend only on the initial values of the data, not on their relative speeds. A system of processes having this property is sometimes called "determinate" or "speed-independent." This problem is considered again in the fourth section, "Concurrent Processes."

Long-Term Storage

A seventh common property of third generation systems is the presence of long-term storage. Second generation systems provided it in the limited form of subroutine libraries (no provision was made for data files). Since these libraries were limited in size and were accessible only via compilers or loaders, the problems of managing them were not severe. However, many third generation systems endeavor to provide users with means of storing their own information in the system for indefinite periods. This gives rise to three new, nontrivial problems: 1) there must be a file system to manage files of information entrusted to the system; 2) the system must provide reasonable guarantees for the survival of a user's information in case of system failure or even the user's own mistakes, and 3) the system must control access to information in order to prevent unauthorized reading or writing

Abstractions Common to Systems

The common properties discussed above have given rise to abstractions about third generation system organization, by "abstraction" we mean a general concept or principle describing a problem area, from which most implementations can be deduced. The abstractions can be regarded as forming at least a basis for a "theory" of operating systems principles.

The five areas in which the most viable abstractions have evolved are

1) programming;
2) storage allocation;
3) concurrent processes;
4) resource allocation; and
5) protection.

The next five sections of this paper study these abstractions and some of their consequences in detail. Other areas, about which

few viable abstractions have evolved, will be discussed in the last section of the paper.

PROGRAMMING

It is generally agreed that the capabilities of the hardware in computer systems should be extended to allow efficient implementation of certain desirable programming language features. For this reason, programming objectives have had a profound influence on the development of computer and programming systems. The four most important of these are discussed in the following paragraphs.

1) *High-level languages:* The introduction of FORTRAN in 1956 and ALGOL in 1958 marked the beginning of a period of intense activity in high-level language development. From the first it was recognized that a programmer's problem-solving ability could be multiplied by high-level languages, not only because such languages enable him to express relatively complicated structures in his problem solution as single statements (contrasted with the many instructions required in machine language), but because they free him from much concern with machine details. The enormous variety of languages testifies to the success of this effort: algebraic languages, block-structured languages, procedure-oriented languages, string-manipulation languages, business-oriented languages, simulation languages, extensible languages, and so on [S2]. During the late 1950s there raged often bitter debates over the efficiency of compilation and execution of high-level language programs as compared to machine-language programs [W5]. These arguments subsided during the 1960s. Now it is generally agreed that compiler-generated code (and the compiling processes themselves) can be made of acceptable quality, and that any efficiency lost by the use of such code is handsomely compensated for by increased programmer productivity.
2) *Program modularity:* This term refers to the ability to construct programs from independently preparable, compilable, testable, and documentable "modules" (subprograms) that are not linked together into a complete program until execution. The author of one module should be able to proceed with its construction without requiring knowledge of the internal structure, operation, or resource requirements of any other module.
3) *Machine independence:* This term refers to the ability of a programmer to write programs without having to worry about the specific resource limitations of a system. It has been used according to one or more of three interpretations: (a) *processor independence*—appropriate compiling techniques enable a single program text to be prepared for execution on a processor with an arbitrary instruction set; (b) *memory independence*—a programmer may write a subprogram without having to know how much main memory might be available at execution time or what the memory requirements of (perhaps yet unwritten) companion subprograms might be; and (c) *I/O independence*—programmers obtain input (create output) by reading (writing) "I/O-streams," a given stream being attachable through interface routines to any device at execution time. A significant degree of machine independence of these three types is already being afforded by high-level languages, yet the techniques are only just beginning to be far enough advanced to permit program "transportability"—i.e., the ability to move a program or programming system written in any language from one machine to another [P2, W1].
4) *Structured data capabilities:* Programming languages capable of handling problems involving structured data (e.g., linked lists, trees) have been increasingly important, especially since the mid 1960s.

These four programming objectives are achieved to varying degrees in most modern computer and programming systems. Achieving them places new responsibilities on the operating system and on the compilers. They enable the creation of programs whose memory requirements cannot be predicted prior to loading time, necessitating

some form of dynamic storage allocation. To achieve a wide variety of languages, there must be a corresponding variety of compilers and a file system for storing and retrieving them. To achieve program modularity, the system and its compilers must provide a "linking" mechanism for connecting modules and establishing communications among them [M3]. This mechanism is normally provided by a "linker" or "linking loader" and can become quite complicated, especially when sharing is permitted [D1, S1]. To achieve memory independence, the system must provide a mechanism for translating program-generated memory references to the correct machine addresses, this correspondence being indeterminable a priori. This mechanism is normally provided in the form of a "relocating loader," "relocation hardware," or "virtual memory." To achieve structured data capability, the system must be able to allocate new storage on demand (should data structures expand) and deallocate storage as it falls out of use (should data structures contract).

Systems Programming Languages

There is an interesting parallel between events prior to 1965 and those following. Just as the question of machine-languages versus high-level language programming efficiency for "user programs" was debated before, so the same question for "systems programs" was debated after. Fortunately, the earlier debate did little to interfere with the development of high-level languages—most users were willing and anxious to use them, and computer centers were willing and anxious to satisfy the demand of the market. Unfortunately, the latter debate has hurt, seriously in some cases, the ability of third generation programming projects to meet their deadlines and to stay within their budgets.

The case for high-level languages for operating systems programming rests on four points [C3, C6]: 1) gross strategy changes can be effected with relative ease; 2) programmers may concentrate their energies and productivity on the problems being solved rather than on machine details; 3) programmers are more likely to produce readable text, an invaluable aid to initial implementation, to testing and documentation, and to maintenance; and 4) a principal source of system bugs, incompatibilities in the interfaces between modules authored by different programmers, can be removed by the more comprehensible intermodular communications provided by high-level languages. However valid might be the argument that machine code is more efficient than compiled code, especially in heavily-used parts of the operating system, it appears to miss the point: the modest improvements in efficiency that are gained using machine code simply cannot offset the cost of failing to achieve the abovementioned four advantages of high-level languages, especially during the early stages of complex projects when the design is not fixed.

The writing of operating systems programs is, however, more difficult intellectually than "ordinary applications" programming. The essential difference is the writer of an ordinary program is concerned with directing the activity of a single process, whereas the writer of an operating system program must be concerned with the activities of many, independently-timed processes. This sets two extraordinary requirements for operating systems programming languages. First, the language must provide the ability to coordinate and control concurrent activities (though this ability is not present in most programming languages, it is present in a number of simulation languages [M1, W3]), and, given this ability, the programmer must master the faculty of thinking in terms of parallel, rather than serial, activity. Secondly, in order to permit and encourage efficient representations of tables, queues, lists, and other system data areas, the language must provide a data structure definition facility. Not only must the programmer be skilled in defining and using data structures, he must be able to evaluate which one is most efficient for a particular need. Although these two requirements must be met whether or not a high-level language is used, a suitable language can smooth the way to their realization.

Several systems—such as the Burroughs B6500 [C6], IDA's time-sharing system [I3], and MULTICS [C3]—have made successful use of high-level languages throughout their operating systems.

Procedures

Because it is intimately related to programming modularity, the procedure (subroutine) has always been an important programming concept. Recursive and shared procedures have assumed positions of particular importance since 1960. An abstract description of "procedure in execution" is useful for understanding how an operating system implements an environment for efficient execution of procedures.

A procedure is said to be *activated* (i.e., in execution) between the time it is called and the time it returns. An activated procedure consists of a set of instructions and an "activation record" (i.e., working storage). Each activation of a procedure must have a distinct activation record because, without it, computations on local or private data would be meaningless; such computations arise under recursion and sharing. The activation record defines a local environment for the procedure, containing all operands that are to be immediately accessible to instruction references. Any objects that must be accessed but are not in the activation record (e.g., certain parameters of the procedure call) must be accessed indirectly through pointers in the activation record; all such objects constitute the nonlocal environment of the procedure activation. An activation record must also contain information by which control can be returned to the caller. Since activation records need exist only when the procedure is activated,* a scheme of dynamic allocation of storage for activation records is required.

An implementation of procedures based on the foregoing ideas must solve three problems: 1) the allocation and freeing of storage for activation records at procedure call and return; 2) the efficient interpretation of local operand references; and 3) the interpretation of nonlocal operand references. The solutions to these problems will depend on the objectives of the system and on the types of languages used; therefore, we shall not attempt to specify them in detail here. Good treatments are given in [J1, W3].

For the purposes of this discussion, assume that the memory is partitioned into two contiguous areas: a (read-only) area to hold the instruction parts of all procedures, and an area to hold activation records. Each activation record is assumed to occupy a contiguous region of memory. Each procedure activation is described at any given point by the value of an *activation descriptor* (i, r), where i is the address of the next instruction to be executed and r is the base address of the activation record. As suggested in Figure 1, the processor contains a pair (I, R) of registers that hold the activation descriptor of the currently executing procedure. (The register I is the instruction counter, and R is an activation record base register.) As in Figure 1, an instruction (OP, x), specifying some operation OP that involves the contents of address x, references the xth location relative to the base of the current activation record (i.e., location $x + c(R)$, where $c(R)$ denotes the contents of R).

As mentioned above, each activation record is provided with the activation descriptor of its caller. Figure 1 shows the first cell of an activation record used for this purpose. The steps in calling a procedure are compiled into the instruction code and include: 1) obtain a free contiguous region in the data area of memory of the requisite length for the new activation record, starting at some address r'; 2) initialize the values of all cells in the activation record, and provide values of, or pointers to, the parameters of the call; 3) let r be the contents of R and i the address of the instruction to be executed after the return; store (i, r) in the first cell of the new activation record—i.e. at location r'; and 4) let i' be the initial instruction of the called procedure, load (I, R) with (i', r') and begin executing instructions. The steps in returning include: 1) release the storage occupied by the current activation record; and 2) load (I, R)

* Parts of the activation record—e.g., own variables in ALGOL or STATIC variables in PL/I—may have to continue their existence even after the procedure is deactivated.

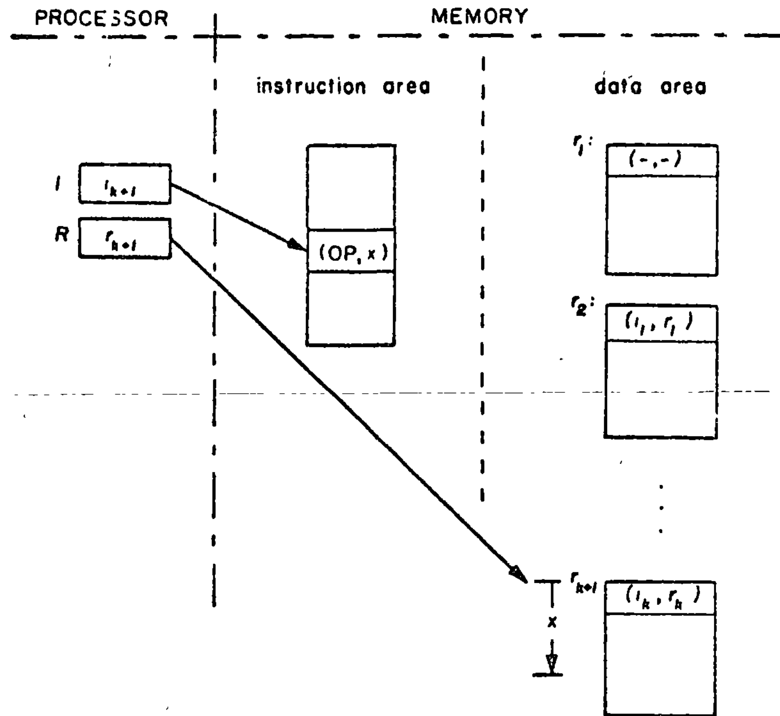


FIG. 1. Procedure implementation

with the activation descriptor in location $r' = c(R)$.

The chain $(i_1, r_1), \cdots, (i_k, r_k)$ of activation descriptors (Figure 1) linking the activation records to their callers is known as the "dynamic chain." Since returns are executed in the reverse order of calls, many implementations handle the allocation of storage for activation records by means of a pushdown stack [W3]. It should be clear from the descriptions above that the stack mechanism is a consequence, not a prerequisite, of procedure implementation.

The most common method for handling nonlocal operand references (those to be interpreted outside the context of the current activation record) is based on the properties of block-structured languages, such as ALGOL, where blocks define a hierarchy of nested contexts. These implementations treat block entry and exit the same as procedure call and return (except for parameter passing), and provide each activation record with a pointer to the activation record of the lexicographically enclosing block in the program text. The chain of pointers so defined is known as the "static chain." Complete descriptions of the problems involved in constructing and using static chains can be found in [J1, W3].

STORAGE ALLOCATION

Of all the resources of a system, memory is perhaps the most scarce, and so techniques and policies for managing it have consistently received much attention. Computers have always included several distinct types of storage media, and the memory has been organized into at least two levels: main (directly addressable) memory, and auxiliary (backup) memory. The desire for large amounts of storage has always forced a compromise between the quantities of (fast, expensive) main memory and (slow, cheap) auxiliary memory.

To understand modern memory systems, one must understand two sets of techniques, *memory management* and *name management*. The techniques of memory management concern the accessing and placement of in-

formation among two or more levels of memory. The most important among these techniques is the "one-level store," or virtual memory. The techniques of name management are concerned with achieving very general forms of programming modularity by providing for the definition of different contexts within which processes can access objects. The most important among these techniques are file systems and "segmentation."

In the following discussion, the terms *computational store* and *long-term store* will be used. The computational store is that part of the memory system in which objects must reside so that a process can reference them using the hardware addressing facilities of the system. The long-term store is that part of the memory system in which objects not immediately required in any computation may reside indefinitely. Both terms refer to memory as presented to the programmer. Both may use main and auxiliary memory in their implementations. Most systems have distinct computational and long-term stores; a few do not.

Motivations for Automatic Memory Management

Most of the modern solutions to the automatic storage allocation problem derive from the one-level store, introduced on the Atlas computer [K1]. This machine made the distinction between "address" and "location," with an address being the name for a word of information and a location being a physical site in which a word of information is stored. The set of all addresses a processor can generate as it references information (program addresses) has come to be known as *address* or *name space*, and the set of all physical main memory locations (hardware addresses) has come to be known as *memory space*. By making this distinction, one is able to remove considerations of main memory management from programming, for the name space can be associated permanently with the program and made independent of prior assumptions about memory space. The memory management problem becomes the system's problem as it translates program addresses into hardware addresses during execution. Examples of contemporary systems are discussed in [R1].

The one-level store implements what appears to the programmer as a very large main memory without a backing store; hence, the computational store is a large simulated (virtual) main memory. Two lines of argument are used to justify this approach: the first reasons that the "overlay problem" can be solved by simulating a large, programmable memory on a machine with relatively smaller main memory; the second traces the possible times at which program identifiers are "bound" to (associated with) physical locations and reasons that the one-level store provides maximum flexibility in allocating memory resources by delaying binding time as long as possible.

The argument for an automated solution to the overlay problem proceeds as follows. Since main memory in the early stored-program computers was quite small by today's standards, a large fraction of program development was devoted to the *overlay problem*—viz., deciding how and when to move information between main and auxiliary memory and inserting into the program text the appropriate commands to do so.* The introduction of algorithmic source languages (e.g., FORTRAN and ALGOL) and the linking loader made it possible to construct large programs with relative ease. As the complexity of programs increased, so grew the magnitude of the overlay problem; indeed, 25 to 40% of programming

* Typically, the programmer would divide his information into "objects" and his computation time into "phases." During any phase only a subset of the objects would be referenced. The problem was to choose the objects and phases such that: 1) the total space occupied by referenced objects during any phase did not exceed the size of main memory; 2) the phases were as long as possible, 3) any object remaining in memory in successive phases did not have to be relocated, and 4) the amount of information that had to be moved out of memory during one phase, to be replaced (overlaid) by information moving in for the next phase, was minimal. In addition to these requirements, it was desirable to overlap the overlaying process as much as possible with execution. For programs involving sizable numbers of phases and objects, finding any reasonable solution (much less an optimal one) to the overlay problem was a formidable task. The problem was always more severe with data objects than with procedure objects.

costs could typically be ascribed to solving the overlay problem [S3]. The need for executing large programs in small memory spaces thus motivated the development of hardware and software mechanisms for moving information automatically between main and auxiliary memory.

The argument for postponing binding time proceeds as follows. There are five binding times of interest: 1) if the program is specified in machine language, addresses in it are bound to storage locations from the time the program is written; 2) if the program is specified in a high-level language, program addresses are bound to storage locations by the compiler; 3) if the program is specified as a collection of subroutines, program addresses are bound to storage locations by the loader. In these three cases, binding is permanent, once fixed. It may, however, also be dynamic: 4) storage is allocated (deallocated) on demand, such as at procedure activation (deactivation) times in ALGOL or at data structure creation (deletion) times in LISP and PL/I; and 5) storage is allocated automatically by memory management hardware and software provided by the computer system itself. The sequence 1 through 5 is, in fact, the historical evolution of solutions to the storage allocation problem. It is characterized by postponement of binding time. Although postponing binding time increases the cost of the implementation, it does increase freedom and flexibility in constructing programs and in allocating resources.

The two preceding paragraphs outline the qualitative justification for automatic storage allocation. Further discussions of these points from different perspectives can be found in [A2, D3, D6, D7, R1]. Quantitative justification is provided by Sayre [S3]. However one arrives at the conclusion that some form of dynamic storage allocation is required, it is apparent that the programmer cannot handle it adequately himself. Not only is he not privy to enough information about machine operation to make allocation decisions efficiently, but solving the overlay problem at the programming level requires extensive outlays of his valuable time, and the results are not consistently rewarding.

Virtual Memory

Modern memory systems are studied by means of the following abstractions: address space, memory space, and address map. These abstractions allow one to discern a pattern among the seemingly endless variety of existing memory systems. They are summarized in the form of a mapping

$$f: N \rightarrow M$$

where N is the address space of a given program, M is the main memory space of the system, and f is the address map. If the word with program address x is stored at location y, then $f(x) = y$. If x is in auxiliary memory but not in main memory, $f(x)$ is undefined, and an attempt to reference such an x creates a fault condition that causes the system to interrupt the program's execution until x can be placed in memory and f updated. The physical interpretation of f is that a "mapping mechanism" is interposed between the processor and memory (Figure 2) to translate processor-generated addresses (from N) into locations (in M). This reflects our earlier requirement that the address space N be independent of prior assumptions about M; as the programmer is aware only of N and not of the two levels of memory, the processor that executes his program can generate addresses from N only.

Let us consider some salient aspects of virtual memory implementations. Diagrams more detailed than Figure 2 indicating the operation of address mapping in specific cases have been given in [D1, D6, D7, R1, W3, W6] and are not reproduced here. The address map f is normally presented in the form of a table so that it can be accessed and updated efficiently. However, when considering viable implementations of f it is necessary to dispense with the notion of providing separate mapping information for each element of N. Instead, N is partitioned into "blocks" of contiguous addresses, and separate mapping information is provided for blocks only,* thus reducing the potential

* If b is a block number, then either $f(b)$ is the base address of a contiguous region of M containing block b, or $f(b)$ is undefined if block b is missing from M. The mapping mechanism makes a sequence of transformations, $x \rightarrow (b, w) \rightarrow (f(b),$

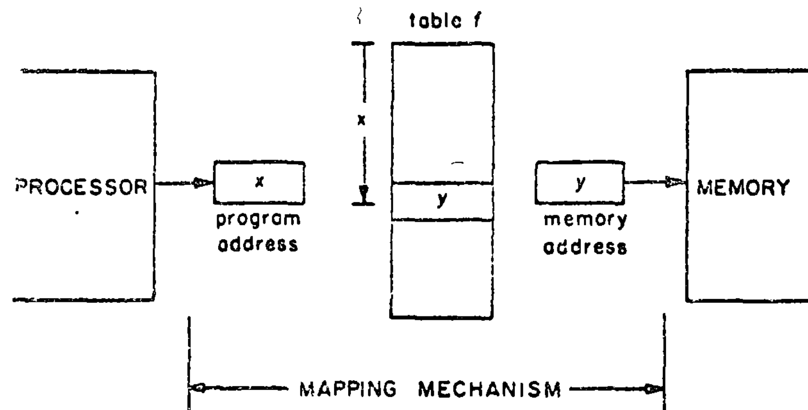


Fig. 2 Address translation.

size of the mapping table to manageable proportions.

In most implementations, the mapping table f is stored in main memory as a directly-indexed table (suggested in Figure 2), and a small associative memory is included in the mapping hardware to speed up the translation process.** With respect to the blocks themselves, there are two alternatives: the block size is either fixed and uniform or it is variable. When the block size is fixed and uniform, the blocks of address space are called "pages" and the blocks of main memory, "page frames." The second alternative arises when one considers defining the blocks of address space according to natural logical boundaries, such as subroutines. When deciding which alternative to use, the designer must take into account the efficiency of the mapping mechanism, the efficiency of storage utilization, and the efficiency of possible allocation policies [D6]. The designer must also be aware of two conflicting tendencies: on the one hand, fixed block size leads to simpler, more efficient storage management systems when properly designed, whereas the need for efficient name management requires that N be partitioned according to logical boundaries. A compromise, known as "segmentation and paging," will be discussed below.

Observe that the foregoing definition of an address map is independent of the physical realizations of main and auxiliary memory. One common type of memory system uses magnetic core main memory and drum auxiliary memory, the translation of addresses and management of memory being implemented by a combination of hardware and software [D6, R1]. Another common type of memory system uses semiconductor register main memory and magnetic core auxiliary memory, the translation of addresses and management of memory being implemented entirely in hardware [L2, W4]. The former type of memory system is the classical "core-drum" system derived from the Atlas computer; the latter is the "cache store" or "slave memory" used in many late third-generation machines (e.g., the IBM 360/85, certain members of the IBM 370 series, and the CDC 7600).

All types of virtual memory systems have the five important properties detailed below

$w) \rightarrow y = f(b) + w$, where x is a program address, b is the number of the block containing x, w is the address of x relative to the base of b, and y is a memory address. The transformation $x \rightarrow (b, w)$ may be omitted if, as is sometimes the case, elements of N are already represented as (b, w) pairs [D6]

** The associative memory consists of cells containing entries of the form $(b, f(b))$. Whenever a reference to block b is made, the associative memory cells are searched in parallel for an entry $(b, f(b))$. If the required entry is found, the base address $f(b)$ of the block is available immediately; otherwise, the table f in memory must be accessed, and the entry $(b, f(b))$ replaces the least recently used entry in the associative memory. In the former case, the time required to complete the reference to the desired address-space word is one memory cycle time while in the latter case it is two memory cycle times. Associative memories of only 16 cells have been found to bring the speed of the mapping mechanism to within 3.5% of the speed at which it would operate if the address map could be accessed in zero time [S4].

1) Virtual memory fulfills a variety of programming objectives. (a) memory management and overlays are of no concern to the programmer, (b) no prior assumptions about the memory space M need be made, and the address space N is invariant to assumptions about M; and (c) physically, M is a linear array of locations and N is linear, but contiguous program addresses need not be stored in contiguous locations because the address map provides proper address translation; thus, the address map gives "artificial contiguity" and, hence, great flexibility when decisions must be made about where information may be placed in main memory.
2) Virtual memory fulfills a variety of system design objectives in multiprogramming and time sharing systems; the abilities to: (a) run a program partly loaded in main memory; (b) reload parts of a program in parts of memory different from those they may previously have occupied, and (c) vary the amount of storage used by a program.
3) Virtual memory provides a solution to the relocation problem. Since there is no prior relationship between N and M, it is possible to load parts of N into M without regard to their order or previous locations in M. However, it may still be necessary to use a loader to link and relocate subprograms within N.
4) Virtual memory provides memory protection: a process may reference only the information in its address space (i.e., in the range of the address map), any other information being inaccessible to it. This is an immediate consequence of the implementation of address translation, according to which each address must be mapped at the time it is generated. It is, therefore, impossible to reference any information not represented in the map. Assuming the correctness of the map, it is likewise impossible to reference unauthorized information. In addition, it is possible to include "protection keys" in each entry of the map, so that only references of the types specified by the keys (e.g., read, write, execute) will be permitted by the mapping hardware.

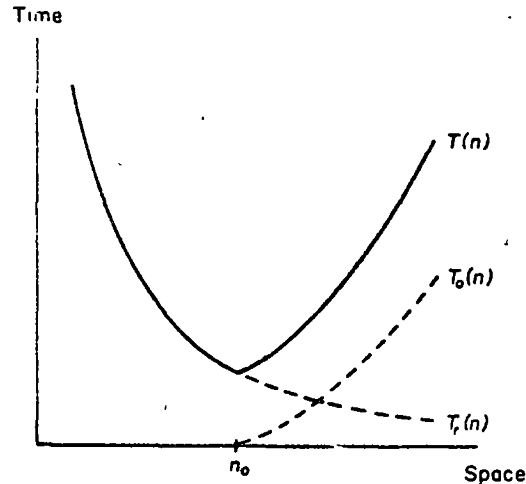


FIG 3. Space and time in virtual memory.

5) The space/time tradeoff does not hold in a virtual memory. Let $T_r(n)$ denote the running time of the fastest program that correctly solves a given problem when the address space size is n and the entire program is loaded in main memory. The space/time tradeoff states that the best program is faster in larger memories; i.e., $T_r(n + 1) \leq T_r(n)$. When the main memory size is m, the observed running time is $T(n) = T_r(n) + T_o(n)$, where T_o is the overhead in the storage management mechanism: $T_o(n) = 0$ for $n \leq m$ and $T_o(n + 1) \geq T_o(n)$ for $n > m$ (see Figure 3). Therefore, when $n > m$, it may easily happen that $T(n + 1) > T(n)$, i.e., space and time need not trade with respect to observed running time. Since prior knowledge of m is denied the programmer, he cannot rely on the tradeoff. To achieve good performance from a virtual memory system, the programmer must instead use techniques that maximize the "locality of reference" in his program [D6, pp. 183–7]

To this point, the discussion has dealt with the mechanisms of virtual memory. Once the mechanism has been chosen, a policy is needed to manage it. A memory management policy must strive to distribute information among the levels of memory so that some specified objective is achieved. If, for example, the objective is maximum system operating speed, the policy should seek to retain in main memory the informa-

tion with the greatest likelihood of being referenced. A memory management policy comprises three subpolicies: 1) the fetch policy, which determines when a block should be moved from auxiliary memory into main, either on demand or in advance thereof; 2) the placement policy, which determines where in the unallocated region of main memory an incoming block should be placed; and 3) the replacement policy, which selects blocks to be removed from main memory and returned to auxiliary. The complexity of the three subpolicies can be compared according to whether block size is fixed or not, and the complexity of a good policy may well affect the choice of whether to use fixed or variable block size. For example, in paging systems all blocks are identical as far as the placement policy is concerned, and in demand paging systems (which fetch on demand and replace only when main memory is full) the memory management policy reduces to a replacement policy [B1, M2].

Demand paging is widely used and well documented in the literature. Many policies have been proposed and studied [B1, D6, M2], the basis for most is the "principle of optimality" for minimizing the rate of page replacements: replace the page having the longest expected time until reuse. Although this principle is not optimal for arbitrary assumptions about program behavior, it is known to be a very good heuristic.

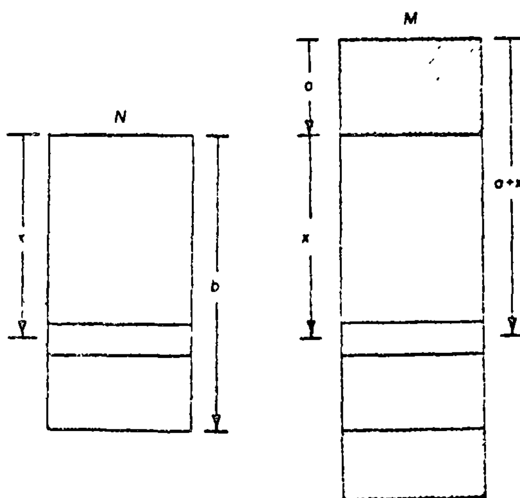


FIG. 4. Relocating an address space in memory.

The study of main memory management can be rounded out by the study of a dual problem, auxiliary memory management. Many of the considerations identical to those discussed above are relevant here: whether or not block size should be fixed, what the block size(s) should be, and how to store tables locating blocks in the auxiliary memory. However, the policies of auxiliary memory management are not the same as those of main memory management, because the objective is reducing channel congestion and waiting times in auxiliary memory queues. The "shortest-access-time-first" policies for managing request queues are known to be optimal or nearly so [A1, D6].

The auxiliary memory of many a large paging system is comprised of both drums and disks, the drums being used to swap pages of active programs to and from main memory, the disks for long-term storage of files. Files used by active programs may exist on both drum and disk simultaneously. The term "page migration" refers to the movement of pages of such files between drum and disk (usually through main memory buffers); in some systems it can be a serious problem with no obvious, simple solution [W6, p. 45].

Extending the foregoing considerations to multiprogramming is straightforward. In most cases, a separate address space and address map are associated with each process. There are two ways to map the resulting collection of address spaces into main memory. The first uses "base and bound" registers to delineate a contiguous region of memory assigned to a given address space. As shown in Figure 4, the address space N, consisting of b words, is loaded contiguously in main memory starting at base address a. The address map $f:N \rightarrow M$ is a simple translation defined by $f(x) = x + a$; it can be implemented by the mapping mechanism shown in Figure 5. The base register (known also as a relocation register) contains the address a, which is added to the address x generated by the processor. The address x can be checked against the bound b (address space size) and an error signal generated if $x > b$. When the processor is switched to a new process, the base and bound registers

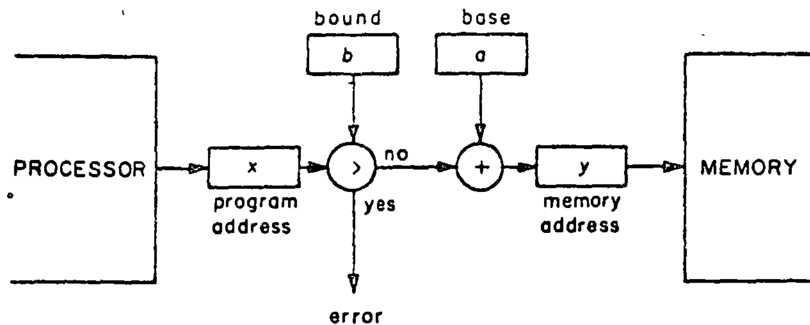


FIG. 5 Use of base and bound registers.

must be reloaded to contain the mapping information for the new process. This method is useful only when main memory can hold several address spaces, as for example in the CDC 6600 or the IBM OS/360-MVT.

The second method of mapping a collection of address spaces into main memory is a direct extension of the block-oriented mappings discussed earlier: main memory is treated as a pool of blocks, and the system draws from this pool to assign blocks to individual address maps as needed. This second alternative is widely used, for it permits mapping of address spaces larger than main memory. When properly designed, it can be especially efficient under paging.

Memory management policies for multiprogramming are best understood with the aid of a *working-set* model for program behavior. A program's working set at any given time is the smallest subset of its address space that must be loaded in main memory in order to guarantee a specified level of processing efficiency. (For proper choices of the parameter T in a paging system, one can use the set $W(t, T)$ of pages referenced among the T page-references immediately preceding time t as an estimator of the working set at time t [D3, D6].) The working-set concept has implications for both the programmer and system designer. A programmer can realize good performance from a virtual memory system by creating programs whose working sets are small, stable, and slowly changing. (If this is achieved, the total amount of address space employed is immaterial.) A system designer can realize good performance under multiprogramming by designing policies which guarantee that each active process has its working set loaded in main memory. If this principle is not followed, attempted overcommitment of main memory may induce collapse of performance, known as thrashing [D4]. The performance of core/drum paging systems can be improved by departing from strict adherence to demand paging: during a time slice, pages can be added on demand to the working set of a process and replaced when they leave it; but at the beginning and end of a time slice, swapping can be used to move the working set as a unit between main and auxiliary memory.

Motivations for Name Management

The techniques of virtual memory outlined above define a computational store which appears to the programmer as a very large, linear name space. But, once presented with a single linear name space, no matter how large, the programmer is still faced with the need to solve four important problems:

1) handling growing or shrinking objects, and creating or deleting objects, without having to resort to using overlays in name space or to introducing dynamic storage allocation procedures into name space (in addition to the ones implementing the computational store);
2) providing long-term storage and retrieval for named information objects;
3) allowing information to be shared and protected; and
4) achieving programming modularity, according to which one module can be compiled (into some part of name space) with-

out knowledge of the internal structure of other modules and without having to be recompiled when other modules are.

A complete discussion of why linear name space fails to satisfy these objectives has been given by Dennis [D7]. The basic problem is that the dynamics of information additions and deletions in name space implied by each of these objectives require some method of structuring and managing names, a method not intrinsically provided by the virtual memory mechanisms as described above.

Modern computer systems use two methods for achieving these objectives: 1) *file systems*, by far the most widely used, which provide a storage system external to the name space; and 2) *segmented name space*, used in MULTICS [B2], which alters the structure of the name space itself. The file system implements a long-term store separate from the computational store, while the segmented name space implements a long-term store identical to the computational store. We shall comment on each below.

File Systems

Many of today's operating systems provide both a virtual memory and a file system, an apparent paradox since the virtual memory is supposed to conceal the existence of auxiliary memory, while the file system does not. The paradox is resolved in the light of the inability of linear name space to satisfy the four objectives given above. The programmer is presented with a pair of systems (N, F), where N is a linear name space and F is the file system. An address map $f: N \rightarrow M$ is implemented in the usual way. The file system consists of a *directory* or *directory tree* [D2, D11] and a set of *file processing primitives*. The directory represents a mapping from file names to auxiliary memory. Arranging a collection of directories in a tree has two advantages: 1) each directory can be identified uniquely by its *pathname*, i.e., the sequence of directory-names on the path from the root of the tree to the given directory; and 2) the same name can be reused in different directories. Observe that the directory tree can be regarded as a "file name space" in which objects (files) are named by pathnames, this is contrasted with the linear name space discussed earlier. In addition, a directory tree serves to provide various contexts within which processes may operate; it does this by allowing each process to have a pointer designating a "current directory" in the tree. The file processing primitives include operations like "search directory"; "change context to predecessor or successor directory"; "link to an entry in another directory"; "open file"; "close file"; "create file"; "delete file"; "read file into N"; and "write file from N." The directory trees of individual users can be made subtrees of a single, system-wide directory tree, and users can employ the linking primitive to share files. Protection keys can be stored in directory entries. Since files can grow and shrink, be created and destroyed, be stored for long periods, be shared and protected, and be program modules, the four programming objectives given above are satisfied.

Yet, file systems have an important limitation, which is a consequence of the distinction between computational and long-term storage systems: in order to take advantage of the efficiencies of the addressing hardware, the programmer is forced to copy information between the file system and the computational store; this, in effect, reintroduces a problem that virtual memory sought to eliminate—the overlay problem in name space. As will be discussed shortly, the segmented name space overcomes this problem by removing the distinction between the computational and long-term storage systems.

Frequently mentioned in discussions about file systems is "file structure," i.e., the organization of information within a file [I1]. Three types of file structures are in common use:

1) bit or word strings (a linear address subspace);
2) chains of records; and
3) a collection of records, each having a unique index (key), with the indexes having a natural order; a given record can be located using hashing techniques when its index is presented, or the records can be searched sequentially in the order of the indexes.

The first structure represents a fundamental way of thinking about files; the second and third do not. The physical properties of auxiliary memories (disks, drums, tapes) have always forced the use of records in the implementation of a file; but, just as the blocks of memory used by the mapping mechanism need not be visible to the programmer, so the records of a tape or the tracks of a disk or drum need not be visible. In other words, records and sequential indexing are not prerequisites to understanding a file system. By confusing the implementation of mappings with the logical properties of files, many systems present a confusing and overly complicated picture to the programmer in this respect.

Segmentation

The power of the segmented name space to solve the four programming problems presented under "Motivations for Name Management," above, without the limitations to which file systems are subject, appears not to be widely appreciated. One can regard a file system as providing a collection of named, linear address subspaces (files) of various sizes, augmenting the main name space implemented by the computational store. From this, we can make the generalization that, instead, the computational store itself comprises a collection of named, linear subspaces of various sizes. Each of these subspaces is called a *segment;* the entire name space is called a *segmented name space;* and the capability for a segmented name space is called *segmentation*. Each word is referenced by a two-dimensional address (segment name, word name); this is contrasted with linear name space in which a single dimension of addressing is used. Since all segments may reside permanently in the name space (no separate file system is required) and the programmer does not have to program file operations or solve an overlay problem, the four programming problems are solved.

Under segmentation, however, the system must handle the linking of segments into a computation, and special hardware and compiling techniques are required to make this process efficient [D1]. (In systems without segmentation, the programmer implicitly solves the linking problem while copying files from the file system into specified areas of address space.) The most commonly proposed approach to sharing segments among the name spaces of distinct processes uses a system-wide directory tree, as discussed above for file systems [B2, D2], so that each segment is represented twice in the system.

There are two methods for mapping a segmented name space into main memory. The first stores a segment contiguously in main memory, in the manner of Figures 4 and 5, except that the base-and-bound information is stored in the mapping table. This technique is subject to the same limitations as the variable-block-size mappings discussed earlier [D6]. The second method, known as *segmentation and paging* [D7], uses paging to map each segment, which is a linear name space in its own right. It uses two levels of tables to map a given segment-word pair (s, x) to a main memory location y in the following steps: 1) the segment name s is used to index a "segment table" whose sth entry gives the base address of a "page table" for segment s; 2) the word name x is converted to a pair (p, w), where p is a page number and w the relative address of x in page p; 3) the page number p is used to index the page table whose pth entry contains the base address q of the page frame holding page p; and 4) the memory address is $y = q + w$. Experiments on the MULTICS system demonstrate that the speed degradation caused by using this mapping technique, together with a 16-cell associative memory, averages 3.5% [S4]; thus, segmentation and paging competes in efficiency with more standard forms of virtual memory.

CONCURRENT PROCESSES

We mentioned earlier that concurrency is an important aspect of computer systems, that mechanisms for controlling it must be available for use in some programming languages, and that programmers must understand the basic concurrency problems and their solutions. Even though the present hardware technology allows for the realization of con-

current activity, it remains the task of the programmer to write the programs that bring it into being.

It appears more natural to regard a computer system as a set of processes cooperating and rendering service to one another, rather than as a set of subroutines performing in the same fashion. In other words, the unit of system decomposition is the process, not the subroutine [B5]. This view pervades the design of Dijkstra's THE multiprogramming system [D14] and has influenced other third generation systems, though not as strongly. Under this view, the most elementary conceivable operating system is one that provides for the creation, removal, control, and intercommunication of processes; all other operating system functions can be organized around this [B5].

The concept "process" appears to have originated in several design projects during the early 1960s; for example, in MULTICS and in OS/360 (where it is called a "task"). It was intended as an abstraction of the activity of a processing unit, and was needed to solve the problem of preempting and restarting programs without affecting the results of their computations. Although many early definitions of "process" were imprecise, they were adequate to enable system designers to implement multitasking. Some examples of early definitions are: "locus of control in an instruction sequence" [D11]; "program in execution on a pseudo-processor" [S1]; "clerk carrying out the steps of an algorithm" [V1]; and "sequence of states of a program" [H5]. The intent of these definitions is to distinguish the static program from the dynamic process it generates, and to suggest that the term "program" be applied to a sequence of instructions, while the term "process" be used to denote a sequence of actions performed by a program.

A precise definition of process can be developed as follows. With a given program we associate a sequence of "actions" $a_1a_2 \cdots a_k \cdots$ and a sequence of "states" $s_0s_1 \cdots s_k \cdots$. For $k \geqslant 1$, action a_k is a function of s_{k-1}, and s_k is the result of action a_k. The states represent conditions of a program's progress, with s_0 the initial state. The sequence $a_1a_2 \cdots a_k \cdots$ is called an *action sequence*, and $s_0s_1 \cdots s_k \cdots$ a *computation*. A *process* is defined as a pair $P = (s, p)$, where p is a program and s is a state. The process (s, p) implies a future action sequence $a_1a_2 \cdots a_k \cdots$ and a computation $s_0s_1 \cdots s_k \cdots$, where $s_0 = s$. Interrupting a process (s, p) after k steps is equivalent to defining the new process (s_k, p), and (s_k, p) is called a "continuation" of (s, p). We have defined "process" in this way in order to capture the flavor of its usage in practice, where a process (s, p) can be implemented as a data structure and be regarded as an entity that demands resources. Neither the action nor the state sequences are by themselves adequate for this.

The interpretation of "action" and "state" depends on the process control problem under consideration. For example, if we were interested in preempting processor or memory resources under multiprogramming, a "state" would consist of a specification (or implied specification) of the contents of all processor registers and name space locations, and an "action" would be an instruction execution. Thus, an adequate description of the process for program p is the vector (i, R, f, p), in which i is the instruction counter, R is the contents of all processor registers, and f is (a pointer to) the address map.* Preempting this process requires saving the information (i, R, f). Other interpretations of "action" and "state" are discussed below.

In order to apply the foregoing concepts to a system, we need to generalize them. To do this we define a *system of processes* to be a collection of processes, together with a specification of precedence constraints among them. If process P_1 precedes process P_2, then P_1 must terminate execution before P_2 can begin. The precedence constraints are assumed to be physically realizable in the sense that no process can precede itself. (In mathematical terms, the precedence constraints partially order the system of processes.) Two processes without any prece-

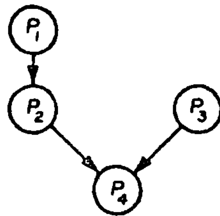
* Note the similarity between this definition of "process" and the activation descriptor definition given above for a procedure in execution. In fact, the definition of "process" is a direct generalization of that of "procedure in execution."

dence constraint between them are *independent*, and may be executed in parallel. An action sequence generated by the system consists of concatenations and merges of the action sequences of its component processes, consistent with the precedence constraints; that is, a system action sequence is obtained by merging the process action sequences such that, if P_1 precedes P_2, then the actions of P_1 must precede those of P_2 in the system action sequence. Even though the process action sequences are unique and the precedence constraints fixed, there may be many possible system action sequences, depending on the total number of possible ways of merging the sequences of independent processes. Stated differently, the action sequence observed in a given run of a system of processes will depend on the relative speeds of the individual processes, which, in general, may not be predictable. Figure 6 illustrates a system of four processes and three possible system action sequences. Finally, of course, each possible system action sequence will correspond to a sequence of system states, i.e., to a system computation.

Let $P_i = (s_i, p_i)$ be the ith process in a given system. If s_i is not the final state of P_i (i.e., if further actions are possible by P_i), then any process P_j that P_i precedes is *blocked* or *disabled*. If P_i is not blocked and s_i is not its final state, then P_i is *active*.

With respect to a system of processes, there are four control problems of importance.

1) The *determinacy* problem arises whenever a system of processes shares memory cells. The system is determinate if, no matter which system action sequence arises, the result of the computation, as manifested in the contents of memory cells, depends uniquely on the initial contents of memory cells. In other words, the outcome of the computation is independent of the relative speeds of the processes in the system.
2) The *deadlock* problem arises in limited resource systems where it may be possible for a subset of the processes to be blocked permanently because each holds nonpreemptible resources requested by other processes in the subset.
3) The *mutual exclusion* problem arises whenever two or more independent processes may require the use of a reusable resource R. Since each process modifies the internal state of R while using it, no two processes may be using R at the same time. Since each process initializes the internal state of R when it first acquires it, the order in which processes gain access to



Precedence Diagram

Process	Action Sequence
P_1	$a_1 b_1 c_1$
P_2	$a_2 b_2$
P_3	$a_3 b_3 c_3$
P_4	$a_4 b_4$

Sample System Action Sequences:

$a_1\, b_1\, c_1\, a_2\, b_2 a_3\, b_3\, c_3 a_4\, b_4$

$a_1\, b_1\, c_1\, a_3\, b_3 c_3\, a_2 b_2\, a_4\, b_4$

$a_1\, b_1\, a_3\, c_1\, b_3 a_2\, c_3 b_2\, a_4\, b_4$

Fig. 6. A system of processes.

R is immaterial. Two processes are mutually excluded with respect to R if at most one may be using R at a time.

4) The *synchronization* problem between two processes arises whenever the progress of one depends on that of the other. Common examples of synchronization include: the requirement that one process cannot continue past a certain point until a signal is received from another; and, in the case of certain cooperating cyclic processes, the requirement that the number of completed cycles of one should not differ from the number of completed cycles of the other by more than a fixed amount.

These four control problems are described in greater detail below. Their solutions constrain the system of processes, either in the design or by the addition of external control mechanisms, so that only the desirable system action sequences and computations arise. The solutions given here should be regarded as models for the desired behavior of systems; they do not necessarily represent the most practical approach for a specific problem. The designer would have to be prepared to modify the solutions to suit the exigencies of the system at hand. Discussions of other approaches can be found in [A3, A5].

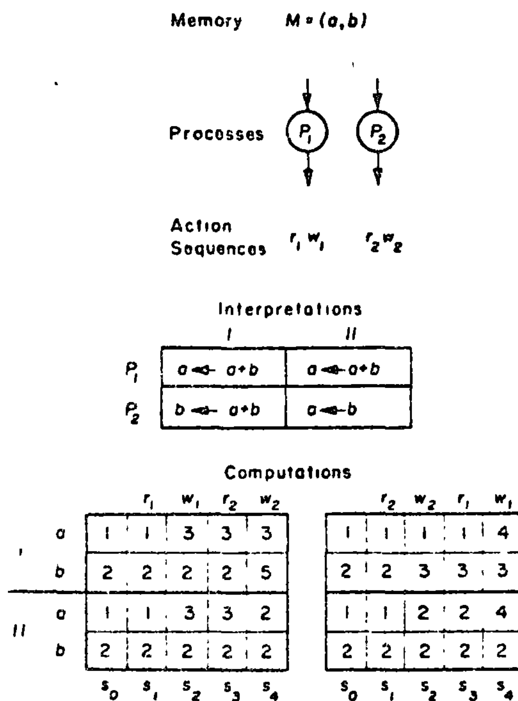


FIG. 7. Nondeterminate system of processes

Determinacy

Consider a system of processes having access to a set M of memory cells. Each process has associated with it fixed sets of *input* and *output* cells, both subsets of M. The actions performed by each process are read or write actions on the input and output cells, respectively, of that process. A state in this system is a vector of values of cells in M; a read action leaves the state unchanged, whereas a write action may modify the values in the output cells of the process. A system computation is the sequence of such states generated by a system action sequence.

An *interpretation* for a process is a specification of the algorithm performed by the program of that process, subject to the requirement that the input and output cells of the process be fixed. A system of processes is determinate if, for all possible interpretations of the processes in the system, the final contents of M depend uniquely on the initial contents of M, regardless of which system action sequence arises. (That is, the final state of a system computation is uniquely determined by the initial state.) Since a process generates a computation depending only on the initial contents of its input cells, individual processes are determinate.

Figure 7 shows a simple example of a nondeterminate system. The two processes P_1 and P_2 are independent and use two memory cells. For $i = 1, 2$, the action sequence of P_i is $r_i w_i$, where r_i denotes a read action on the input cells of P_i and w_i denotes a write action on the output cells of P_i. Two possible interpretations, I and II, are given. For each interpretation, two computations are shown in the table, the rows being the contents of the indicated memory cells after successive actions in the action sequence, and the columns being the system states. For both interpretations, the final system state depends on the order in which the two processes perform their read and write operations; hence, the system is non-

determinate under both interpretations. It is not hard to see that interpretation I is nondeterminate because each process writes into the input cells of the other, and that interpretation II is nondeterminate because both processes attempt to write a value into the same output cell.

Two processes are *noninterfering* if: 1) one precedes the other in the system of processes; or 2) they are independent and no output cell of one is an input or output cell of the other. It can be shown that noninterference is a sufficient condition for the determinacy of a system of processes [A6]. Under general conditions, noninterference is a necessary condition for determinacy.

Deadlocks

A deadlock is a logical problem arising in multitask systems in which processes can hold resources while requesting additional ones, and in which the total demand may exceed the system's capacity even though the demand of each individual process is within system capacity. If at some point in time there is a set of processes holding resources, none of whose pending requests can be satisfied from the available resources, a deadlock exists, and the processes in this set are deadlocked.

Figure 8 is an informal illustration of the essential features of the deadlock problem [C1]. The figure shows a two-dimensional "progress space" representing the joint progress (measured in number of instructions completed) of two processes. Any sequence of points, (x_1y_1), (x_2y_2), $\cdots$, (x_ky_k), $\cdots$, starting at the origin (0, 0) in this space, in which each point is obtained from its predecessor by a unit increment in one coordinate, is called a "joint progress path." Such a path may never decrease in either coordinate because progress is assumed to be irreversible. Now, the system has one unit each of resource types R_1 and R_2. Since each process requires exclusive control over R_1 or R_2 during certain stages of its progress, no joint progress path may pass through the region of progress space in which the joint demand of the processes exceeds the capacity in either R_1 or R_2. This is called the *infeasible region.* Deadlock is possible in this system

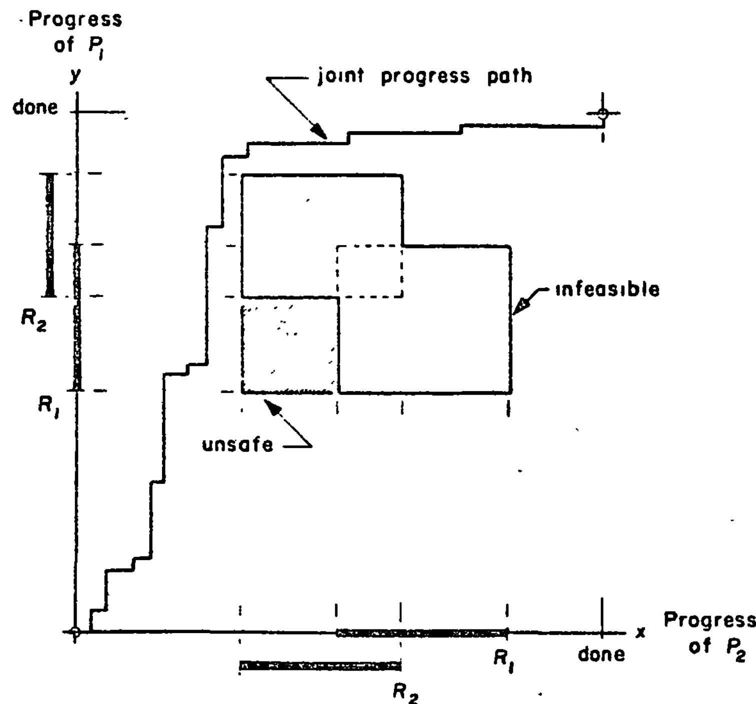


FIG. 8. Illustration of deadlock problem.

because, if joint progress ever enters the unsafe region, there is no way to avoid the infeasible region without violating the assumption of irreversibility of progress. A deadlock will exist when joint progress reaches the point at the upper right corner of the unsafe region. At that time, P_1 will be holding R_1 and requesting R_2, while P_2 will be holding R_2 and requesting R_1.

From this example one can see that there are three conditions necessary for deadlock to occur:

1) each process can claim *exclusive control* over the resources it holds;
2) each resource is *nonpreemptible*, i.e., it can be released only by the process holding it; and
3) there is a *circular wait*, each process holding a resource of one type and requesting the resource of the other type.

If a system can be designed such that any one of these conditions is precluded a priori, the system will be free from deadlock. It is not always possible to prevent deadlocks by denying the first two conditions because of various properties of resources. The third condition can, however, be precluded by a rather interesting method called "ordered resource usage": the resource types $R_1, \cdots, R_n$ are ordered and requests are constrained so that, whenever a process holds R_i and requests R_j, then $i < j$ [H2]. Restricting requests in this way is possible in assembly-line type systems (e.g., in input-execute-output programs) where a process progresses upward in the resource ordering, but never re-requests a resource type it holds or once held.

The more difficult aspects of the deadlock problem concern the detection of deadlock and the avoidance of unsafe regions in progress space. To study them, it is necessary to formalize the problem, modeling the way in which processes request and release resources. For the system-of-processes model, it is sufficient to study a system of independent processes, as deadlock can arise only among processes simultaneously competing for resources. The actions performed by a process are "requests" and "releases" of resources. A system state is a specification of: 1) how much of each resource type is currently allocated to each process; and 2) how much of each resource type is being requested by each process. The space of all possible system states constitutes a progress space analogous to that described above, and an action sequence of the system of processes generates a joint progress path. As before, there will be infeasible and unsafe regions. Coffman et al. have shown how to represent requests and releases by vectors, and system states by matrices [C1]. Holt has developed algorithms that examine a system state and decide whether or not it contains a deadlock, in a time linearly proportional to the number of competing processes [H4].

The most difficult aspect of the deadlock problem is the construction of algorithms that decide whether or not a given system state is safe; such algorithms would be useful in deciding whether granting the request of some process would put the system into an unsafe state. This problem is complicated by the requirement that some information about future resource usage of processes be available. Habermann has developed an algorithm for deciding the safeness of a state in a time linearly proportional to the number of processes, under the practical assumption that the maximum possible resource claim for each process is known in advance [H1].

Deadlock detection algorithms are used in a few contemporary systems, primarily in conjunction with peripheral device assignments [H2]. Since deadlock avoidance algorithms are a recent development, they have not yet come into wide use.

Mutual Exclusion

We stated earlier that some external control mechanism may be required to implement mutual exclusion, i.e., the requirement that, at most, one process have access to a given resource R at any one time, the order in which processes use R being immaterial. Examples of resources subject to mutual exclusion include: processors, pages of memory, peripheral devices, and writable data bases. Mutual exclusion of processors, memory, and other peripheral devices is normally provided by the system's multiplexing

mechanisms. However, there are many programming problems in which two programs may contain *critical sections* of code, i.e., sequences of instructions that must be performed by at most one program at a time because they reference and modify the state of some resource or data base [D12]. In these cases, a mechanism is required for implementing mutual exclusion at the software level.

The simplest such mechanism is provided by memory "lockout" instructions. For a given address x, this mechanism takes the form of machine instructions **lock** x and **unlock** x, whose operations are:

> **lock:** [**if** $x = 1$ **then** $x \leftarrow 0$ **else goto lock**]
> **unlock:** [$x \leftarrow 1$]

The brackets indicate that the enclosed actions are *indivisible*, i.e., attempting to execute more than one **lock** or **unlock** at the same time has the same effect as executing them in some order, one at a time. If the actions were not indivisible, two processors could, for example, find $x = 1$ and pass the **lock** instruction, whereas the intended effect is that only one should pass. It is important to note that the successful operation of these instructions normally depends on the fact that memory-addressing hardware arbitrates accesses to the same location, permitting one reference per memory cycle. The mutual exclusion already implemented at the lower (hardware) level is made available by the **lock** and **unlock** instructions for implementation at the higher (software) level.

To implement mutual exclusion with respect to some critical section using some resource R, a memory cell x initially containing value 1 is allocated; the programming is:

IN PROCESS P_i	IN PROCESS P_j
⋮	⋮
lock x;	**lock** x;
critical section;	critical section;
unlock x;	**unlock** x;
⋮	⋮

If both P_i and P_j attempt to access the critical section together, one will succeed in setting $x = 0$ and the other will loop on the **lock** instruction; therefore, at most one of P_i and P_j can be positioned between the instructions **lock** and **unlock**. This solution generalizes immediately to more than two processes.

Even though this is a correct solution to the mutual exclusion problem, it has three limitations. First, a processor cannot be interrupted while performing a critical section, else the variable x will remain locked and other processes may be denied the use of R for an indefinite period. Accordingly, many implementations of **lock** disable interrupts while **unlock** enables them, and some time limit on the duration of a critical section is imposed lest the processor remain uninterruptible for an indefinite period. Secondly, the solution uses the *busy form of waiting* [D13]; i.e., a processor may loop on the **lock** instruction. A more desirable mechanism would allow the processor to be preempted and reassigned. Thirdly, the solution is not "safe" if programs are cyclic; i.e., it is possible for one processor looping on the **lock** to be blocked indefinitely from entering the critical section because a second processor looping through its program may pass the **lock** instruction arbitrarily often. For these reasons the **lock** and **unlock** instructions are not normally available to users. Instead, a generalized form of **lock** and **unlock** are used, and implemented as operating system routines called by interrupts. Although the generalized forms solve the second and third problems, a timer is still required to deal with the first.

Dijkstra has defined generalized **lock** and **unlock** using the concept of *semaphore*, a variable used in interprocess signaling [D13]. A semaphore is an integer variable s with an initial value $s_0 \geqslant 0$ assigned on creation, associated with it is a queue Q_s, in which are placed the identifiers of processes waiting for the semaphore to be "unlocked." Two indivisible operations are defined on a semaphore s:*

wait s: [$s \leftarrow s - 1$; if $s < 0$ the caller places himself in the queue Q_s, enters the waiting state, and releases the processor]

* Dijkstra uses P and V for **wait** and **signal**, respectively. (The more descriptive names used here have been suggested by P. Brinch Hansen and A. N. Habermann.) OS/360 uses the ENQ and DEQ macros, which operate on the queue rather than the semaphore.

signal s: [$s \leftarrow s + 1$; if $s \leqslant 0$ remove some process from Q_s and add it to the work queue of the processors]

Semaphore values may not be inspected except as part of the **wait** and **signal** operations. If $s < 0$, then $-s$ is the number of processes waiting in the queue Q_s. Executing **wait** when $s > 0$ does not delay the caller, but executing **wait** when $s \leqslant 0$ does, until another process executes a corresponding **signal**. Executing **signal** does not delay the caller. The programming for mutual exclusion using **wait** and **signal** is the same as for **lock** and **unlock**, with $x_0 = 1$ (**wait** replaces **lock**, and **signal** replaces **unlock**). By definition, the second problem with **lock** and **unlock** is solved by **wait** and **signal**; the third problem can be solved by using a first-in first-out discipline in queue Q_s.

Synchronization

In a computation performed by cooperating processes, certain processes may not continue their progress until information has been supplied by others. In other words, although program-executions proceed asynchronously, there may be a requirement that certain program-executions be ordered in time. This is called synchronization. The precedence constraints existing among processes in a system express the requirement for synchronization. Mutual exclusion is a form of synchronization in the sense that one process may be blocked until a signal is received from another. The **wait** and **signal** operations, which can be used to express all forms of synchronizations, are often called *synchronizing primitives*.

An interesting and important application of synchronization arises in conjunction with cooperating cyclic processes. An example made famous by Dijkstra [D13] is the "producer/consumer" problem, an abstraction of the input/output problem. Two cyclic processes, the producer and the consumer, share a buffer of $n > 0$ cells; the producer places items there for later use by the consumer. The producer might, for example, be a process that generates output one line at a time, and the consumer a process that operates the line printer. The producer must be blocked from attempting to deposit an item into a full buffer, while the consumer must be blocked from attempting to remove an item from an empty buffer. Ignoring the details of producing, depositing, removing, and consuming items, and concentrating solely on synchronizing the two processes with respect to the conditions "buffer full" and "buffer empty," we arrive at the following abstract description of what is required. Let $a_1a_2 \cdots a_k \cdots$ be a system action sequence for the system consisting of the producer and consumer processes. Let $p(k)$ denote the number of times the producer has deposited an item among the actions $a_1a_2 \cdots a_k$, and let $c(k)$ denote the number of times the consumer has removed an item from among the actions $a_1a_2 \cdots a_k$. It is required that

$$0 \leqslant p(k) - c(k) \leqslant n \qquad (1)$$

for all k. The programming that implements the required synchronization (Eq. 1) is given below; x and y are semaphores with initial values $x_0 = 0$ and $y_0 = n$:

```
pro: produce item;        con: wait x;
     wait y;                   remove item;
     deposit item;             signal y;
     signal x;                 consume item;
     goto pro;                 goto con;
```

To prove that Eq. 1 holds for these processes, suppose otherwise. Then either $c(k) > p(k)$ or $p(k) > c(k) + n$. However, $c(k) > p(k)$ is impossible since it implies that the number of completed **wait** x exceeds the number of completed **signal** x, thus contradicting $x_0 = 0$. Similarly, $p(k) > c(k) + n$ is also impossible since it implies that the number of completed **wait** y exceeds by more than n the number of completed **signal** y, thus contradicting $y_0 = n$.

Another application of synchronization is the familiar "ready-acknowledge" form of signaling, as used in sending a message and waiting for a reply [B5]. Define the semaphores r and a with initial values $r_0 = a_0 = 0$, the programming is of the form:

```
IN THE SENDER             IN THE RECEIVER
      ⋮                         ⋮
generate message;         wait r;
signal r;                 obtain message;
wait a;                   generate reply;
obtain reply;             signal a;
      ⋮                         ⋮
```

The synchronizing primitives can be used to implement the precedence constraints among the members of a system of processes. Whenever P_i precedes P_j in a system, we may define a semaphore x_{ij} with initial value 0, suffix to the program of P_i the instruction **signal** x_{ij}, and prefix to the program of P_j the instruction **wait** x_{ij}. (Implementations with fewer semaphores can be found, but this one is easiest to explain.) Letting S_1, S_2, S_3, and S_4 denote the statements of the programs of the four processes in Figure 6, the system of Figure 6 can be implemented as follows:

```
P1 . begin; S1, signal x12 ; end
P2   begin; wait x12, S2 ; signal x24 . end
P3 . begin; S3 , signal x34 , end
P4 . begin; wait x24 , wait x34 , S4 , end
```

As a final example, let us consider how the synchronizing primitives can be used to describe the operation of an interrupt system. Typically, the interrupt hardware contains a set of pairs of flipflops, each pair consisting of a "mask flipflop" and an "interrupt flipflop." The states of the flipflops in the ith pair are denoted by m_i and x_i, respectively. The ith interrupt is said to be "disabled" (masked off) if $m_i = 0$, and "enabled" if $m_i = 1$. When the hardware senses the occurrence of the ith exceptional condition C_i, it attempts to set $x_i = 1$, if $m_i = 0$, the setting of x_i is delayed until $m_i = 1$. The setting of x_i is supposed to awaken the ith "interrupt-handler process" H_i, in order to act on the condition C_i. By regarding m_i and x_i as hardware semaphores with initial values $m_i = 1$ and $x_i = 0$, we can describe the foregoing activities as an interprocess signaling problem:

```
In Hardware
Ci occurs: wait mi ;
           signal xi ;
           signal mi ;
  disable: wait mi ,
   enable: signal mi ,

In Interrupt Handler Hi
 start. wait xi ;
        process interrupt;
        goto start,
```

The foregoing is, of course, intended more as an illustration of the interprocess signaling aspects of interrupt systems than as an accurate description of interrupt hardware operation.

RESOURCE ALLOCATION

The purpose of automatic (system-controlled) resource allocation is to regulate resource usage by processes in order to optimize given measures of system efficiency and good service throughout the user community. Though it is possible to develop the mechanisms of resource sharing first and the policies for using them later, desirable policies may require special mechanisms. In other words, the system designer must know which general class of resource allocation policies will be used before he specifies the final choice of mechanism.

Resource allocation policies are generally of two kinds: short-term policies, which are for making decisions on a time scale comparable to or slower than human reaction times, and long-term policies. Long-term policies tend to be used for economic considerations, such questions as: "How does one forecast the demand on the system?" "How does one charge for resource usage?" and "What is the role of pricing policy in controlling demand?" Pricing policies have been found to be important as a tool for rationing resources and as a means for controlling the tendency for users to develop countermeasures that render short-term policies ineffective [C2]. Since long-term policies are of more concern to system administrators than to system designers, we shall not pursue them further here. Good discussions of the various issues can be found in [M4, N1, S5, S6, S7, W8].

The term "automatic resource allocation" refers to the study and implementation of short-term policies within the operating system. Automatic resource allocation is necessitated—and complicated—by various factors, including:

1) system designers' consistent endeavors to relieve programmers of the burdens of resource allocation by transferring those burdens to the operating system;
2) the need to control the interference and

interaction among processes that results from the sharing of equipment by concurrent processes;

3) multitasking, which exposes the system to the danger of deadlock; and

4) the need to regulate the competition for resources because the instantaneous total demand of the user community is time-varying, subject to random influences, and may exceed the supply of resources, thus interfering with overall system performance.

These factors are the motivation for a theme to be developed in the subsequent discussion: management of processors, memory, and other resources cannot be handled by separate policies; they are aspects of a single, large resource allocation problem.

The goals of short-term policies are of two distinct and often conflicting types: efficiency and user satisfaction. The former include measures of throughput, resource utilization, and congestion on information-flow paths, the latter include measures of response time, turnaround time, and funds expended to run a job. Figure 9 illustrates the conflict between a measure of user satisfaction (waiting time) and a measure of efficiency (fraction of time a processor is idle) such as might be encountered in a simple, monoprogrammed time-sharing system. The conflict follows from a result in queueing theory, which asserts: the more likely it is that the processor queue is nonempty (and therefore the processor is not idle), the more likely it is that the queue is long (and therefore a user must wait longer for service). (Indeed, if α is the idleness of the processor, the expected waiting time in queue is proportional to $(1 - \alpha)/\alpha$ [M4].)

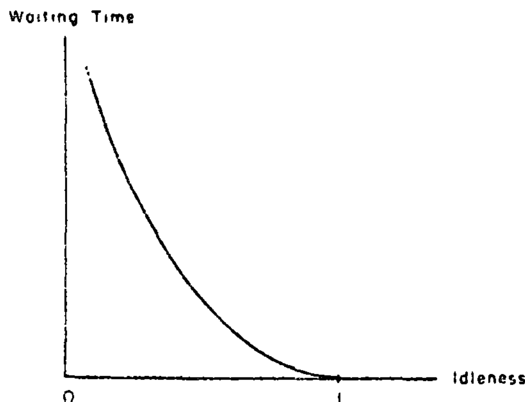


Fig. 9 A conflict between satisfaction and efficiency

There may be other sources of conflict than service objectives. For example, in systems where deadlock avoidance routines are used, an allocation state is deemed "safe" if there exists a schedule ordering resource requests and releases of processes such that all may be run to completion without deadlock. The designer must be careful that such schedules are consistent with any pre-existing priority rules; otherwise, a deadlock may be generated by a conflict between the order of processes specified in the deadlock-avoidance schedule and the order specified by the priorities.

A Resource Allocation Model

Most systems are organized so that, at any given time, a process will be in one of three *demand-states*: 1) in the *ready* state it is demanding, but not receiving the use of, processor and memory resources; 2) in the *active* state the working set of the process resides in main memory; and 3) in the *blocked* state it is not demanding use of processor or memory, but is either demanding to use peripheral equipment or waiting for a message from another process. The allowable transitions among the states are shown in Figure 10. These demand-states are reflected in the organization of queues in computer systems, giving rise to a network of queues with feedback. In general, there are three subnetworks corresponding to each of the three states: 1) ready processes are distributed among "levels" of a queue, each level corresponding to a prediction of future resource usage; 2) active processes may be partitioned into "young" and "old," depending upon whether they were recently initiated or not (they may also be partitioned into executing and page-waiting processes in a virtual memory system); and 3) blocked processes may be partitioned according to the reasons for their being blocked. In all three cases, the subdivisions within a given class reflect various demand substates of interest.

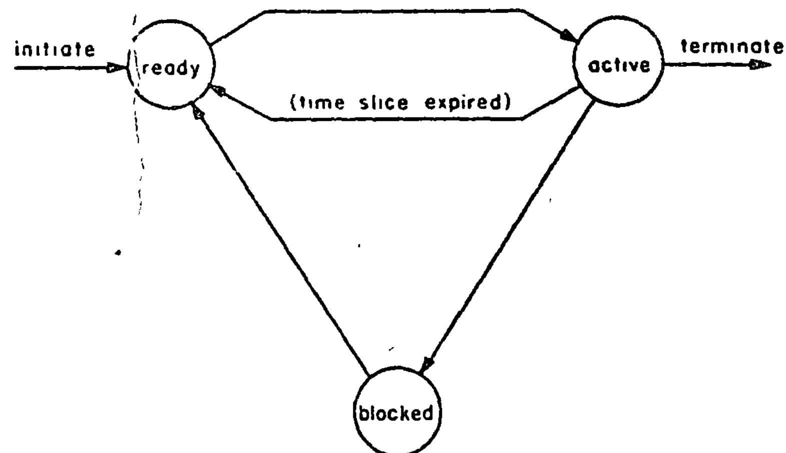


FIG. 10. Process demand-states.

The extensive literature on "scheduling algorithms," accumulated before multiprogramming became predominant, ignores the blocked state of Figure 10. Good surveys of these methods are found in [C2, M4], and a unified view of their operation in [K3]. These early models have been found to be of limited applicability in multiprogramming systems since they treat systems with only a single point of congestion (e.g., processor), whereas Figure 10 implies a system with many points of congestion (e.g., auxiliary devices, main memory, console input/output routines, etc.). In fact, memory management is often more critical than processor scheduling because processors may be switched among processes much more rapidly than information among memory levels. Thus, the applicability of these models is limited by their failure to reflect accurately job flow and system structure, not by their frequent use of exponential assumptions in analyses [F1]. But despite their shortcomings, they have made an undeniable contribution to our present understanding of resource allocation.

Successful analyses of queueing networks representative of Figure 10 have demonstrated their applicability to modern systems, and have shown that resource allocation policies may be regarded as job-flow regulation policies [B6, K2]. This approach promises to yield insight into optimal policies.

The working-set model for program behavior can be used to gain insight into those aspects of the resource allocation problem dealing with the flow between the ready and active states. It illustrates what appears to be a manifestation of a more general principle: there may exist situations in which a process must be allocated a critical amount of one resource type before it can use any resources effectively. Specifically, a process cannot be expected to use a processor efficiently if its working set is not loaded in main memory. This fact has two implications with respect to multiprogramming. First, attempted overcommitment of main memory in core-drum systems may result in the overall collapse of system processing efficiency, which is known as thrashing [D4, D6]. This situation is illustrated in Figure 11. Without the aid of the working-set model, one might conjecture that processing efficiency approaches unity as the degree of multiprogramming n (number of active processes) increases; but, in fact, processing efficiency exhibits a marked dropoff when the degree of multiprogramming exceeds a certain critical level n_0. The explanation of this behavior is simply that, when $n > n_0$, there is no longer sufficient space in memory to accommodate all active working sets. The second implication with respect to multiprogramming is closely related to the first. As suggested in Figure 12, the cost of wasted (unused) memory decreases with n, and

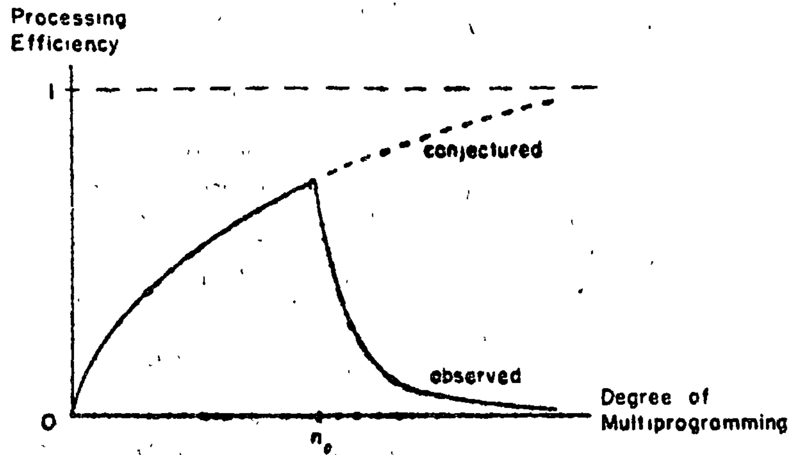


FIG. 11. Thrashing.

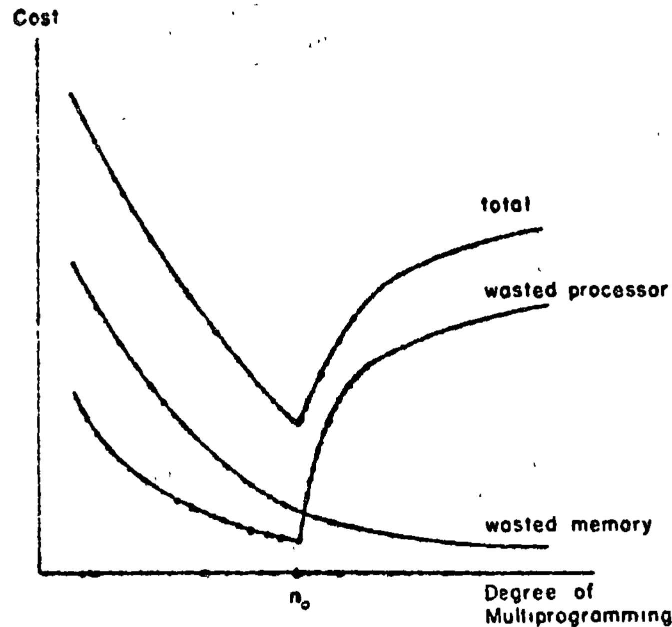


FIG. 12 Cost of multiprogramming.

the cost of wasted processor follows a curve whose shape is similar to the one in Figure 11; at some point in the vicinity of n_0 the total cost is minimum. Thus, it is desirable to load as many working sets as possible into memory; this approximates an optimal degree of multiprogramming.

System Balance

The foregoing considerations illustrate a general phenomenon: overall system performance may suffer if resource usage is not balanced. Stated differently, over- (or under-) commitment of one resource may result in others being wasted. "Balance" can be given a precise definition as follows. At each moment of time, each process has associated with it a vector, its *demand*, whose components reflect the amount of each resource type required by the process at that time. Thus, process i will have a demand $\mathbf{d}_i = (d_{i1}, \cdots, d_{im})$ in which d_{ij} is

the amount of type j resource it requires. The demands $\mathbf{d}_i$ are random variables. The *total demand* $\mathbf{D}$ of the set A of active processes is the vector sum over the demands of all processes i in A:

$$\mathbf{D} = \sum_i \mathbf{d}_i = (\sum_i d_{i1}, \cdots, \sum_i d_{im}). \quad (1)$$

The total demand $\mathbf{D}$ is also a random variable. Two vectors are associated with the resources, $\mathbf{p} = (p_1, \cdots, p_m)$ and $\mathbf{c} = (c_1, \cdots, c_m)$, in which p_j is an allowable overflow probability and c_j a capacity of resource type j. The system is "safely committed" if the probability that the total demand $\mathbf{D}$ exceeds the system capacity $\mathbf{c}$ is bounded by the probabilities $\mathbf{p}$; i.e.,

$$\Pr[\mathbf{D} > \mathbf{c}] \leqslant \mathbf{p}. \quad (2)$$

The system is "fully committed" if adding one more ready process to the active set A would violate condition (2). If the system is overcommitted, then for some resource type j,

$$\Pr[\sum_{i \text{ in } A} d_{ij} > c_j] > p_j. \quad (3)$$

The system is *balanced* with respect to a given degree of multiprogramming (size of A) if the capacities $\mathbf{c}$ have been chosen so that

$$\Pr[\mathbf{D} > \mathbf{c}] = \mathbf{p}. \quad (4)$$

Thus, system balance in the sense of Eq. (4) implies that: 1) the system is fully committed; and 2) resources are fully utilized to the extent allowed by the overflow probabilities. Under these conditions, A can be called the "balance set."

In the case of a system in which the designer is interested in balancing only a single processor and memory, Eq. (4) can be interpreted to mean: 1) the memory is filled with working sets to the extent allowed by the memory overflow probability; 2) processor time is committed so that response time can be guaranteed within some fixed time in the future, depending on processor overflow probability; and 3) the amount of memory is as small as possible. These points have been treated more fully in [D5]. It is interesting to note that Eq. (4) is not difficult to evaluate in practice, since the central limit theorem of probability theory applied to Eq. (1) asserts that the distributions of the components of $\mathbf{D}$ can be approximated by normal distributions.

Eqs. (3) and (4) show that a given system need not be balanced under arbitrary selections of equipment capacities, i.e., the equipment configuration. Only certain equipment configurations are consistent with system balance and given demand distributions of the user community. Other configurations would only give rise to wasted equipment.

Wilkes [W7] has shown that the efficiency of system operation can be improved by implementing the following policy within the balance set. The "life" of a process is the amount of processor time it consumes between interactions with the user or with other processes. The "age" of a process at a given time is the amount of processor time it has been observed to consume since its last interaction. One property of the process-life distribution is that the expected remaining life, given the age, increases monotonically with increasing age to a maximum. Thus, in order to minimize the probability of removing a page of a process from main memory shortly before the remaining life of that process expires (i.e., to give processes whose next interaction is imminent a chance to reach it rapidly), it is desirable to protect the pages of young processes, over those of old ones, from removal. To do this Wilkes proposes the partitioning of processes in memory into a "resident regime" (young processes) and a "swapping regime" (old processes). Similar considerations are made in CTSS [C4].

The foregoing is not the only possible way to approach system balance. Balancing a system has also been called "tuning" and "load leveling." Bernstein and Sharpe have formulated it in terms of regulating job flow so that measured resource-usage functions track desired resource-usage functions [B3]. Wulf has formulated it in a similar way, but with more attention to measuring usage rates and demands [W10]. Wilkes shows how control theory may be applied to achieving stability when the system attempts to balance itself by controlling the number of users accessing the system at any given time [W9].

Choosing a Pc[illegible]

The notions of system balance given above are quite flexible. In effect, they make a general statement about the total flow in and out of the balance set, and allow the resource allocation policy considerable latitude in choosing which ready processes shall be elements of that flow. In other words, balance can be viewed as a *constraint* within which a resource allocation policy can seek to optimize some measure of user satisfaction. It thus represents a means of compromising between the conflicting goals of efficiency and satisfaction.

Even with the aid of all these abstractions to give him insight, it remains the task of the system designer to determine which considerations and tradeoffs are important in the system at hand. In addition to the various modeling techniques outlined above, various simulation techniques are available to help him evaluate strategies [M1]. Modeling has the advantages of conceptual simplicity and flexibility, and the disadvantages of being sometimes incomprehensible to clients and of resting on oversimplified assumptions that may not reflect accurately the complexities of a given system. Simulation, on the other hand, has the advantages of being comprehensible and allowing realistic detail, the disadvantages of being time consuming and subject to programming and random errors. When optimal strategies are desired, simulation, although valuable for comparing alternatives, has the added limitation of not being useful in locating optima. A combination of simulation and analysis can be used to improve the efficacy of both [G2, K2].

PROTECTION

The protection of information and other objects entrusted to a computer system has been the subject of considerable concern, especially since data banks have become feasible. The term "protection" has been used in various ways, both technical and nontechnical. The subsequent discussion will deal exclusively with technical aspects, i.e., the techniques used within the system to control access by processes to system objects. The nontechnical aspects of protection—preventing systems programmers from reading dumps of sensitive material, checking the identity of users, keeping magistrates from misusing information stored in a data bank, and so on—will not be discussed here; the reader is referred to [H3] for a survey. Neither will the integrity problem—protection against loss of information due to accident, system failure, hardware error, or a user's own mistakes—be discussed here; the reader is referred to [W6] for a description.

The motivations for a protection system are well known. They concern the shielding of processes in the system from error or malice wrought by other processes. No process should be able to read, destroy, modify, or copy another process's information without authorization. No faulty process should be able to affect the correct operation of any other (independent) process. No user should be required to trust a proprietary program, nor should a proprietary program be required to trust its users [D11, L1, L2]. No debugging program should be required to trust the program it debugs.

The following treatment of protection is based on the abstract model for a protection system proposed by Lampson [L1, L2] and refined by Graham [G4]. Although this model is not the only way of studying the problem, it does have the property that many practical protection systems can be deduced from it.

The model for a protection system consists of three parts. First is a set X of *objects*, an object being any entity in the system that must be protected (i.e., to which access must be controlled). Examples of objects include files, segments, pages, terminals, processes, and protection keys. Each object has associated with it an identification number which is unique for all time, and identification numbers, once used, are never assigned again. (MULTICS, for example, uses the time in μsec starting from 1900 as identification numbers for files.) The second part of a protection system model is a set S of *subjects*, a subject being any entity that may access objects. A subject can be regarded as a pair (process, domain), where a "domain"

is a set of constraints within which the process may access certain objects. Examples of domains include supervisor user states in a machine, the blocks of the multiprogramming partition in the OS/360-MVT or the CDC 6600, and the file directories in MIT's CTSS. Various terms have been used to denote the idea of domain: protection context, protection state, sphere of protection [D11], and ring of protection [G3]. Since subjects must be protected from each other, subjects are also objects, i.e., S is contained in X. The third part of a protection system model is a collection of *access rules* specifying the ways in which a subject may access objects. A method of specifying access rules will be described shortly.

A protection system fulfills two functions: *isolation* and *controlled access*. Isolation refers to constraining each subject so that it can use only those objects to which access has been authorized. Controlled access refers to allowing different subjects to have different types of access to the same object. Isolation is very simple to implement, but controlled access is in fact the culprit complicating most protection systems. To illustrate this point, we shall first consider a protection system that implements isolation only; then we shall study how and why controlled access may be incorporated.

In the simplest case, each subject is a process. Each object is accessible to exactly one subject, i.e., the objects of the system are partitioned according to subjects. Each subject s has a unique identification number i_s. Subjects interact by sending messages. Each message consists of a string of data to which the system prefixes the identification number of the sender, all messages directed to a given subject are collected in a message buffer associated with that subject. When subject s wishes to transmit a message α to subject s', it calls on the system by an operation

send message (α, s');

and the system then deposits the pair (i_s, α) in the message buffer of s'. As will be seen, the principle that guarantees the correct operation of protection is that *the identification number of a sender subject cannot be forged*. In other words, even though the identification numbers may be known to everyone, there is no way for s to send α to s' that can alter the fact that message (i_s, α) is placed in the message buffer of s'.

This very simple protection system is complete from the viewpoint of isolation. Protected calling of subroutines (protected entry points [D11]) is implemented in the obvious way: if s wishes to "call" s', s sends s' a message consisting of the parameter list α and enters a state in which it waits for a reply message (return) from s'; when s' finds message (i_s, α) in its message buffer, it performs its function with parameters α, then replies to s with a message containing the results. There is no way that s can "enter" s' at any point other than the one at which s' inspects its message buffer. By a similar argument, subroutine return is protected. The protection implemented here goes beyond guaranteeing that subroutines call and return only at the intended points, it is clearly possible to guarantee that s' will act on behalf of authorized callers only, since s' can choose to ignore (or report to the supervisor) any message (i_s, α) from an unauthorized caller s. Access to an arbitrary object x in this system is also completely protected: a subject s may access freely any x associated with it; but if x is associated with another subject s', s must send a message to s' requesting s' to access x on its behalf. The RC-4000 system operates in precisely this fashion [B5].

The system described above has several limitations that motivate the addition of a controlled-access mechanism to the protection system:

1) it is not possible to stop a runaway process, since there is no way to force a receiver either to examine his message buffer or to do anything else—which, among other things, makes debugging difficult;
2) there is no systematic, efficient way of sharing objects among subjects; and
3) it necessitates the establishment of a possibly elaborate system of conventions to enable receivers to discover the identification numbers of authorized senders.

It is necessary, therefore, to generalize the foregoing system so that some subjects can

OBJECTS

	subjects			files		processes		terminals	
	s_1	s_2	s_3	f_1	f_2	p_1	p_2	t_1	t_2
SUBJECTS s_1	owner control	*owner		*read	read owner	wakeup	wakeup	read write	
s_2			control	*write	execute				read
s_3					write	stop		write	

* – copy flag set

FIG. 13. An access matrix.

exercise control over others, share objects with others, and discover access rights easily. In the generalization, the correctness of the protection system is guaranteed by a generalization of the nonforgery principle stated above: whenever subject s accesses object x, the access is accompanied by the identification number i_s (which cannot be forged), and the protection system permits access only if i_s has been authorized.

The generalized protection system requires an explicit specification of the access rules defining the access that subjects have to objects. (In the simple system, the access rules were implicit in the message transmitting mechanism.) These access rules can be specified in the form of an *access matrix*, A, whose rows correspond to subjects and whose columns correspond to objects. An entry $A[s, x]$ in this matrix contains a list of strings specifying the *access rights* that subject s has to object x. Thus, if string α is in $A[s, x]$, we say that s has access right α to x. For example, in Figure 13,

s_1 is the owner of s_1 and s_2 ;
s_1 may read f_1 ;
s_1 may wakeup p_1 ;
s_2 may execute f_2 ; and
s_3 may stop p_1 .

Each type of object has associated with it a *controller* through which accesses to objects of that type must pass. A subject calls on the appropriate controller when it wishes to access an object. For example:

TYPE OF OBJECT	CONTROLLER
subject	protection system
file	file system
segment or page	memory addressing hardware
process	process manager
terminal	terminal manager

An access would proceed as follows:

1) s initiates access of type α to object x;
2) the system supplies message (i_s, x, α) to the controller of x; and
3) the controller of x asks the protection system whether α is in $A[s, x]$; if so, access is allowed; otherwise, it is denied and a protection violation occurs.

It is important to note that the meaning of a string α in $A[s, x]$ is interpreted by the controller of x during each access to x, not by the protection system. The correctness of this system follows from the system's guarantee that the identification i_s cannot be forged and that the access matrix A cannot be modified except by rules such as those given below.

The three steps above concern the use of the access matrix once its entries have been specified. In addition to these, rules must exist for creating, deleting, and transferring access rights. The rules proposed by Lamp-

son [L2] use the access rights "owner," "control," and "copy flag" (*):

1) *Creating:* s can add any access right to $A[s', x]$ for any s' if s has "owner" access to x. For example, in Figure 13, s_1 can add "control" to $A[s_2, s_2]$ or "read" to $A[s_3, f_2]$.
2) *Deleting:* s can delete any access right from $A[s', x]$ for any x if s has "control" access to s'. For example, s_2 can delete "write" from $A[s_3, f_2]$ or "stop" from $A[s_3, p_1]$.
3) *Transferring:* if $*\alpha$ appears in $A[s, x]$, s can place either $*\alpha$ or α in $A[s', x]$ for any s'. For example, s_1 can place "read" in $A[s_2, f_1]$ or "*owner" in $A[s_3, s_2]$.

The copy flag is required so that one subject can prevent untrustworthy other subjects from maliciously giving away access rights that it has given them.

The three rules given above should be regarded as examples of rules for creating, deleting, and transferring. The exact nature of these rules will depend on the objectives of a given system. For example, according to Graham [G4], there is no need for "owner" and "control" to be distinct. Further, the notion of transferring can be extended to include a "transfer-only" mode according to which the transferred access right disappears from the transferring subject, thus, if the symbol ⊛ indicates this mode, then s can place $\circledast\alpha$ or α in $A[s', x]$ whenever $\circledast\alpha$ appears in $A[s, x]$, but, in so doing, $\circledast\alpha$ is deleted from $A[s, x]$. One may wish to limit the number of owners of an object to exactly one; assuming that each object initially has one owner, this condition can be perpetuated by allowing only "⊛owner" or "owner," but not "*owner."

A practical realization of a protection system can neither represent access rights as strings nor store the access matrix as a two-dimensional array. Access rights are encoded; for example, the entry in $A[s, x]$ might be a binary word whose ith bit is set to 1 if and only if the ith access right is present. The size of the access matrix can be reduced by grouping the objects X into disjoint subsets, called *categories* [G1], using the columns of matrix A to represent these subsets; thus, if α is in $A[x, Y]$ for category Y, then subject s has α access to each and every object in category Y. Storing the matrix A as an array is impractical since it is likely to be sparse. Storing it in the form of a table whose entries are of the form

$$(s, x, A[s, x])$$

is likely to be impractical in systems with many objects: the entire table cannot be kept in fast-access memory, especially since only a few subjects and objects are likely to be active at any given time, nor can it be guaranteed that a search for all x to which a given s has access (or all s having access to a given x) can be performed efficiently.

There are, however, several practical implementations. One of these stores matrix A by columns; i.e., each object x has associated with it a list, called an ***access control list***, with entries of the form $(s, A[s, x])$. When a subject attempts to access x, the controller of x can consult the access control list of x to discover $A[s, x]$. This is precisely the method used to control access to files in CTSS and to segments in MULTICS [B2, C8, D2]. A second method stores the matrix A by rows; i.e., each subject s has associated with it a list with entries of the form $(x, A[s, x])$, each such entry being called a *capability*, and the list a *capability list*. When a subject s attempts to access x, the controller of x can consult the capability list of s to discover $A[s, x]$. This is precisely the method proposed by Dennis and Van Horn [D11], and implementations of it are discussed in [L1, W6].

Of the two preceding implementations, the capability list appears to be more efficient than the access control list. In either case, the identification number of the current subject can be stored in a protected processor register, so that attaching it to any access is trivial. If the list to be consulted by an object controller is in a protected, read-only array in fast-access memory, the verification of access rights can be done quite easily. In the case of capability lists, the list has to be in fast-access memory whenever the subject is active. In the case of access control lists, however, the list need not be available in fast-access memory as a particular object

may not be active often enough to warrant keeping its list available at all times.

A third practical implementation is a refinement of the capability-list implementation in which each subject has associated with it a series of lists, one for each possible access right: the list of subject s for access right a, denoted by s_a, can be represented as a binary word whose ith bit is set to 1 if and only if subject s has access right a to the ith object x_i. When s attempts a access to object x_i, the controller checks to see whether or not the ith bit of s_a is 1. A combination of this and the category technique is used on the IDA time-sharing system and has been found quite efficient [G1].

A fourth possible implementation represents a compromise between the access-control-list and the capability-list implementations. Associated with each subject s is an object-key list consisting of pairs (x, k), where x is an object name and k is a "key," and this list is inaccessible to s. Each key k is an encoded representation of a particular access right a and (x, k) will be placed in the object-key list of s only if the a corresponding to k is in $A[s, x]$. Associated with each object x is a list of "locks," each lock being the encoded representation for some a. When s attempts to access x, the controller of x checks to see whether or not one of the keys of s matches one of the locks on x, and permits access only if a match is found. The system of "storage keys" and "locks" used in the OS 360 is a case of this approach.

In addition to the considerations above, tree-structured naming schemes can be introduced to make the process of locating an object more efficient. In the access-control-list implementation, the objects may be associated with the nodes of a tree and the access control lists attached to these nodes; the CTSS and MULTICS file systems exemplify this idea when the objects are files. In the capability-list implementation, the subjects may be associated with the nodes of a tree and the capability lists attached to these nodes; both the hierarchy of spheres of protection suggested by Dennis and Van Horn [D11] and the hierarchy of processes suggested by Brinch Hansen [B5] exemplify this idea. Implementing a subject-hierarchy has the additional advantage of defining protocol for reporting protection violations: a violation detected in a given subject is reported to the immediately superior subject for action.

How does memory protection, such as was discussed in the third section, "Storage Allocation," fit into this framework? Memory protection is enforced by the addressing and relocation hardware. Two properties of this hardware are exploited by protection systems: 1) only information in the range of an address map is accessible to a process, and 2) protection codes can be associated with each entry in a map and used to check the validity of each and every reference. The requirement that information associated with a given subject be protected can be realized by associating an address space with each subject.* For information object x, then, an entry in the address map f is of the form $(x, f(x), p)$, where p is a protection code. (The code p might contain bits giving read, write, or execute access to object x.) Thus, the address map is a form of a capability list, and the protection rules are enforced by the hardware. Sharing a segment among subjects is handled by putting common entries for that object in the address maps of the various subjects sharing it.

CONCLUSIONS

A few points are worth mentioning in closing. The concepts presented here have demonstrated their utility in the practical task of improving systems design and operation; in fact, most are straightforward generalizations of widely accepted viewpoints. These concepts have also demonstrated their utility in the classroom, easing, as they do, the task of explicating the often difficult principles of operating systems. However, these concepts can be considered, at best,

* Associating a separate address space with each subject is not necessarily efficient. It would, for example, force processes to pass symbolic names of segments or files in messages. These symbolic names would have to be converted to local address-space names in order to take advantage of memory addressing hardware.

only a beginning in the evolution of a "theory" of operating systems, for there remains a multitude of problems and issues for which no viable abstractions have yet been developed. Table II lists the more important of these, together with references to the literature that concerns them.

The importance of evolvable systems—ones that can expand or change indefinitely—appears to be widely preached, though rarely implemented. MIT's CTSS, an extensible system, is an excellent example of a system that evolved into a form very different (and considerably more useful) from that originally implemented. However, for a system to be evolvable, it is not sufficient that it be extensible, it requires a community atmosphere conducive to encouraging users to develop or improve on system services. In other words, evolution will not in fact occur if the users themselves neither gain from it nor contribute to it. A file system, or other program-sharing facility, is an essential catalyst in the evolutionary process.

Although the computing era is still young, it has evolved so rapidly that we have sometimes come to regard as obsolete any idea published more than five years ago. One is frequently struck by the similarities between our current problems and those faced by our predecessors. Many new problems are merely abstractions of old ones. The two most obvious examples are the systems programming language problem and the virtual storage allocation problem. Thus, the importance of being familiar with some of the old ideas—so that we can avoid retracing already-trodden paths—cannot be overstated. (Of course, not every current problem is an abstraction of an old one; the parallel process control problem and the protection problem, for example, are new.)

Two factors that have pervaded and flavored approaches to software design since the mid-1950s are the existence of magnetic core main memory and moving-medium auxiliary memory (e.g., drums, tapes). Three properties of memory systems using these devices have been particularly influential: 1) the time to access a word in main memory is of an order of magnitude slower than logic speeds; 2) the locations of core memory are organized in a linear array; and 3) the speed ratio between auxiliary and main memory is on the order of 10,000 or more. The introduction of new memory technologies (of which the cache store is the first step) is going to remove some or all of these assumptions, perhaps precipitously, and is going to require an extensive re-evaluation of all aspects of computer architecture depending on them. System designers will require abstractions of the kinds presented here, both for assessing the effects of the change in technology on existing designs and for developing efficient implementations of the same abstractions in the new technology.

TABLE II Literature on Problems Requiring Abstractions

Issue	*References*
Administration	N1, P1, S5, S6, S7, W6, W8
Compatibility	C3, C6
Complexity	C3, D14, H5, S1
Data communications, networks and utilities	A3, A4, D8, P1
Design techniques and correctness	B5, D9, D14
Evolvability	D9, D10
Generality	C3, D9
Performance evaluation	A7, A9, W10
Reliability and recovery	W6
System objectives	A8, B5, C4, C5, C7, D9, D10, I1, P1, W2, W6
Transportability of software	P2, W1

ACKNOWLEDGMENTS

I have profited from discussions with many individuals while preparing this paper. I should particularly like to thank Carol Shanesy (New York City Rand Institute); Dan Berry (Brown University); Edward G. Coffman, Jr. (Penn State University); Jack B. Dennis (MIT); Robert M. Keller (Princeton University); Butler W. Lampson (Xerox Palo Alto Research Center); Peter Wegner (Brown University); and Maurice V. Wilkes (Cambridge University). Most of all, however, I am indebted to the referees.

ANNOTATED BIBLIOGRAPHY

A1 Abate, J.; and H. Dubner. "Optimizing the performance of a drum-like storage." *IEEE Trans. Computers* C-18, 11 (Nov. 1969), 992-997.

A description and analysis of the shortest-access-time-first queueing discipline, which minimizes the accumulated latency time of requests for rotating-medium auxiliary stores.

A2. ACM. "Storage Allocation Symposium" (Princeton, N.J., June 1961). *Comm. ACM* 4, 10 (Oct 1961).

These papers argue the cases and techniques for static and dynamic storage allocation. They provide a picture of the state of considered thought on storage allocation as of 1961

A3. ACM. "Proc. Symposium on Operating System Principles" (Gatlinburg Tenn., Oct. 1967). Some of the papers appear in *Comm. ACM* 11, 5 (May 1968).

These papers provide, collectively, a summary of the ideas through the mid 1960s. The session titles are: Virtual Memory, Memory Management, Extended Core Memory Systems, Philosophies of Process Control, System Theory and Design, and Computer Networks and Communications.

A4. ACM. *Proc. Symposium on Problems in the Optimization of Data Communications Systems* (Pine Mt., Ga , Oct. 1969).

These papers provide a summary of ideas on computer data communications and networks as of 1969. The session titles are: Advanced Networks; Advances in Data Communications Technology; Preprocessors for Data Communications; Legal, Political, and Privacy Considerations; Graphic Displays and Terminals, Systems Optimization; Human Factors, and Software Implications.

A5. ACM. *Proc. 2nd Symposium on Operating Systems Principles.* (Princeton Univ., Oct 1969).

These papers provide, collectively, a summary of ideas through the late 1960s The session titles include General Principles of Operating Systems Design; Virtual Memory Implementation, Process Management and Communications, Systems and Techniques, and Instrumentation and Measurement.

A6 ACM. *Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation* (Wood's Hole, Mass , June 1970).

These papers deal with various aspects of parallel computation, such as Petri Net theory, program schemata, and speed-independent computation. A complete bibliography of pertinent literature is included.

A7 ACM. *Proc. Symposium on System Performance Evaluation* (Harvard Univ , Cambridge, Mass , April 1971).

These papers deal with various aspects of the performance evaluation problem The session titles are Instrumentation, Queueing Theoretic Models, Simulation Models, Measurement and Performance Evaluation, and Mathematical Models

A8. Alexander, M. T. "Time sharing supervisor programs (notes)." Univ Michigan Computer Center, May 1969.

A comparison of four systems: the Michigan Time Sharing System (MTS), Multics, CP/67, and TSS/360.

A9. Arden, B. W.; and D. Boettner. "Measurement and performance of a multiprogramming system." In *Proc 2nd Symposium on Operating Systems Principles* [A5], 130-146.

A description of simple but useful performance evaluation techniques used on the Michigan Time Sharing System.

B1. Belady, L. A. "A study of replacement algorithms for virtual storage computers" *IBM Systems J.* 5, 2 (1966), 78-101.

A detailed empirical study covering properties of program behavior manifested through demand paging algorithms Various algorithms are compared for different memory and page sizes. An optimal algorithm, MIN, is proposed.

B2. Bensoussan, A.; C. T. Clingen, and R. C. Daley. "The Multics virtual memory." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 30-42.

A description of the motivations for, and design of, the segmented address space and directory structure used in Multics.

B3 Bernstein, A. J ; and J Sharpe "A policy driven scheduler for a time sharing system." *Comm. ACM* 14, 2 (Feb 1971), 74-78

A proposal for resource allocation policy that attempts to allocate resources so that a measure of observed progress "tracks" a measure of desired progress.

B4 Brinch Hansen, P. (Ed.). *RC-4000 software multiprogramming system* A/S Regnecentralen, Falkoner Alle 1, Copenhagen F, Denmark, April 1969

The description of the RC-4000 system its philosophy, organization, and characteristics

B5. Brinch Hansen, P "The nucleus of a multiprogramming system." *Comm. ACM* 13, 4 (April 1970), 238-241, 250

A specification of the bare minimum requirements of a multiprogramming system It is related to the THE system [D14] in its orientation toward providing an efficient environment for executing parallel processes. See also [B4].

B6. Buzen, J. "Analysis of system bottlenecks using a queueing network model " In *Proc Symposium on System Performance Evaluation* [A7], 82-103

An analysis of a cyclic queue network for a computer system with a single central processor (round-robin scheduling) and a collection of peripheral processors Some results concerning optimal job flows are given

C1 Coffman, E G , Jr , M Elphick; and A Shoshani. "System deadlocks" *Computing Surveys* 3, 1 (June 1971), 67 78

A survey of the methods for preventing detecting, and avoiding deadlocks. Includes

summaries of the important analytical methods for treating deadlocks

C2 COFFMAN, E. G., JR.; AND L KLEINROCK "Computer scheduling methods and their countermeasures" In *Proc AFIPS 1968 SJCC*, Vol. 32, AFIPS Press, Montvale, N.J., 11–21.

A review of the basic time-sharing scheduling models and results, and of ways to gain high priority for service.

C3 CORBATÓ, F. J "PL/I as a tool for systems programming." *Datamation* 15, 5 (May 1969), 68-76.

Argues the case for a high-level systems programming language, and recounts the MULTICS experience with PL/I

C4. CORBATÓ, F. J; M MERWIN-DAGGET; AND R. C DALEY. "An experimental time sharing system" In *Proc AFIPS 1962 SJCC*, Vol 21, Spartan Books, New York, 335-344. Also in [R2].

Overview of the CTSS (Compatible Time Sharing System) at MIT, including the proposal for the multilevel feedback queue.

C5. CORBATÓ, F. J., AND V. A. VYSSOTSKY. "Introduction and overview of the MULTICS system." In *Proc AFIPS 1965 FJCC*, Vol 27, Pt. 1, Spartan Books, New York, 185–196. Also in [R2].

A description of the hardware and software facilities for the MULTICS (MULTiplexed Information and Computing Service) system at MIT.

C6. CREECH, B. A. "Implementation of operating systems." In *IEEE 1970 Internatl. Convention Digest*, IEEE Publ 70-C-15, 118–119.

Stresses the importance of a programming language for systems programming, and reports experience with the Burroughs B6500

C7. CRITCHLOW, A J "Generalized multiprogramming and multiprocessing systems" In *Proc. AFIPS 1963 FJCC*, Vol. 24, Spartan Books, New York, 107-126

A review tracing the evolution of multiprogramming and multiprocessing techniques and concepts through the early 1960s.

C8. CRISMAN, P. A. (ED). *The compatible time sharing system: a programmer's guide.* MIT Press, Cambridge, Mass., 1965

The complete description and specification of CTSS.

D1. DALEY, R. C.; AND J B. DENNIS. "Virtual memory, processes, and sharing in MULTICS." *Comm ACM* 11, 5 (May 1968), 306–312 Also in [A3].

A description of the hardware and software facilities of the GE-645 processor which are used to implement virtual memory, segment linkages, and sharing in MULTICS.

D2. DALEY, R. C.; AND P. G. NEUMANN "A general-purpose file system for secondary storage." In *Proc. AFIPS 1965 FJCC*, Vol. 27, Pt 1, Spartan Books, New York, 213–229.

Describes a file directory hierarchy structure, its use and conventions, and implementation problems.

D3 DENNING P J. "The working set model for program behavior." *Comm ACM* 11, 5 (May 1968), 323-333 Also in [A3].

A machine-independent model for program behavior is proposed and studied.

D4 DENNING, P. J. "Thrashing: its causes and prevention" In *Proc. AFIPS 1968 FJCC*, Vol 33, Pt. 1, AFIPS Press, Montvale, N.J., 915–922

A definition of, and explanation for, thrashing (performance collapse due to overcommitment of main memory) is offered, using the working set model [D3] as a conceptual aid Some ways of preventing thrashing are discussed.

D5. DENNING, P. J. "Equipment configuration in balanced computer systems" *IEEE Trans Computers* C-18, 11 (Nov. 1969), 1008 1012

Some aspects of designing systems with understandable behavior are defined and applied to the equipment configuration problem

D6. DENNING, P. J. "Virtual memory." *Computing Surveys* 2, 3 (Sept. 1970), 153-189

A survey and tutorial covering the definitions, implementations, policies, and theory of automatic storage allocation.

D7. DENNIS, J. B. "Segmentation and the design of multiprogrammed computer systems" *J ACM* 12, 4 (Oct. 1965), 589-602. Also in [R2]

The segmented name space and addressing mechanism are proposed and developed. Comparisons with other name space structures are made.

D8. DENNIS, J. B. "A position paper on computing and communications" *Comm. ACM* 11, 5 (May 1968), 370–377. Also in [A3].

The problems that face computer networks are outlined and some solutions prop[illegible]

D9 DENNIS, J. B. "Programming generality, parallelism, and computer architecture" In *Proc. IFIP Cong. 1968*, Vol. 1, North H[illegible] Publ. Co, Amsterdam, 484–492. (Also in MIT Project MAC Computation Structures Group Memo No. 32.)

Defines "programming generality" [illegible] ity to construct programs from coll[illegible] modules whose internal operations [illegible] known—and explores its consequen[illegible] system design.

D10 DENNIS, J. B. "Future trends in tim[illegible] ing systems." MIT Project MAC Comp[illegible] Structures Group Memo No 36-1, June [illegible]

Classifies time-sharing systems and [illegible] sizes the importance of progr[illegible] ity in achieving the "information utilit[illegible] in solving the "software problem"

D11. DENNIS, J. B.; AND E. C VAN HORN "Pro[illegible] gramming semantics for multiprog[illegible] computations." *Comm. ACM* 9, 3 (March [illegible]), 143-155.

The system requirements for impl[illegible] segmentation, protection, sharing, [illegible] lelism, and multiprogramming are d[illegible] in terms of "meta-instructions" The con[illegible] cept of "capability" is introduced

D12. DIJKSTRA, E. W. "Solution of a problem in concurrent programming control." *Comm. ACM* 8, 9 (Sept. 1965), 569.

This is the first published solution to the mutual exclusion problem, programmed in standard ALGOL without lock and unlock machine instructions being available.

D13. DIJKSTRA, E. W. "Cooperating sequential processes." In *Programming languages*, F. Genuys (Ed.), Academic Press, New York, 1968, 43-112.

The first study of parallel processes and concurrent programming, including a well-illustrated and complete development of the minimum requirements on the coordination and synchronization control primitives.

D14. DIJKSTRA, E. W. "The structure of THE-multiprogramming system." *Comm. ACM* 11, 5 (May 1968), 341-346. Also in [A3].

An overview of a process-oriented, multilevel, highly organized operating system designed so that its correctness can be established a priori. Includes an appendix on the synchronizing primitives *P* and *V*. (THE—Technische Hogeschule Eindhoven)

F1. FUCHS, E.; AND P. E. JACKSON. "Estimates of random variables for certain computer communications traffic models." *Comm. ACM* 13, 12 (Dec. 1970), 752-757.

An empirical study of interarrival and service distributions frequently encountered in computer systems. The exponential and gamma distributions are found to be useful in many cases of practical interest.

G1. GAINES, R. S. "An operating system based on the concept of a supervisory computer." In *Proc. Third Symposium on Operating Systems Principles*, to appear in *Comm. ACM* 15, 3 (March 1972).

A description of the structure of the operating system on the CDC 6600 computer at IDA (Institute for Defense Analyses), Princeton, N.J. The design is related to that of the RC-4000 [B4, B5].

G2. GAVER, D. P. "Statistical methods for improving simulation efficiency." Carnegie-Mellon Univ., Pittsburgh, Pa., Management Science Research Group Report No. 169, Aug. 1969.

Discusses a number of methods for shortening the computation time of simulations by improving the rate of convergence.

G3. GRAHAM, R. M. "Protection in an information processing utility." *Comm. ACM* 11, 5 (May 1968), 365-369. Also in [A3].

A description of the "ring" protection structure in MULTICS.

G4. GRAHAM, G. S. "Protection structures in operating systems." Master of Science Thesis, Univ. Toronto, Toronto, Ont., Canada, Aug. 1971.

A detailed investigation of Lampson's metatheory [L2] and its system implications, together with its application to Project SUE at the University of Toronto.

H1. HABERMANN, A. N. "Prevention of system deadlock." *Comm. ACM* 12, 7 (July 1969), 373-377.

Definition of the deadlock problem, and the "banker's algorithm" method of determining safe allocation sequences.

H2. HAVENDER, J. W. "Avoiding deadlock in multitasking systems." *IBM Systems J.* 7, 2 (1968), 74-84.

Treats some of the deadlock problems encountered in OS/360, and proposes the "ordered resource usage" method of preventing deadlock.

H3. HOFFMAN, L. J. "Computers and privacy: a survey." *Computing Surveys* 1, 2 (June 1969), 85-103.

An overview of the philosophical, legal, and social implications of the privacy problem, with emphasis on technical solutions.

H4. HOLT, R. C. "Deadlock in computer systems." PhD Thesis, Rep. TR-71-91, Cornell Univ., Dept. Computer Science, Ithaca, N.Y., 1971.

An exposition of the deadlock problem for systems consisting of both reusable and consumable resources. The emphasis is on graphical methods of description and on efficient algorithms for detecting and preventing deadlocks.

H5. HORNING, J. J.; AND B. RANDELL. "Structuring complex processes." IBM Thomas J. Watson Research Center Report RC-2459, May 1969.

A formal, state-machine-oriented model for the definition, control, and composition of processes.

I1. IBM. "Operating System 360 concepts and facilities." In *Programming Systems and Languages* [R2], 598-646.

Excerpts from IBM documentation on the structure and operation of OS/360.

I2. IEEE. "Proc. 1969 Computer Group Conference." *IEEE Trans. Computers* C-18, 11 (Nov. 1969).

Of interest is the session on "Computer system models and analysis."

IRONS, E. T. "Experience with an extensible language." *Comm. ACM* 13, 1 (Jan. 1970), 31-40.

Another example of achievable success in designing a time-sharing operating system (for the CDC 6600) using an extensible, high-level language.

J1. JOHNSTON, J. B. "The contour model of block structured processes." In *Proc. Symposium on Data Structures in Programming Languages* (Gainesville, Fla., Feb. 1971), J. T. Tou & P. Wegner (Eds.), special issue of *SIGPLAN Notices*, ACM, New York, Feb. 1971.

The contour model is primarily a pictorial way of describing the properties, operation, and implementation of processes generated by programs written in block structured programming languages. A "contour" is the boundary of a local environment of an activated block in the diagrams.

K1 KILBURN, T., D. B. G. EDWARDS, M. J. LANIGAN; AND F. H. SUMNER. "One-level storage system." *IRE Trans.* EC-11 (April 1962), 223-238.

The first detailed description of the paging mechanism on the Atlas Computer, including the loop-detecting "learning" algorithm for page replacement.

K2. KIMBLETON, S.; AND C. A. MOORE. "Probabilistic framework for system performance evaluation." In *Proc. Symposium on System Performance Evaluation* [A7], 337-361.

A process is modeled as cycling through the states "blocked," "ready," and "executing." The model is analyzed and shown to predict certain properties of the Michigan Time Sharing System.

K3. KLEINROCK, L. "A continuum of time sharing scheduling algorithms." In *Proc. AFIPS 1970 SJCC*, Vol. 36, AFIPS Press, Montvale, N.J., 453-458.

A piecewise linear priority function is associated with each job, and the scheduler runs the job or jobs of highest priority. A job enters the system with zero priority, it gains priority while waiting according to a "waiting slope," and it gains priority while running according to a "serving slope." A two-dimensional plane displays the possible combinations of the two slopes and the corresponding scheduling algorithms.

L1. LAMPSON, B. W. "Dynamic protection structures." In *Proc. AFIPS 1969 FJCC*, Vol. 35, AFIPS Press, Montvale, N.J., 27-38.

An exposition of the programming implications and power of a system of protection based on the capability idea [D11] with hardware similar to that discussed in [W6].

L2. LAMPSON, B. W. "Protection." In *Proc. 5th Annual Princeton Conf.*, Princeton Univ., March 1971.

A generalized model for protection systems, incorporating an "access matrix" and nonforgeable "domain identifiers," is proposed. Existing protection systems are shown to be special cases. Properties of "correct" protection systems are treated.

L3 LIPTAY, J. S. "Structural aspects of the System/360 model 85: the cache." *IBM Systems J.* 7, 1 (1968), 15-21.

A description of the organization and behavior of the cache store on the IBM 360/85.

M1. MACDOUGALL, M. H. "Computer system simulation: an introduction." *Computing Surveys* 2, 3 (Sept. 1970), 191-209.

The reader is introduced to the construction and operation of event-driven simulation techniques by a running example of a multiprogramming system.

M2. MATTSON, R. L.; J. GECSEI, D. R. SLUTZ, AND I. L. TRAIGER. "Evaluation techniques for storage hierarchies." *IBM Systems J.* 9, 2 (1970), 78-117.

The concept of "stack algorithm" as a model for paging algorithms is introduced and its properties studied. Efficient algorithms are derived for determining the "success function" versus memory size for a given paging algorithm. Priority, optimal, and random-replacement algorithms are shown to be stack algorithms.

M3 MCCARTHY, J.; F. J. CORBATÓ; AND M. MERWIN-DAGGETT. "The linking segment subprogram language and linking loader." *Comm. ACM* 6, 7 (July 1963), 391-395. Also in [R2].

Describes a mechanism for achieving program modularity.

M4. MCKINNEY, J. M. "A survey of analytical time-sharing models." *Computing Surveys* 1, 2 (June 1969), 105-116.

The important queueing theory models for time-sharing scheduling are defined, and the analytic results stated and discussed.

N1 NIELSEN, N. R. "The allocation of computer resources—is pricing the answer?" *Comm. ACM* 13, 8 (Aug. 1970), 467-474.

The philosophy, common conceptions, and common misconceptions of pricing structures for computer systems are discussed.

P1 PARKHILL, D. *The challenge of the computer utility.* Addison-Wesley Publ. Co., Reading, Mass., 1966.

A good description of the issues, philosophy, methods, hopes, and fears that existed during the early 1960s, when enthusiasm for the computer utility ran high.

P2 POOLE, P. C.; AND W. M. WAITE. "Machine independent software." In *Proc. 2nd Symposium on Operating Systems Principles* [A5], 19-24.

Describes the requirements and objectives for realizing transportable software. See also [W1].

R1. RANDELL, B.; AND C. J. KUEHNER. "Dynamic storage allocation systems." *Comm. ACM* 11, 5 (May 1968), 297-306. Also in [A3].

Various automatic storage allocation systems and hardware addressing mechanisms are classified, compared, and defined.

R2 ROSEN, S. (ED.). *Programming systems and languages.* McGraw-Hill, New York, 1967.

A collection of "classic" papers on programming language design, definition, assemblers, compilers, and operating systems to 1966.

R3. ROSEN, S. "Electronic computers: a historical survey." *Computing Surveys* 1, 1 (March 1969), 7-36.

An overview of electronic computing machines through the mid 1960s. The emphasis is on the properties of the various machines and the forces that led to their development.

R4. ROSIN, R. F. "Supervisory and monitor systems." *Computing Surveys* 1, 1 (March 1969), 37-54.

A historical overview of second and third generation operating systems. The emphasis is on IBM approaches.

S1 SALTZER, J. H. "Traffic control in a multiplexed computer system." MIT Project MAC Report MAC-TR-30, 1966.

A description of the operating principles of the MULTICS interprocess communication facility.

S2 SAMMET, J. E *Programming languages*. Prentice-Hall Inc., Englewood Cliffs, N.J., 196[illegible]

Encyclopedic coverage, comparing and contrasting over 100 programming languages.

S3 SAYRE, D. "Is automatic folding of programs efficient enough to displace manual?" *Comm. ACM* 12, 12 (Dec. 1969), 656–660.

Potent empirical evidence supporting the assertion that automatic storage allocation can be at least as cost-effective as manual storage allocation

S4 SCHROEDER, M. D. "Performance of the GE-645 associative memory while MULTICS is in operation." In *Proc. Symposium on System Performance Evaluation* [A7], 227–245.

A description of the associative memory used in the MULTICS segmentation and paging hardware, and empirical data on its performance.

S5. SELWYN, L. L. "Computer resource accounting in a time-sharing environment " In *Proc. AFIPS 1970 SJCC*, Vol. 36, AFIPS Press, Montvale, N J., 119–129

A description of the management, accounting, and pricing systems of CTSS-like systems. Should be read in conjunction with [N1, S7, W8].

S6 SHARPE, W. F *The economics of computers*. Columbia Univ. Press, New York, 1969

This is a self-contained text covering the economic theory peculiar to computers and to system selection and evaluation.

S7. SUTHERLAND, I E. "A futures market in computer time." *Comm ACM* 11, 6 (June 1968), 449–451

Describes a supply-and-demand pricing system for computer utilities. Should be read in conjunction with [N1, S5, W8].

T1. TRIMBLE, G., JR "A time-sharing bibliography " *Computing Reviews* 9, 5 (May 1968), 291–301.

V1. VAN HORN, E C "Computer design for asynchronously reproducible multiprocessing " MIT Project MAC Report MAC-TR-34, 1966.

One of the first doctoral dissertations addressing the problems of determinate computation by parallel processes.

W1. WAITE, W. M. "The mobile programming system. STAGE2." *Comm. ACM* 13, 7 (July 1970), 415–421

Describes a bootstrap sequence technique for implementing transportable software.

W2. Wegner, P. (Ed). *Introduction to systems programming*. Academic Press, New York, 1964.

A collection of some "classic" papers on operating systems principles and design during the early 1960s.

W3 WEGNER, P. *Programming languages, information structures, and machine organization*. McGraw-Hill, New York, 1968.

Discusses the general aspects of third generation machine and software organization; assemblers and macro-generators; ALGOL runtime environment implementation; PL/I and simulation languages; and Lambda Calculus machines.

W4. WILKES, M. V. "Slave memories and dynamic storage allocation " *IEEE Trans. Computers* C-14 (1965), 270–271

Proposes an automatic multilevel memory system, the predecessor of the "cache store" [L3].

W5. WILKES, M. V. "Computers then and now." *J. ACM* 15, 1 (Jan. 1968), 1–7.

A thought-provoking piece containing historical perspective and evaluation of trends, written as a Turing Lecture.

W6. WILKES, M. V. *Time sharing computer systems*. American Elsevier Publ. Co., New York, 1968

A masterful 96-page overview of the properties and problems of CTSS-like time-sharing systems.

W7 WILKES, M. V "A model for core space allocation in a time sharing system." In *Proc. AFIPS 1969 SJCC*, Vol. 34, AFIPS Press, Montvale, N.J., 265–271.

Uses experience with time-sharing systems to develop a minimal relationship between processor and core allocation policies.

W8 WILKES, M. V.; AND D F HARTLEY. "The management system—a new species of software?" *Datamation* 15, 9 (Sept. 1969), 73–75

Outlines the motivation and requirements for a management system as illustrated by the Cambridge University Multiple Access System. Should be read in conjunction with [N1, S5, S7].

W9 WILKES, M V "Automatic load adjustment in time-sharing systems " In *Proc. Symposium on System Performance Evaluation* [A7], 308–320

A model of the load-leveling mechanism used on CTSS is presented and analyzed using some techniques from control theory The emphasis is on preventing instability in the system whose load is being controlled

W10 WULF, W. A. "Performance monitors for multiprogramming systems " In *Proc 2nd Symposium on Operating Systems Principles* [A5], 175–181.

A proposal for a general method of performance monitoring and control of resource usage rates.

BIBLIOTECA DE
C.I.M.A.S.

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 5, 5-18 (1975)

DSOS—A Skeletal, Real-Time, Minicomputer Operating System*

DENNIS J. FRAILEY
Southern Methodist University, CS/OR Department, Dallas, Texas 75275, U.S.A.

SUMMARY

This paper describes a skeletal operating system which provides the services and structure necessary to implement real-time applications. Features which would require a specific configuration or a non-minimal computer are either omitted or optional. In particular, although some protocol is established, no peripheral devices are required. System features include dynamic storage allocation, memory protection, multiprogramming, pseudo-clocks, and co-ordination and communication primitives. Due to its simple design, the system 'kernel' has proven to be remarkably error free.

KEY WORDS Real-time Operating systems Minicomputers

PREFACE

Although traditionally viewed as different from each other, real-time systems and general purpose operating systems have grown more alike in recent years. Capabilities such as multiprogramming, dynamic resource allocation and priority scheduling are essential to real-time systems, although this terminology is not always used. Design principles based on the work of Spooner,[1] Dijkstra[2] and others whose efforts were primarily motivated by the problems encountered in large scale, general purpose multiprogramming systems, have obvious application in real-time environments.

The DSOS system illustrates much of this. Its features and capabilities, though perhaps interesting, are by no means unusual and, in retrospect, several weaknesses and limitations are apparent. The simplicity of design resulting from the influences mentioned, however, has produced systems in which software reliability is a fact rather than a goal to which only lip service is paid. A simple, straightforward 'kernel' has been produced and, in effect, thoroughly debugged. Systems built around this kernel can rely on its integrity and programmers can concentrate on the specific problems at hand rather than who was at fault for the most recent system crash.

INTRODUCTION

Development of a real-time application typically occurs in one of two ways: the specific 'from the ground up' approach, or the 'general purpose real-time system'. The former is often used when time and memory are tightly constrained, and the result is typically a workable but error-prone system which is difficult to debug. Documentation is often limited due to time constraints and the dynamic nature of system development. Serious bugs are often detected years after such systems have been declared operational, and the only reason

* This work was carried out while the author was a consultant to Texas Instruments, Inc., Services Group.

Received 19 February 1974

© 1975 by John Wiley & Sons, Ltd.

they succeed at all may well be that the magnitude of the application is sufficiently small to allow one or two 'expert' programmers to know the system thoroughly.

One of the most unfortunate results of this approach is that good software ideas tend to be hidden, buried or 'lost in the shuffle'. Management is traditionally secretive with any development which can be described as an improvement, especially in the presence of competition or sales potential. In the few cases where detailed descriptions are published the readership is likely to be limited to those interested in the application. Years later when similar ideas are published in a software design environment the practitioners will scoff that 'the academics are only now discovering something we published years ago' (probably in an article on real-time control of widget grinding).

The 'general purpose real-time system' is often developed when a series of related applications are anticipated on a given computer; similar hardware configurations are also usually expected. Use of one system for several applications enables the programming staff to concentrate on the application rather than the details and idiosyncracies of the hardware. Drawbacks of this approach include a tendency to consume large amounts of memory and other overhead for capabilities which are more powerful than is really necessary; configuration dependence which creeps in along with the 'standard' features often required to make a sale; and reluctance to use the system because of the 'not invented here' syndrome. However ideas developed in this context tend to spread more readily, since publication, if it occurs, will probably be in a less specialized journal. Furthermore, at least potentially, over-all documentation is more thorough and system organization is more straightforward since a variety of end uses must be anticipated.

The DSOS system was born in an environment similar to that described above, namely anticipation of a series of related real-time applications on a given computer; however time and memory constraints as well as uncertainties about the ultimate configuration details led to an unpretentious design in which only the kernel of the system and its over-all organization are 'cast in concrete'. The configuration details, the logic of the real-time application, the sizes of numerous tables and various other system parameters are specified for each application, with a considerable amount of freedom in the approach taken. Certain constraints are, of course, imposed on the design of the application system; the difference from larger 'general purpose' systems is more a matter of degree. While DSOS-based systems have had several disadvantages characteristic of the 'general purpose' approach, such as requiring more storage than originally expected, these effects have been only moderate whereas benefits have been substantial: the software systems have been remarkably error-free, even in their initial releases (see concluding remarks); extension of one system to several new applications has been achieved without modifying the 'kernel'; and the discipline (or, perhaps, bureaucracy) imposed by the design has aided documentation and program simplicity.

To summarize, DSOS is not an operating system in the traditional sense, for it provides no direct capabilities for job management, I/O support or operator interaction. It forms, rather, a basis on which a variety of systems could be developed. It would appeal particularly to the OEM environment where specialized software must be developed for minimum configurations and non-standard peripheral devices.

DEFINITIONS AND TERMINOLOGY

We provide a set of working definitions and terms. In several cases where other terms have been used in the literature to describe similar concepts, examples of this alternate terminology are included in parentheses.

The reader should understand that, as viewed here, an operating system controls tasks, not programs. In particular, when a program is re-entrant, it may be involved in numerous, concurrent tasks of varying priorities. Thus it is convenient to view programs as resources used by tasks. From this viewpoint a re-entrant program is a shareable resource, much like a read-only data base. An operating system, then, simply allocates resources and schedules tasks.

A *procedure* (routine, program) is defined to be any sequence of code which performs some function. A *task* (activity, process) is an *invocation* (activation, call) of a procedure, achieved by combining that procedure with an *invocation record* (working storage) consisting of a set of *arguments* (parameters) passed from the calling task as well as several other quantities describing the task. The invocation record is defined to include all data which may be read and/or written by the procedure during execution of the task.

If a procedure is invoked with identical arguments from two separate tasks, the resulting tasks must be uniquely identifiable; thus the identity of the calling task is a part of the invocation record. Another implied member is the real-time at which the task was invoked, since this may be needed to distinguish similar calls from the same parent task. Finally, the contents of the high speed registers in the CPU are viewed as part of the invocation record, a part which dynamically changes to reflect the state of the task as its execution proceeds.

A procedure whose output (cells into which it writes) consists solely of the computer's high speed registers and arguments passed by the calling routine (i.e. a procedure which does not modify its own code or internal data) may be used concurrently as part of several tasks. Such a procedure is called *re-entrant.* All DSOS routines are classified as re-entrant or non-re-entrant and the former may be shared by any number of tasks. Non-re-entrant routines are usually at least *serially reusable* (i.e. can be used again, once run to completion). This is relatively standard in a real-time environment. In fact most routines can also be used again if terminated abnormally (as in a system restart without reloading). The latter property is called *reusable.* Generally speaking, all re-entrant programs are reusable and all reusable programs are serially reusable. Sometimes, although its individual routines are not all reusable, the system as a whole has this property. DSOS, in particular, is reusable without reloading except for the system checksum which is computed after the initial load and checked on any restart. Any application system using DSOS may also be made reusable by providing initialization entries for routines which save local variables between calls.

One important subclass of the non-re-entrant routines which is a traditional problem in real-time environments is the non-re-entrant procedure which may be called upon by two or more concurrent tasks. Such procedures must not be allowed to handle a call from a second task until the first call is complete. We refer to these as *NR* procedures to distinguish them from those which, due to the logic of the application, cannot be called concurrently. In DSOS, calls to NR procedures are queued (first come first served or priority) to assure that concurrent use will not occur. In effect, they must be treated as non-shareable resources.

A task has *read access* to a section of memory if it may read the contents of that section. Similarly, *write access* implies the ability to modify the contents of memory and *transfer access* the ability to branch (transfer control, jump) to code located there or to initiate a task whose procedure is located there.

HARDWARE CHARACTERISTICS

DSOS is implemented on a TI 980-A 16-bit minicomputer with an I/O bus allowing up to sixty-four devices, direct memory access for up to eight devices and several other interesting

features. The only capabilities which are important to the over-all design of DSOS are a *base register* (B) which allows access to a segment of memory remote from the accessing code, two *memory bounds registers* which allow the upper- and lower-most portions of memory to be protected and a *status register* which allows the prohibition of I/O instructions, bounds register settings and other dangerous activities. In short, it is possible to have a protected operating system, to divide a program into two segments and to forbid certain activities in inappropriate places.

A notable omission from the standard hardware is a *real-time clock*, although an interval timer is available. In most real-time applications some kind of clocking device is needed to generate interrupts on a regular, periodic basis. DSOS is designed to run independently of any such device; but it provides support for one if it exists (the length of the clock period being irrelevant to DSOS but very important to the given application).

There are four standard interrupts on the 980-A and each causes control to transfer to a separate address. An optional feature is a *priority* or *vectored interrupt* capability with which each peripheral device may be assigned to a separate interrupt address in memory, with an associated hierarchy of interruptability. Again, since this feature is non-standard, DSOS is designed to operate with or without it. When the feature is used, a block of memory is relegated (by hardware) to support linkage. When the feature is not available DSOS uses this same block of memory for a table which links devices to device support software. Standard interrupt handlers determine which device interrupted and transfer control to appropriate software through this table. The format of the table is identical to that required by the priority interrupt hardware so that little change is needed when switching between configurations with and without the feature.

OVERVIEW

DSOS is a simple multiprogramming system design to support real-time applications. Standard features include:

(1) Several priority levels (the number is arbitrary but must be specified at system assembly time).

(2) Memory protection.

(3) Concurrent execution of tasks, where appropriate.

(4) Dynamic storage allocation for data areas, buffers, etc.

(5) Configuration independence.

(6) Support for re-entrant code.

Optional features (depending on configuration) include:

(a) Real-time clock support.

(b) Support for certain peripheral devices (teleprinter, paper tape, etc.).

Since DSOS is designed to run on any configuration, including one with no peripherals, it neither requires nor supports a mass storage device, and is fully contained in memory with no overlays, swapping, etc. If the latter are needed they are viewed as part of the real-time application and must be supported therein.

The minimum configuration required for DSOS is a TI 980-A mainframe with a 4K (K = 1,024 16-bit words) memory. With the minimum configuration, about 1K words are available for the application program(s); in a typical application, a configuration of 8K or more would be suitable.

DSOS is designed to simplify the development of real-time applications by providing a framework and a set of services which are needed in such an environment, including task scheduling and co-ordination, communication between tasks, memory management and others. Most of these services are supported by system requests which may be invoked by assembly language macros, so the effort required to utilize them is minimal.

An application using DSOS will appear in memory as a series of 'shells' corresponding to priority and memory-protection levels (Figure 1). Each shell has two parts known as the *data block* and *code block*. Shell 0, which consists of the two outer portions of memory, is occupied by a series of peripheral device handlers and interrupt processors. For the most part these routines are specific to a given application, although several routines are available for standard devices and configurations. Routines in shell 0 command highest priority and have read, write and transfer access to all of memory.

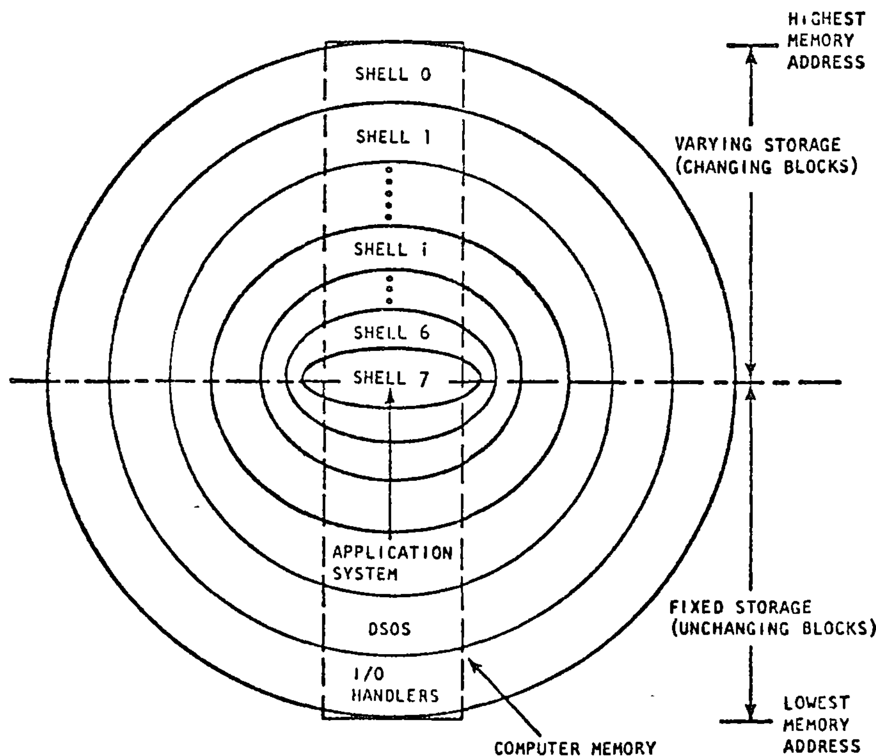


Figure 1. Shell structure (physical layout, memory access)

Shell 1 contains the 'kernel' of DSOS and shells 2 and higher contain application routines and certain optional DSOS routines. Shell 2 will have highest priority within this group, with shells 3 and higher having successively lower priorities. This enables a protected hierarchy of environments within the application, similar to those found in a number of general purpose operating systems.[2,3] For those familiar with Dijkstra's work it should be pointed out that Figure 1 represents a *physical* layout which is 'inside out' from the 'layer' picture usually associated with the *logical* structure of his systems. In fact, the logical structure of DSOS resembles that recommended by Spooner[1] (Figure 2).

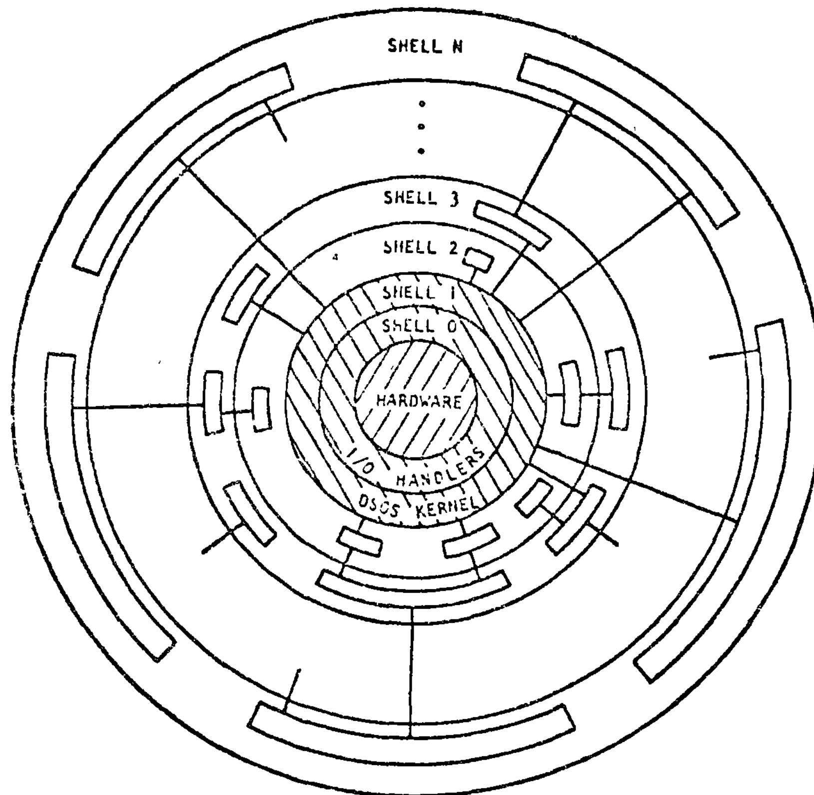


Figure 2. Shell structure (logical organization)

In general, routines at shell N have read, write and transfer access to all portions of memory allocated to shells N, N+1, etc., but have only transfer access to lower-numbered shells. Furthermore, transfer access to a lower-numbered shell is limited to its code block. Routines at shell 2 or higher have *no* access to shells 0 and 1 except through strictly controlled operating system requests. This provides an extra level of protection for the kernel of DSOS and the interrupt routines.

By convention all routines in DSOS consist of an unchanging block and a changing block. *All* code and some unchanging data (I/O format descriptions, etc.) are kept in the former, with changing data kept in the latter. These two blocks are stored in separate areas of storage, one in each part of the shell, and each program's base (B) register contains a pointer to its data block. If this convention is followed in the application programs as well, the net effect is that a program at shell *n* has write access only to changing blocks in shells *n* or higher. Some additional advantages of this convention are as follows:

(1) Since all unchanging blocks are contiguous in storage, a checksum for system integrity is easily implemented.

(2) When memory dumps are required, only the changing blocks need be dumped, thus reducing the volume of output.

(3) Such dangerous techniques as self-modifying code are prohibited.

(4) Re-entrant code is conveniently implemented.

Due to the instruction set of the TI 980-A, certain I/O and operating-system instructions must be modified during system operation; thus, in shell 0 *only*, the code block may violate the conventions just mentioned.

One substantial limitation is that shell boundaries are static once the system has been loaded. This restriction, though not inherent in the structure, was made to simplify implementation.

TYPICAL OPERATION

After a DSOS-based application is loaded and powered up, it will have initiated a series of tasks at various shells. Once initiated a task may be in *waiting* or in *active* status. In the latter (active) case it is either *running* or *idle*. A waiting task is one which cannot progress further until some event occurs (such as completion of another task, the setting of a flag by an interrupt processor, etc.) and which is scheduled to be activated when that event occurs. A running task is one to which the processor is currently assigned. An idle task is one which is waiting to use the processor.

The DSOS scheduler processes all active tasks at shell 0, then all at shell 1, etc., until no more remain. If, while a task is running, a task at a lower shell (higher priority) is activated, the current task is interrupted until the lower-shell task either completes or changes to waiting status. As a result of this, tasks at low priority levels (high shell numbers) are serviced only occasionally when the system is busy. Tasks are initiated and activated by other tasks and by hardware interrupt handlers. A task may terminate (complete) by returning to the operating system or to the task which initiated it.

A task may 'call' (in the sense of a subroutine call) for other tasks at any shell. A task may also initiate other tasks in an independent manner; i.e. task A may initiate task B which, conceptually and perhaps literally, will proceed in parallel with task A. A good example is a background (low-priority) task which initiates a high-priority task to type out a message. Due to the printer's relatively low speed, the high-priority task will frequently be waiting for a printer interrupt (i.e. in 'waiting' status), during which time the background task may run. Whenever the printer requires service, the high-priority task will be activated (briefly), interrupting the background task.

SPECIFIC SYSTEM FEATURES

Dynamic storage allocation

The last portion of the data block of each shell is a set of free storage which may be dynamically allocated to tasks in that shell. The method of allocation is the 'buddy' system in which all allocated blocks are of size 2^m for some m. If a block of size 2^m is not available, a block of size 2^{m+1} can be split in two or two 'buddy' blocks of size 2^{m-1} combined. This simplifies storage recovery while allowing flexibility in block size. Additional information on this system can be found in Knuth's First Volume[4] although the method used in DSOS is somewhat different in that returned blocks are not collected into larger blocks unless (and until) necessary.

The latter approach is taken because, in the kind of system envisioned, a state of 'equilibrium' is expected in which a fairly steady number of smaller sized blocks are needed. If enough memory is available, this state can exist with little overhead from block splitting and collecting. Furthermore, by listing the free block chain after an extended period of system operation, it is possible to ascertain that space was unused and thus 'tune' the system by

assigning this space elsewhere. It can be argued that a thorough analysis of the programs would provide more accurate information on block usage, but with the continuous changes and informal definitions involved in a typical application, such analysis would be difficult to achieve.

A separate free space area is provided for each shell, and each may be of any desirable size (specified at assembly time). Normally, blocks will be allocated to a routine at shell j from the free space area of shell j; however, any task may also request that a block be allocated from the highest-numbered shell, shell N. Such blocks are called *universal blocks* since they may be accessed from any shell. A drawback due to fixed shell boundaries is that if one shell runs out of space it is impossible to automatically borrow space from another.

Software interrupts

Among the essential components of a real-time system are mechanisms for communicating between tasks, resolving mutual interference problems and scheduling tasks on the basis of various events. A common solution to all of these is a device for sending (posting) and/or awaiting a signal and/or message.[5] To augment this, some systems provide devices to send or await responses to messages or signals.[6] Protection is provided by requiring messages to be accompanied by an identification of the sender and perhaps also by severely constraining message formats.

In view of the skeletal nature of DSOS the mechanisms provided for these purposes are rather simple and non-constraining, with protection mechanisms superimposed, if desired, by the application system. A parallel with hardware interrupts is used both for simplicity and to foster hardware independence. A set of flags called *software interrupts* are maintained (the total number is arbitrary) and these may be accessed by any task through primitives of the kernel. Capabilities provided include: initiate a task when a flag is set; wait (suspend task) until a flag is set; and set, clear, test and set, test and clear, or test and invert a flag. Event driven scheduling is achieved by associating flags with events. Flags may also correspond to signals, and messages are handled by associating fixed message locations or blocks of storage from free space. Mutual exclusion and task co-ordination problems are handled in straightforward ways, though there are more elegant solutions in the literature.[2, 7, 8] Certain problems which are indigenous to real-time systems can also be handled by the uninterrupted copy and pseudo-clock features whose descriptions follow.

Uninterrupted copy

It is often the case in a real-time environment that several routines share a block of data and, in particular, that one or more of the routines will periodically modify the data. Perhaps the block contains the current value of a message which is frequently updated. Rather than queueing copies of the message it is important that only the most recent copy be used. It is also usually important that no reads be initiated or interrupted during an update because the resulting read would be inconsistent. Sorenson[9] has discussed this type of problem and illustrated why conventional solutions may be unacceptable in a real-time environment. His solutions, however, are more elaborate than that of DSOS.

DSOS allows any task to request an *uninterrupted copy* of up to sixty-four words of data (this limit is arbitrary and may easily be changed, but would endanger critical timing deadlines if allowed to become too large). If all reading (copy out) and updating (copy into) of a particular message block are performed using this feature, the desired conditions will be met. Since all access occurs in priority order, blocking for long periods is not a problem for high priority tasks.

Real-time clock support

Any device which generates interrupts on a regular, periodic basis may be designated as the system real-time clock. When such a clock is available DSOS provides mechanisms by which any task may request the creation of a *pseudo-clock*, a counter which is updated on each cycle of the real-time clock. Pseudo clocks may be classified as follows:

(1) A *timer* runs only when its requesting task is running, whereas a *real-time* clock always runs unless specifically stopped by its task.

(2) An *incrementing* clock is incremented by 1 on each cycle. A *decrementing* clock is decremented by 1.

Thus there are four possible types of pseudo clocks: incrementing timer, decrementing timer, incrementing real-time clock and decrementing real-time clock.

Numerous clock manipulation and utilization functions are available. Capabilities provided include:

(a) Generate a software interrupt when a clock reaches a certain value.

(b) Stop or start a real-time clock.

(c) Read, write or add to the value of a clock.

In conjunction with the system's capabilities for scheduling tasks when software interrupts occur, clocks enable a wide variety of timing relationships to be arranged. Some examples are given later in this paper. The power of these simple features has led to a more thorough analysis of the types of timing mechanisms used in system implementation languages, and some of the results thereof are discussed in another paper.[10]

Peripheral devices and hardware interrupts

Processing of hardware interrupts and peripheral devices is highly configuration- and application-dependent; thus few routines are provided. Several conventions are required and certain support vehicles are available. Optional routines for frequently used devices are supplied with DSOS and these may serve as models for similar devices in a particular application.

Each peripheral device, whether capable of generating an interrupt or not, is treated as though it does so. Each device is assigned (at assembly time) to one of the sixty-four priority interrupt locations *regardless of whether priority interrupts are available.* This location is used thereafter to refer to the device or its handler, and to determine its servicing priority.

Each device is assigned to a priority interrupt location corresponding to its servicing priority. If priority interrupts are not available or are not used by a particular device, the direct memory access or I/O bus interrupt routine 'simulates' the actions of a priority interrupt. This feature enables a relatively straightforward upgrade to a system with priority interrupts. Such simulation is not necessary in the case of a device which does not use interrupts, but the assignment of a priority interrupt location is still made to assure consistency in device/handler referencing and to allocate a servicing priority.

A software vehicle permits the assignment of handlers to specific devices (or, rather, to specific priority interrupt locations). It is possible to dynamically change handlers for a specific device or to prevent such changing for system protection.

Peripheral device handlers, which must be located in shell 0, may be called by system tasks or via hardware interrupts. Their function is to perform direct communication between devices and the application system. All functions which do not involve direct communication with a device, such as buffering, code conversion or error checking, are performed by routines

at shell 2 or higher. This tends to minimize the occurrence of system crashes due to errors in I/O handling since only the 'kernel' of the device handling function is allowed to run at a priority/protection level which would permit overwriting of DSOS.

Subroutine calls and task initiation

DSOS supports a variety of subroutine and task invocation functions. One may call a subroutine at any shell (except 0 and 1) and, if the shell is higher than the calling shell, may assign it either priority. One can initiate a task which is executed (conceptually and perhaps literally) in parallel with the calling task, and these two tasks may retain communication or become completely independent.

Other capabilities include testing for completion of previously initiated tasks, queueing tasks in a specific sequence and scheduling tasks to be initiated on the occurrence of hardware or software interrupts. Any task may make itself idle temporarily, allowing other tasks at the same priority to proceed, or may suspend itself until a specific software interrupt occurs.

A convenient feature of the DSOS architecture is that calling tasks need not know the priority of subroutines or other tasks which they initiate. Only the name of the task and an argument list need be specified. Similarly, a task need not know the identity or priority of its initiator. Furthermore, a task may be called as a subroutine (calling routine awaits its completion) or initiated as a parallel task in various ways. In other words, a program need not know how it will be used.

SYSTEM ORGANIZATION

Over-all structure

There are two types of system components. *Kernel* components are those required in all DSOS systems. *Optional* components are those useful in some cases but not required. A 'minimal' system would consist of the kernel and an application program. A typical system would also contain I/O routines and selected optional components.

Certain of the kernel and optional components must be tailored at assembly time for each specific system. For example, the table of NR routines must contain one entry for each such routine (the 'minimal' version has an empty table). By and large, all tailoring is done in data areas, with only an occasional situation where reassembly of a code block is necessary. Among the other tailoring possibilities are: number of shells; size of free space (for block allocation) in each shell; number of distinct software interrupts.

Among the optional components are the real-time clock support routines; I/O support for a teletypewriter; a FORTRAN-like I/O format scanner (re-entrant and device independent); re-entrant libraries of arithmetic functions (double length integer and simulated floating point); error message accumulation and printing routines; checksum routines; a memory dump/write routine used from the operator's teletypewriter; support routines for the standard I/O bus and direct memory access interrupts which 'simulate' priority interrupts.

The kernel components can be subdivided into fixed data areas, dynamically allocated data areas and system routines.

Fixed data areas

Corresponding to each shell is a *control point* indicating which tasks are currently active or waiting in that shell, the memory boundaries of the shell and other such information. In addition, each shell has a block of *free space* and a set of pointers and flags pertaining to the current status of free space.

For software interrupts there are a *flag table* (1 bit/flag) and a *task table* (1 word/flag) indicating the tasks to be initiated when interrupts are generated. The priority interrupt locations serve as the *interrupt table*, relating peripheral devices to device/hardware interrupt handlers.

Finally, there is the *non-re-entrant table* containing entry points for NR routines (non-re-entrant routines which might have concurrency troubles if called while already active). Each entry in the table contains the address of the routine, a pointer to the TDB (see below) of the task currently using the routine, a pointer to a list of waiting tasks and assorted flags.

Dynamically allocated data areas

Each active or waiting task has a *task descriptor block* (TDB) which describes the current status of that task. TDB's are chained to the control points of the corresponding shells; i.e. control point 3 points to a chain of TDB's describing tasks at shell 3. Actually, there are two chains per control point—one each for active and waiting tasks.

A TDB may contain the high speed register file and status flags for an idle or waiting task, a pointer to the TDB of a task which is waiting for the first task to complete, a pointer to a chain of EDB's (see next paragraph) and other such information. A task is identified by its TDB, thus a 'task pointer' is a pointer to the TDB of that task.

Whenever a block of storage, a clock or other resource is assigned to a task, an *entity descriptor block* (EDB) is created and added to the EDB chain of the task's TDB. This enables an arbitrary number of such resource entities to be assigned to a task and provides for their automatic recovery when the task terminates.

The EDB's and TDB's are stored in blocks obtained dynamically from the shell 1 data area. If this data area should overflow, the system will refuse the requested function or, in some cases, abort. The space limit, of course, applies only to the total number of blocks, regardless of type. Specification of the limit must be made for each application and is a function of the maximum amount of expected activity.

System routines

The *block allocator* is a re-entrant routine which allocates blocks from the data area passed to it as an argument. The *scheduler* causes tasks to be activated when software interrupts occur, determines the proper task to be run next and dispatches that task. The *internal interrupt processor* is called by all system requests (which are initiated by 'out of bounds' branch instructions) as well as by certain hardware-detected errors. Its function is to determine the desired system request (or the nature of the error) and route the request appropriately. In most cases the request is processed by this routine, but occasionally it calls upon the block allocator or other service routines. The internal interrupt processor always exits to the scheduler.

The only other routines in the kernel are assorted service routines and a *memory dump* routine which actually resides outside of the system and is loaded only when enough memory is available.

Application system organization

An application system will have an initial 'power up' task whose function is to initiate everything else. This routine is entered when a 'system reset' is initiated by the operator (DSOS resets itself and then initiates the 'power up' task).

The remainder of the application system may take any form desired. Typically it will resemble an operating system with high priority functions and control routines at lower shells

and background tasks at higher ones. Certain functions such as pseudo clocks and dynamic allocation of *data* space would be handled directly by DSOS. Dynamic allocation of *program* space, if desired, would be handled by the application system and would probably be limited to higher numbered shells so as to permit a maximal amount of checksumming.

EXAMPLES

I/O diagnostic system

This example is a mythical composite of several existing systems. Its function is to test out one or more of a set of I/O devices. Its components include the kernel, the teletypewriter and real-time clock routines, the format scanner, I/O support routines for all devices and diagnostic routines for each device. This system would normally run in 8K and could be modified to run in 4K if the number of devices were limited.

The first task of the application waits for the operator to key in certain flags via sense switches on the control panel. No I/O interrupts are allowed and at this point no I/O devices need be connected to the computer. The second step, initiated by a switch setting, is to thoroughly test out the teletypewriter (without allowing interrupts). If a real-time clock is available, it is also thoroughly tested. Finally, the 'main loop' is entered.

The purpose of the main loop is to control all other testing. It runs at shell 2 and thus is protected from and has higher priority than the diagnostic routines. It first connects an 'error' handler to each possible I/O device, so that any interrupt can be caught and the operator informed. Next, the operator types in a set of commands indicating which devices are to be tested and how they are connected to the hardware (which device number, etc.). As entries are made, support routines are 'connected' (by software) to specific devices.

Devices may be tested singly or in parallel, with or without interrupts. Once a group of devices has been specified, the operator types a command which allows interrupts (if specified) and causes the requested device diagnostic routines to be started. Diagnostic routines are written in such a way as to frequently idle or deactivate themselves, allowing others to proceed in parallel. (Usually the diagnostic routines wait for hardware or software interrupts.) The routines will be located in shells 3 and higher.

If a real-time clock is available, the main loop may incorporate time limits on the activation of diagnostic routines. If no clock is available it must rely on operator action or the diagnostic routines themselves to give it control.

An operator command on the teletypewriter can cause the main loop to become active (though not necessarily stopping the diagnostic tests). When an 'end of test' is called for, the main loop prevents interrupts, 'disconnects' all I/O handlers, restores the 'error' handlers to all devices and waits for commands pertaining to the next test. At this time new devices may be safely connected to the computer.

An optional feature of the diagnostic system is to continuously run a background, low priority checksum of all code blocks.

Navigation systems

This series of systems is the primary application for which DSOS was designed. The systems control three interrelated functions: navigation of a boat based on position data obtained from several devices; taking geophysical measurements of the terrain underneath the boat; and displaying and recording these data on assorted devices. A certain amount of processing is done on some of the data before recording, and other background functions

may be performed from time to time. The GEONAV* system, a typical member of the series, uses most of the optional components of DSOS and has about twenty distinct interrupting devices to attend to, as well as several which do not generate interrupts. The system requires 32K of memory, divided into eight shells (0–7), and required about 2 man years of effort for software development.

The POWERUP routine (shell 2), which is entered after DSOS is started up, connects all devices to their handlers and then issues a 'lock' command by which future device/handler connections are prohibited. It then proceeds through a dialogue with the operator to determine initial data values and set up various system tasks. The last step is to ask for the correct time. When the time is provided the system allows interrupts and exits to a low priority loop (shell 7) which checks settings of various switches and handles other background tasks such as calling for periodic checksums of the code block.

The moderate priority ONESEC routine (shell 4) uses a pseudo-clock to activate itself once per second. Its main function is to adjust the current estimate of the boat's position on the basis of data read during the past second. Other functions include incrementing the internal clock by 1 second (for recording purposes), and updating numerous display devices.

Most input devices have associated routines in shells 2, 3, 5 or 6 which process their data and, after smoothing, averaging, etc., pass data on to be used by the navigation and recording routines. These devices generate interrupts when data are ready and the associated routines are activated when blocks of data have been accumulated.

The re-entrant mathematical routines are stored in shell 7 so that they may be called from anywhere and run at whatever priority desired. I/O functions of the system utilize the dynamic storage allocation of DSOS for buffers. Over fifty software interrupts are used in numerous ways, such as for synchronization between tasks sharing common data. The uninterrupted copy feature is frequently used to avoid concurrency problems with key blocks of positioning information.

FINAL REMARKS

DSOS has been used successfully in several application systems. Of particular value have been the device independence and modular organization. The systems have easily incorporated changes in devices, device characteristics and associated software, as well as substantial variations in over-all architecture. On the GEONAV system, the grouping together of unchanging blocks reduced memory dump time from 12 minutes (code and data) to about 3 minutes (data only), a welcome saving on the sometimes heavily used test machines. Initial design and implementation of DSOS (including time to learn the 980-A architecture and instruction set) was begun in November 1972 and required about 6 months of part-time effort. Three bugs were discovered in the 'kernel' within the next few months, two of these being found during final documentation efforts. None have since been found.

The most serious problem has been the difficulty of training programmers, schooled in traditional concepts of programming, to the task and resource orientation which is fundamental to the design. In particular, the abstraction from 'program' to 'task' has been difficult to get across.

A few of the features described here, notably timers, were never fully implemented for lack of necessity in the systems under construction. Others, including some of the more powerful task initiation capabilities (e.g. create a task and notify creating task when it has terminated), have seldom been used and, in view of the space required, are of questionable

* Trademark of Texas Instruments, Inc.

worth. Certain task scheduling functions were not needed because software interrupts could achieve the desired results with more flexibility and little additional effort. In fact software interrupts are probably the most heavily used feature of the kernel. Pseudo-clocks, though usually few in number, tend to be important to the systems in which they are used.

ACKNOWLEDGEMENTS

The author is indebted to Russ Keenan, Jim Kalan, Walter Hicks, Mike O'Hagan and Bill Nylin, who programmed and/or helped design parts of DSOS.

REFERENCES

1. C. R. Spooner, 'A software architecture for the 70's: part 1—the general approach', *Software—Practice and Experience*, 1, 5–37 (1971).
2. E. W. Dijkstra, 'The structure of "THE"—multiprogramming system', *Comm. ACM*, 11, 341–346 (1968).
3. E. W. Dijkstra, 'Solution of a problem in concurrent programming control', *Comm. ACM*, 8, 569 (1965).
4. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison–Wesley Reading, Mass., 1968.
5. *Specifications for the Texas Instruments Multiaccess Executive (TIME) for the 980 A*, Texas Instruments Incorporated, Digital Systems Division, Houston, Texas, TI-514-5C, 1973.
6. P. B. Hansen, 'The nucleus of a multiprogramming system', *Comm. ACM*, 13, 238–241, 250 (1970).
7. F. J. Corbato and J. H. Saltzer, 'Multics—the first seven years', *AFIPS—Conf. Proc.* 40, 571–583 (1972).
8. D. E. Knuth, 'Additional comments on a problem in concurrent programming control', *Comm. ACM*, 9, 321–322 (1966).
9. P. G. Sorenson, 'Interprocess communication in real-time systems', *Operating Systems Review* (ACM SIGOPS), 7, No. 4, 1–7 (1973).
10. D. J. Frailey, 'Timing features for systems implementation languages', *IFIP 74—Conf. Proc.*, 354–358 (1974).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 5, 147-167 (1975)

The Design of a Real-Time Operating System for a Minicomputer. Part 1

BIBLIOTECA DE
C.I.M.A.S.

W. F. C. PURSER AND D. M. JENNINGS
System Dynamics Ltd., Dublin, Eire

SUMMARY

The design objectives of real-time operating systems for minicomputers are considered. Simplicity, efficiency and avoidance of superfluities are the main criteria. A typical multi-tasking structure is sketched and the executive elements to support it are analysed. Examples of their use in simple Input/Output are given. Filing systems are then discussed, followed by the interface with the operator. Finally, the problem of linking and loading the modules which make up a simple core-resident real-time system is reviewed.

KEY WORDS Multi-tasking Virtual instructions Interruptability Executive

INTRODUCTION

Minicomputers are used extensively in on-line systems, controlling electronic or mechanical equipment and maintaining communication with one or more operators in parallel. To do this, some sort of real-time operating system is required, which may be a one-off 'lash-up' written by the user, or a general-purpose manufacturer's package. However, as minicomputer systems become more sophisticated, and particularly since the universal introduction of the magnetic disk (fixed-head, cartridge or disk-pack), so 'lash-up' operating systems become more complicated and unsatisfactory. Instead, most manufacturers* now provide a real-time operating system themselves, and many large-scale users (e.g. software houses)[1] will develop their own standard system if they find the manufacturer's package either inadequate or too general to be efficient.

A real-time operating system for a minicomputer differs from that for a large system mainly in approach. The minicomputer user is nearly always concerned with efficiency in speed, core and disk usage. He wants no superfluous facilities, and the system is built from the bottom up. It is frequently aimed at a dedicated application. It is the purpose of this paper to discuss how such a system is constructed. This should enable users to evaluate manufacturers' systems, to appreciate what facilities can be provided and with what implications, or to write their own system if they feel it necessary.

* Examples of manufacturers' real-time operating systems for minicomputers are:
Digital Equipment Corporation (PDP-11) RSX-11A to D. M.
Interdata (Models 70, 80) RTOS.
Data General (Nova, Supernova) RTOS, RDOS.
Honeywell (Series 16) RTX-16.
Philips (P-855, P-860) Basic Realtime Monitor.
Hewlett-Packard (2100) RTE.
Computer Technology Limited (Modular One) E4.
Digico (Micro 16V) RTOS-16.

Received 15 July 1973

© 1975 by John Wiley & Sons, Ltd.

The paper is divided into two parts. Part 1 discusses system structure, the constituent parts of the executive, basic input/output handling, the filing and disk system, communication with the operator and the important topics of configuring, linking and loading. Part 1 therefore, describes a simple core-resident operating system, with a filing system, such as is used in many small dedicated real-time applications. Part 2 (to be published later) is concerned with additional features which may be required, including store protection and capabilities, overlay and dynamic store allocation, high level language and background programs. These facilities are associated with larger systems and continuing program development.

SYSTEM STRUCTURE

The basic requirement of all real-time systems is multi-programming, in which many processes (following Dijkstra[2]) conceptually operate in parallel. In practice, control is swapped between processes in response to external demands such as interrupts, internal activations such as buffer-passing and forced rescheduling caused by semaphore operations or, in some systems, by the clock ('time-slicing'). A 'process' is equivalent to the concept of a 'task' or an 'activity' (Barron[3]). 'Task' is used by many manufacturers.

Processes and PCBs

In broad outline, the complete program consists of an executive and a number of processes. Some processes, notably those concerned with file-handling and Input–Output, together with the executive form the real-time operating system; while other processes are applications dependent. Each process is allocated a process control block (PCB) or its equivalent. A PCB is a data area holding all information about that process which the executive needs to know. For example, the PCB contains the values (links) to be set in all programmable registers when the process is activated. It also contains active/suspended and priority information about the process. When the code corresponding to a process is inserted into the system a PCB must be created for it. This is done when initially configuring the system, the PCB generally being created manually, although certain processes (e.g. driver processes for standard devices) may have precoded PCBs which only require linking to the rest of the executive.

A PCB is associated with a process in a one-to-one relationship, but the only link between a PCB and the code of its process is probably the initial entry point, which must be set in the PCB. The PCB contains, essentially, data and work space for the executive and is not directly accessible by its process—this being enforced on systems with store protection. Each process has a PCB, but each process need not have distinct code. Re-entrant code, if the machine allows it, will enable one driver program to handle several identical I/O devices, and to do so in parallel. However, non-re-entrant work- or stack-space will almost certainly be necessary for each process.

Communication between processes

For such a system to work in co-ordinated fashion, processes must be able to communicate with each other. Dijkstra presented the semaphore as the basic tool for communications: it is a stop/go flag, capable of being tested and/or set by more than one process. Processes may use it to signal to each other to halt or to proceed.

To pass information between processes buffers are required. When process A passes a buffer to process B it may need to activate B to enable it to operate on the buffer. Similarly, when process C tries to get a buffer from process D it may need to suspend itself if no

buffer is available. Thus buffer passing is associated with activate/suspend signals which can either be effected by explicit semaphore operations or implicitly built into the buffer-passing operations.

With semaphores and buffers we now have the basic facilities allowing, for example, an input process A to fill buffers and pass them to an applications process B, which operates on the data and fills further buffers for an output driver process C. Processes A, B and C can all run in parallel, control swapping between them in response to semaphore operations (among other causes) with temporary overruns of data being taken up in buffer queues.

It will be seen that semaphores and buffers form two classes of items which do not belong either to the executive or the processes. They are inter-process data and depend on the configuration. Communications between tasks can, of course, be made via common core or disk areas (see below).

Virtual instructions

The basic instruction code of the computer is frequently supplemented with virtual instructions (VIs). These are subroutines which are available for performing certain critical operations: they are coded as such for the usual reason for writing a subroutine (saving repetition of code) but also, more importantly, to incorporate them into the executive. VIs perform operations on executive and other data on behalf of processes, with the result that processes do not have to operate on such data (including their own PCBs) directly. Many VIs can be constructed but there are four basic types which are:

(1) The p- and v-operations on semaphores. The p-operation sets a semaphore to 'stop' and suspends the calling process if it already was at 'stop'. The v-operation activates a process which was stopped on the semaphore, or sets the semaphore to 'go' if no process was stopped. The p- and v-operations thus work on the semaphore (indicated to them by a calling parameter) and on the PCB of the calling or a 'stopped' process.

(2) The Put and Get operations on buffers. The Put operation attaches a buffer to a queue and activates a process if one is stopped awaiting a buffer from the queue. The Get operation takes a buffer from a queue, or suspends the calling process if no buffer is available. The Put and Get operations thus operate on queues (indicated to them by a calling parameter), individual buffers (calling parameter for Put, returned parameter for Get) and on the PCBs of the calling or 'stopped' process.

In all cases, suspension of the calling process (p-operation, Get) results in a reschedule, while activation of another process (v-operation, Put) may or may not cause rescheduling according to choice.

Depending on how the system is designed in detail a fifth VI for peripheral driver processes may be necessary. This VI is:

(3) Wait for Interrupt. The calling process is suspended until the awaited interrupt occurs. This can be associated with 'masking-in' the appropriate interrupt line. Alternatively interrupts can simply add buffers to the process input queue, and the process uses Get instead of Wait for Interrupt. But this method means that a process is looking in two directions at once—for example, it may really be waiting for an output transfer to finish and instead get a further buffer for output.

Since the four basic VIs operate on interprocess data—semaphores and queues—the question of protection arises. Two processes must not operate on the same item of data 'simultaneously'. The easiest way to ensure this is to make the VIs 'primitives', which is

done by inhibiting interrupts or equivalently by incorporating them into the microprogram. A different approach for systems with the need for very fast interrupt responses is to permit interrupt servicing but inhibit rescheduling during the execution of VI. It is important, however, to recognize that it is the semaphore or queue we wish to protect, not the code.[1] Thus a more sophisticated method would be to check that two processes do not operate on the *same* semaphore or queue in parallel by using a super-semaphore of some sort for mutual exclusion. This solution is of interest when two physical processors work on the same schedule of processes, but is too complex for ordinary systems.

In general, therefore, a VI is not entered in parallel and hence is non-re-entrant. It operates directly on executive data such as PCBs and, indirectly, by means of parameters passed to them by the calling processes, on the inter-process data.

Global subroutines

Any sophisticated real-time operating system does not confine itself to basic semaphore and buffer-passing operations, although these are quite adequate for the expert programmer. To isolate from the application programmer the configuration problems associated with creating semaphore and queue structures, subroutines such as Open, Close, Read and Write are provided. These global subroutines (GSs) communicate with device driver processes and contain VI calls within them.

If full device interchangeability is aimed at, a single call Open, say, will serve to open either a disk file or a line printer. The device will be indicated by a parameter with the call. Frequently the parameter is only a pointer to a 'linkblock' in the application code which contains the name of the device. A preamble to Open, or any such GS, is thus a section which determines the target device. This 'entry-to-global subroutine' is often included in the executive since it is short and always wanted. (It and a corresponding exit from GS are essential with memory protection.) However, the main code of Open is not in the executive. Different versions will exist: an Open for disk, an Open for magnetic tape, an Open for teletype or similar character devices, and only those which correspond to an existing device will be included. Thus the 'entry-to-GS', probably coded as a VI, will decode the device and cause entry to the appropriate GS.

A global subroutine is closely related to the corresponding output driver process. Between them are the interprocess data—the semaphores and queues. GSs are doubly re-entrant. Firstly, in certain cases, in particular a disk, processes A and B may want to access the same device X in parallel. By this is meant that B may wish to put a quest-to-Write to the disk driver before A's request-to-Read is satisfied. This enables the disk to optimize search times, and perhaps permits B to continue, assuming its write command proceeds independently. (A and B are not allowed to use X in parallel, e.g. if X is a line-printer, the Open and Close GSs include p- and v-operations respectively on a mutual exclusion semaphore.) Secondly, the subroutine must be capable of handling more than one driver process, so that process A can output on device X and process B on a similar device Y in parallel. If the GS called by both A and B is not re-entrant, entry to it would have to be serialized, with the probability of a hold-up on device Y if one occurs on device X.

In practice this double re-entrancy presents some problems. A GS must not only know the semaphores and queues of its calling process, it must also know those of the particular target device which it is addressing. One solution is as follows:

Each process has an input queue with a pointer to it in its PCB. In the case of driver processes, which are in the nature of 'slave' processes, requests for I/O transfers are attached to this queue. The queue is found using the target device identifier, passed to the

GS, to access the driver's PCB. Application processes' input queues are used by GSs to await the results of I/O transfers. A GS locates the queue using the calling process' identifier to find its own PCB.

A source of empty buffers must also be available. A pool common to a group of devices handled by a re-entrant GS-driver pair is often adequate. The address of this pool can be explicitly programmed into the code, although the size is probably fixed at configuration time. Application processes may have a second input queue where they receive messages from other application processes: these are probably passed using a calling GS associated with the process. (Such messages, unlike the replies from I/O requests to drivers, are unsolicited.)

Finally, besides a PCB, drivers are usually given a device control block (DCB), chained to the PCB. A DCB will contain data particular to one device being handled by a re-entrant driver, for example its semaphore (for seizing the device) and the device's physical address.

Structure summary

The basic structure may now be summarized (Figure 1). We distinguish five main module types, which are central to the discussion of linking below. They are:

(1) The executive including VIs, and detailed in the third section.

(2) Application processes' code, possibly with associated GSs and buffer pool.

(3) Configuration data for application processes, including their PCBs and input queues structures.

(4) Peripheral driver processes and GSs including the code of the processes and the GSs, and associated common data such as buffer pools.

(5) Configuration data for driver processes, including the PCBs, DCBs and input queues for the drivers.

EXECUTIVE CONSTITUENTS

Although the executive is central to the operating system, it is relatively small in size. A powerful executive for a minicomputer can easily be written in 500 to 1,000 words of store. Its function is to control the driver and applications processes which form the rest of the system. It is composed of four main components: initializing code, interrupt handling, scheduling and virtual instructions (VIs).

Initialization code

The system must be self-initializing, this feature being particularly important during program development where manual resetting of PCB contents, semaphores, queues, etc. is unacceptable. Generally, the policy followed is that the executive initializes its own data and the PCBs, and then schedules each process in turn after arranging that they start their individual initialization. A typical process initialization will involve resetting of counts and clearing its queues and semaphores. This may be done by a VI, for example, to clear the process' input queue; or possibly by a call on a GS to create a pool of buffers, e.g. for file transfers to and from disk. Clearly, if pools are being recreated, the space formerly occupied by them must previously be realeased by the executive initialization: the free store pointer should be reset to its value when the system was first loaded.

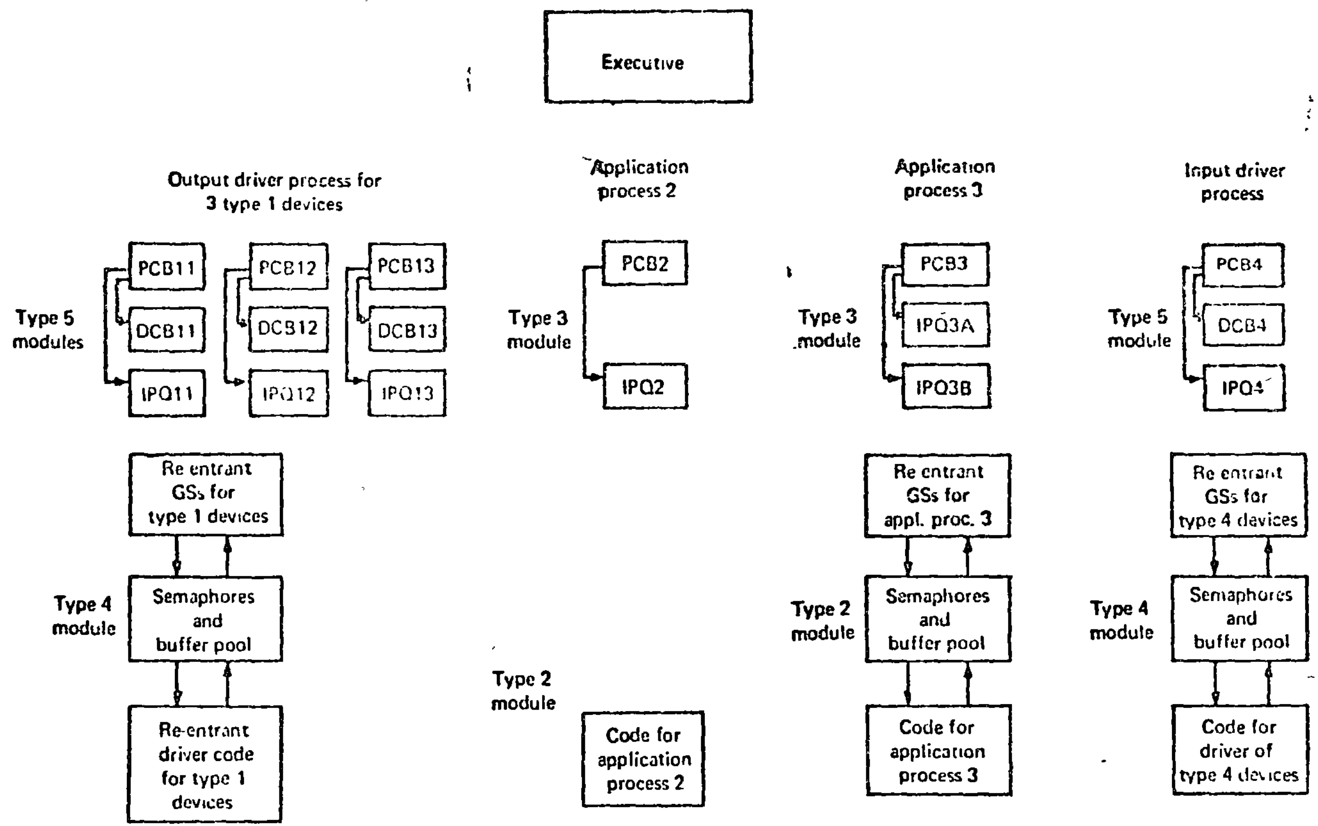


Figure 1. Example system containing two application processes, one input driver and three output drivers for identical devices

Interrupt handling

The function of interrupt handling is:

(1) To identify the interrupt.

(2) To activate the appropriate process.

(3) To schedule.

Certain interrupts requiring very fast response may be handled directly using a simulated direct-memory-access technique, either in the microprogram (cf. automatic I/O on the Interdata machines) or the executive itself. Again, depending on the inhibit/enable and priority strategy followed, it may be that an interrupt arrives for an already active process; in which case a 'fielding' technique, which adds the interrupt to the process' input queue, is used. Generally, interrupt identification is largely automatic using interrupt vectors (e.g. on the PDP-11).

A special interrupt handler is often provided for the clock. Basically this performs count-downs on processes which are suspended awaiting an explicit or default time-out. Such processes have their PCBs chained to the clock handler.

Scheduling

Scheduling can be very simple—pick the first active process encountered—or quite complex, involving assessing the relative priorities of active process, the time elapsed since they were last run, etc. Contrary to experience on large machines, where there are big overheads in changing processes and a finely tuned scheduling algorithm can be very important, many mini-systems, particularly core-resident ones, are insensitive to scheduling strategy provided that no obvious stupidities are committed.

Virtual instructions

The VIs discussed so far are normally privileged and may only be accessed by GSs, drivers or other parts of the operating system. Many more VIs can be invented. Typically they are used to administer processes: kill them, suspend them awaiting a time-out, change their priority, etc. (cf. the meta-instructions of Data General's RTOS). Such VIs are called from the processes themselves, or by the operator via a command interpreter process. A further two may be the Entry-to-GS and Exit-from-GS VIs, calls on which are normally embedded in the macro expansions of Read, Write, etc.

In practice, entry to VIs is usually performed by trap instructions, supervisor calls or internal interrupts which allow a simultaneous change in processor status (which is convenient for moving to a non-re-entrant mode of operation), and do away with the need for linking the VI entry points to the processes' code. The return links present no problem if the VIs are non-re-entrant. If the VIs are re-entrant, or in the case of the entry-to-GS VI which is followed by a re-entrant GS (which may call further VIs and GSs), the ideal place for saving links is on the process' private stack.

Re-entrancy and interruptability of the executive

It has been shown that VIs are best coded non-re-entrantly, although this may not always apply. The question of re-entrancy and interruptability, however, is posed for other parts of the executive code.

Many approaches can be taken and they are likely to be hardware dependent, but a good policy is as follows:

(1) Interrupt handling code is interruptable and re-entrant, provided there are adequate stack facilities. On terminating a pass through the code, return is 'down-the-stack' to the last interrupted pass. If no such one exists, control proceeds to the scheduler.

(2) The scheduler's links and work space are disposable. If it is interrupted a reschedule will occur, and the half-completed schedule operation can be forgotton.

(3) Post-scheduling code is re-entrant or non-re-entrant depending on the cleverness of the coder. There are critical moments between the selection of a process from the active chain to its actual entry when such items as the current process indicator, the stack pointer, etc. are vulnerable. The code can, however, be interruptable with rescheduling inhibited, like a VI; indeed post-scheduling is similar to a VI in the sense that it is a sort of run-in to the process code.

(4) All interruptable but non-re-entrant executive code (implying an inhibition of rescheduling) must terminate with a check whether a forced reschedule should take place, owing to a received but unserviced interrupt.

These conditions bear on the executive's ability to respond rapidly to interrupts. They are, however, more concerned with aesthetics than practicalities.

AN EXAMPLE—BASIC I/O

As examples of the use of semaphores, queues and the basic VIs, two simple global subroutines and corresponding drivers are sketched. One is for output of a character buffer, e.g. to a printing device; one is for input, e.g. from a keyboard.

The configuration of Figure 1 is used. A common buffer pool is shared between all three identical output devices. It is supposed to have been created by the initialization routine for the device, and arrangements must be made so that only one pool is formed although all three devices are initialized by three calls on the re-entrant driver.

To output to the device the programmer writes:

Write (⟨device name⟩, ⟨private buffer pointer⟩)

A macro-assembler converts this into a trap or supervisor call instruction, the parameters being passed on the process' stack, or from the locations following the trap instruction. The 'entry-to-GS' VI uses ⟨device name⟩ to index a table which contains the address of the Write GS for the device, and an identifier for the device (e.g. a pointer to its driver's PCB). The VI causes entry to the Write GS, leaving the return links on the process' stack.

Figure 2 illustrates the re-entrant Write GS. A buffer is got from the common pool using the Get VI. The pool is known to the GS since it is internal to the GS/driver program module. The GS copies from the process' private buffer to the pool buffer, puts the pool buffer to the driver process, using the device identifier, then exits back to the calling process.

The re-entrant driver gets the buffer from its input queue using the Get VI. The driver knows the input queue for the specific device, the pointer from the PCB perhaps being loaded into a program-accessible register on each activation of the driver made on behalf of a device. The driver empties the buffer character by character, waiting for an interrupt to signal the readiness for the next one. When the buffer is empty the Put VI returns it to the pool.

More elaborate approaches are possible. There could be separate buffer pools for each device; to avoid copying from a private to a pool buffer the application programmer could be given another macro which would get him a pool buffer for filling prior to calling Write; Open and Close macros could be added to seize the device for a given application process; ⟨device name⟩ could be held in a 'linkblock' and a pointer to the linkblock passed as a parameter so that altering the contents of the linkblock would enable the device to be readily changed; error messages could be returned to the calling process which would be obliged to wait to check successful output, etc.

In Figure 1 the GS/driver pair for input are supposed re-entrant although there is only one device. There are command and data buffer pools common to the module. In general, input must be requested by a process, either to activate the device (e.g. a paper tape reader) or to tell the driver the ultimate destination. Thus, in Figure 3, the Read GS passes a

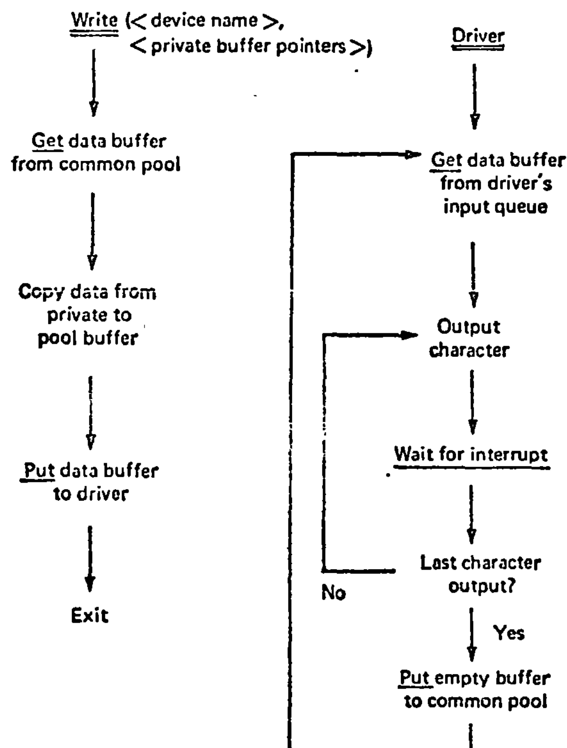


Figure 2. Simple output, with copying, using a Write GS

command buffer to the driver before receiving a data buffer from it. It also seizes the device, so that two processes do not request input in parallel on the one device and interchange their data by mistake.

The process uses the macro:

Read (⟨device name⟩, ⟨private buffer pointer⟩)

This results in entry to the Read GS (as with Write). The device is seized using the p-operation VI on a semaphore found in the DCB, via the device identifier. A command buffer is obtained from the local pool, filled with the calling process' identifier (say a pointer to its PCB set up in a register by the entry-to-GS VI), and the buffer is Put to the driver. The GS gets the returned data buffer from the input queue of the calling process (found from its PCB), copies it to the private buffer indicated by the calling parameter, puts the

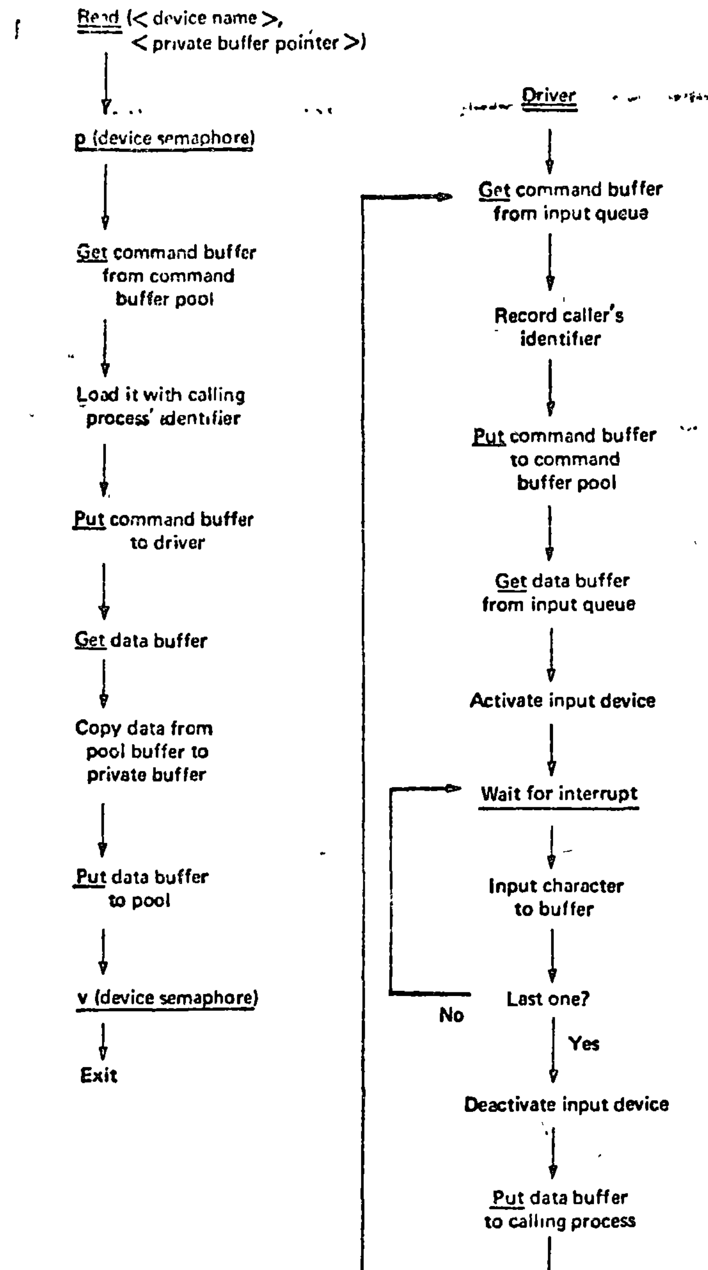


Figure 3. Simple input with copying, using Read GS

data buffer to the local pool, calls the v-operation VI to release the device and exits. The driver is self-explanatory. It uses the caller's identifier in the command buffer to locate the input queue for the data buffer. Again, input allows many variations: avoid copying by supplying an explicit return-data-buffer GS; build in an echo facility for keyboard, etc.

These two examples show that the drivers and GSs are, in a certain sense, trusted code. They do not operate on executive data such as PCBs, which is done for them by VIs, but they do use pointers to PCBs, DCBs, etc. which they could corrupt. However, the applications programmer is prevented from handling such entities directly.

A DATA BASE AND FILING SYSTEM

Central to almost all real-time operating systems is a common data structure. Such a structure does not belong to any individual process, but is shared between several processes. It may be a core-resident data base or it may be a collection of files on disk. Typical examples are: a data base containing the current state of the plant in a process control application, the data base being updated by scanning processes and accessed by control, logging and display processes; a routing table in a message switching system, being accessed by many processes for reading, and occasionally being written to when the network is changed; tables of stock records in a small real-time stock control system, again being read from some sources and written to from others. In the very simplest case, where the data base is only read from and is core resident, there is no need to provide any centralized accessing system: all processes can read in parallel. But if the data is changing, then it is important that those processes which are reading the data have controlled access to them, otherwise they may read half-changed information. If the data base is on a backing store, obviously access must be further controlled to stop processes issuing direct commands to that store in parallel. A disk driver process must be supplied, in the same way as a driver for any other I/O peripheral must exist.

Basic structure

We suppose that the data base is composed of individual files. A file is a collection of data which has a unique accessing channel so that, whereas processes A and B can access files X and Y respectively, completely independently, they cannot both access file Z without being aware that the other is also doing it, and acting accordingly. Access is controlled by the basic Open and Close commands. These are calls to global subroutines (GSs), described above. They use parameters passed from the calling process which specify the file name, and other information such as the access required (read/write), and perhaps lockout-avoiding information such as the maximum number of file buffers required at a time by the process. The purpose of the Open GS is to establish a core-resident file header block (FHB) which will subsequently be used to control all transfers to or from the file. This FHB will be filled initially from the file header, which will be found via the file directory (on disk, if it is a disk file), and remains in existence as long as the file is open. All processes opening files must first check for an already existing FHB, which is done for them by the Open GS, and if they find one either use it or suspend themselves (on a file protection semaphore), depending on the details of the system. If many processes are allowed to have the same file open at a time, then the FHB will contain an Open Count, amongst other things. Closing the file will decrement the count and when it reaches zero the FHB is relinquished.

A process may read from or write to a file which it has already opened. Read and Write are also GSs, which check that the file is open, and then perform the transfer. With a core-resident data base these GSs are relatively straightforward, but with a filing system on disk, writing involves all the business of getting buffers and putting them to the disk driver, already discussed. Read and Write have as parameters, besides the file name such items as the word or record number at which the transfer is to start, and the number of words or records to be transferred. Using information the FHB obtained from the directory, they are translated into absolute core or disk addresses.

Two more GSs, Create and Delete, are necessary to allow files to be created and deleted on-line. They have as parameters the file name and, in the case of Create, additional items such as file type, size, etc. Their function is to add to or subtract from the file directory, and to book or release space on disk.

Create, Delete, Open, Close, Read and Write are the basic GSs for accessing the data base. They can of course be elaborated, in particular to distinguish between transfers directly to and from system buffers and those which use copying. It may also be convenient to supply a GS which allows the entire filing system to be by-passed. This is particularly convenient for providing overlay and swapping facilities which are performed in any case by trusted code.

File design

The designer of a small real-time filing system is faced with a serious problem. Either he maintains a compact and fast access system, implying many limitations, or he supplies a general system with often massive overheads. Generally and wisely, the manufacturer imposes fairly stringent limitations on the files. If we consider a disk-based filing system, we are confronted with decisions on many subjects, of which the following list is a sample.

(1) The types of file.
Basically the files are either fixed in size when created or have the capacity of growing. Fixed size files are best allocated contiguous space on a disk but files which can grow will need either an indexing or chaining structure.

(2) Sub-divisions within file.
Files can be composed of a continuous stream of data whose structure is only intelligible to a program which can read it, or the data may be split up into blocks of characters, individual words or records. These may or may not be of fixed length.

(3) Space allocation.
A record must be kept of the free space on disk. This can be done by maintaining a map of all the disk sectors, or chaining all the free sectors together (with the risk of losing sectors should the chain be broken). If the record of free pages is kept in core, it will occupy a lot of space and also leave the disk meaningless in case of a computer failure, or transfer of the disk to another machine.

(4) The file directory.
Besides the files themselves and the record of the free space a third class of data must be kept on disk, the directory. This enables files to be found by name. Each entry in the directory contains the basic header information for forming FHBs when opening a file.

(5) Off-line compatibility.
It is desirable that all data on the disk is itself file structured in some sense. Ideally, the directory, free sector record and on-line files should be accessible as ordinary

files to some off-line disk operating system. This enables their establishment in the first place, and their analysis later.

As regards the facilities provided to the on-line user, there are additional decisions:

(6) Probably a limit must be put on the number of files a given process may have open at the one time. This then puts some limit on the number of buffers for FHBs which need to be provided by the system.

(7) Either only one process is allowed to open a given file at a time, in which case suspension mechanisms must be implemented for other processes, or decisions must be made on multi-access methods. For example, many processes might be allowed read access in parallel but only one to have write access at a time. Again, this sort of access might be applied to records rather than to the file as a whole.

(8) Depending on the transfer methods used, there is probably a restriction on the maximum number of data buffers which a process may have at any time. An infringement, for example by attempting to read more data from a file before the previously read buffers had been returned to the system, would cause the calling process to be aborted. This maximum number could be allocated at configuration time, or could be passed as a parameter when calling the Open GS, and vetted by the system using, for example, Dijkstra's bankers' algorithm.[2]

The decisions made in such area will radically affect the size of all the data structures involved. For example, a system permitting multi-access to a given file but with control on access to a record within that file would require an FHB with space to contain, in principle, the current record number for every process which had the file open at that time.

Mechanisms of file handling

The mechanisms used are obviously highly dependent upon the design of the files which they must support. The basic principles have already been outlined and are illustrated again in Figure 4. The process wishing to access the filing system enters a GS, such as Open, which puts and obtains buffers between itself (on behalf of the calling process), the disk driver and the buffer pools. In the case of the Open GS, after a check that there is no already existing FHB for that file, the GS puts a request to the disk driver for a page or pages from the directory, obtains the pages back (off the calling process' input queue) after an appropriate delay, obtains a buffer area in which to set up the FHB and then returns control to the calling process. This is a simplification. The request buffer must be obtained from a pool, and the buffers with the directory pages must be returned to a pool. The search through the directory may be done by the GS (with repeated calls for directory pages), or by the disk driver, or be avoided altogether by having some skeletal core-resident index which enables the appropriate page of the directory to be found directly.

It will be seen that there is a great deal of scope for lockouts of one form or another. By a lockout we mean what is often called a 'deadly embrace', wherein process A cannot proceed until process B has finished and process B cannot finish until process A has finished. Wherever processes are accessing buffers or other resources in parallel, lockouts can occur because one process does not release the buffers it already has until it has obtained some more. For this reason many small real-time operating systems have limits on the number of buffers per process and ensure that sufficient buffers are provided so that each process may at least go up to this limit.

There is also scope for introducing chaos through parallel access. It has been said that, in general, GSs are re-entrant, but if we consider the case of the Open GS, it is clear that two distinct processes may try to open the one file simultaneously, both finding that it has no FHB and so both creating one for it in parallel. This could be avoided by protecting with semaphores that part of Open which starts with a test for an existing FHB and which ends with the establishment of one after a successful read from disk. Alternatively, the protection could be made much shorter allowing the creation of a rudimentary FHB prior to the

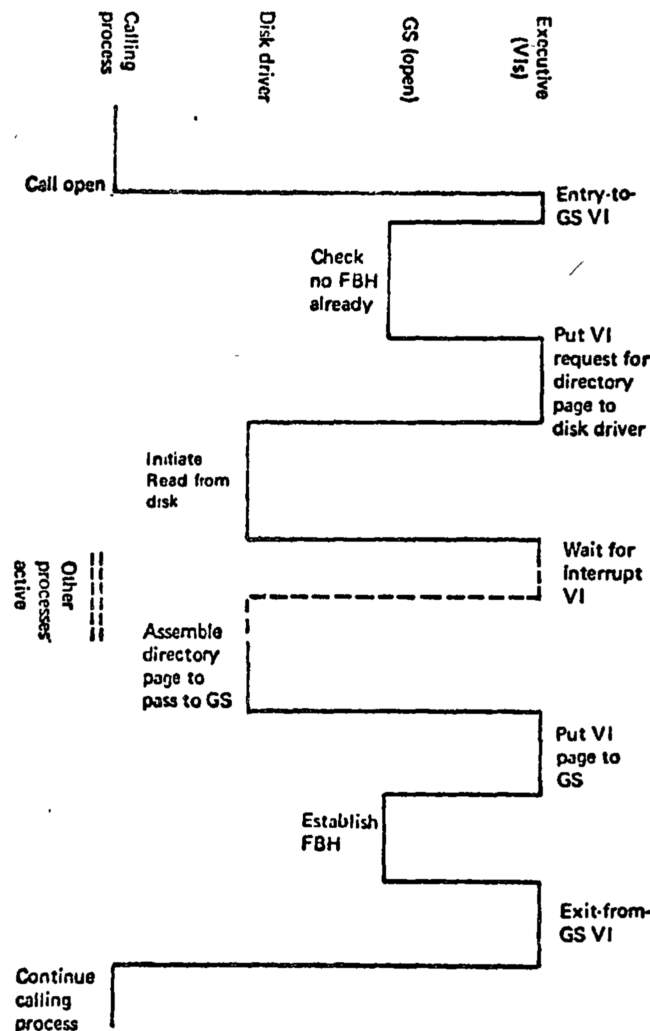


Figure 4. Simplified diagram of opening a file on disk

request for directory information from the disk. This serialization of calls on the GSs of the filing system is highly undesirable in the case of Read and Write, can be avoided with some ingenuity in the case of Open and Close, but is frequently necessary in the case of Create and Delete. This is a measure of how much of the data is in core and how much on disk. In the case of Create and Delete it is essentially disk data which is being changed, and

it is important that two processes do not try to add to the one directory in parallel or to take file space on disk in parallel.

If there is more than one disk it is natural to use a re-entrant driver which will handle all of them. The driver obtains requests from its input queue, orders those requests to optimize disk access times, picks the highest priority one, initiates the seek operation for it and then waits for the interrupt which signifies that the transfer is complete. On output this means that the buffer is available for return to the pool; on input it means that the buffer must be put back to the input queue of the process which called the GS which asked for it. But if several disks are on the one controller, the re-entrancy of the driver must be limited. In general, it is possible for seeks to proceed in parallel, but not for the actual data transfer. So on a moving head disk, once the track has been found, the driver must book the controller with a p-operation on the appropriate semaphore, initiate the search and transfer and finally release the controller with a v-operation. Many minicomputer systems use head-per-track disks which simplify both the design of the driver and that of the optimizing technique used on requests for sectors.

Other considerations

It will be appreciated that the design and implementation of an adequate filing system is probably more complicated than that of the basic executive itself. It is certainly larger. By the time GSs, drivers, tables, semaphores, FHB buffers and data buffers have been allowed for, probably 4,000 words of store have been consumed. This would only cover a basic filing system.

As an example of additional facilities, three subjects can be mentioned.

Shutdown

It is desirable to be able to shut down all or part of the filing system in an orderly fashion. This would involve closing all open files, releasing all buffer space, etc. In particular, it is a very useful facility to be able to close down an individual process without locking up the filing system. It must be possible to identify all the open files belonging to that process and to close them using some supervisory process.

File capabilities

Files do not always exist as a data base. They may be passed about in the same way that core buffers are passed between processes. A typical example is in a message switching environment, where the files are the messages which are being passed through the system.[9] The names of files are passed between processes (which will be discussed later under store protection). Under these circumstances, the Create and Delete GSs must be expanded to take into account the fact that while one process may wish to delete a file, another process is still using it. Existence counts for files must be built into the directory. This is another reason for serializing the Create and Delete GSs.

Recovery

The whole theme of security and recovery has not been touched on. It is sufficient to say that if files are to be checkpointed and audit trails are to be maintained for them, the checkpoints and audit trails will themselves be files, perhaps of a special sort, perhaps standard ones. They must obviously be kept on a different disk to the one whose data they are recording.

COMMUNICATION WITH THE OPERATOR

In the field of communication with the operator, there is often a big difference between the facilities offered in a manufacturer's package and those provided on a special-purpose system. Obviously, a package must be more general, and therefore offer more. However, it is also true that once the basic communication facilities have been provided, it is very easy to expand them, and unless this temptation is resisted, a lot of unnecessary and store-consuming items can be included.

Messages to the computer

Messages to the system as such, as opposed to messages to any individual process, are frequently identified by having the initial character a full stop, a dollar sign or some other special symbol. When typed in at a keyboard, the input process which buffers the characters into a message, knows to put the completed message to a special process sometimes referred to as the Command Interpreter Process (CIP). The CIP examines the message and, if it recognizes it, it acts on it. If it does not recognize it, it outputs an appropriate error message to the terminal whence it came. The method whereby the command is effected depends upon what it is, but in many cases a VI is the best: this particularly applies to commands which affect executive data, for example, the status of processes held in their PCBs. Other commands can be effected by passing buffers to appropriate processes, or possibly by direct calls on GSs. Examples of such commands are as follows.

Basic real-time data

Commands which set up the time, the date, the identifier for the current operator, the machine number and any other information of this sort which will subsequently be required for logging on printouts.

Control over processes

These commands include ones which will change the priority of processes, activate them, stop them, make them continue after a stop, abort them completely or perform any other operation which effects their position and status on the schedule.

Re-configuration of I/O devices

The simplest illustration is a change of control console. The operator may, for example, designate another teleprinter as the console. In more sophisticated systems he may change peripherals, causing data which was coming out on a line printer to come out on a paper tape punch. There are various methods of doing this, which vary from simply changing the peripheral hardware addresses in a driver (assuming that the new device is compatible with the old, and that there is no contention for it), to changing the device name held in the linkblocks of the application processes. This latter is the best method as it results in a completely different GS being called, so that in principle output could be changed from a device like a teleprinter to one as different as a magnetic tape. Clearly, there are problems as to when this takes place. We do not wish to change from device A to device B when device A has already been opened by the process, and device B has not. Here the facility to make a process release all its claims on peripheral resources is obviously important. To enable the CIP to locate all the link blocks in the system, they will probably have to be changed to some table structure in the executive. The CIP, using a VI, can scan this chain

for the names of the devices which are to be interchanged. The setting up of such a chain must be done at the initial configuration time.

Programmers' aids

Facilities can be provided to enable a programmer to monitor and if necessary change given locations in core or on disk The message to the CIP may use absolute addresses, with all the dangers that this implies, or may refer to structures by symbolic names. In this case the CIP would have to open a disk file, for example, and perform read/write GS calls just like any other process.

Messages from the computer to the operator

These messages may be of any type whatever, since the operator is the only person who has to interpret them. They fall into two categories: those which require a reply, and those which do not. Both are initiated by a special write call to a GS which handles the operator's console, which is not necessarily fixed (see above). The buffered messages are output in the ordinary way, except that messages which await a reply must previously arrange for the console to be seized, using a mutual exclusion semaphore, to ensure that the reply returns to the sender. Typical uses of this facility are: statistical outputs about devices for the system as a whole, and error messages. Error messages frequently require a reply, to tell the system whether to proceed or to try again. Irrecoverable errors will require the abortion of the associated process, and if this is done by a system message, as opposed to a reply to the error message, care must be taken to ensure that among the other resources released by a process is included the operator's console.

More sophisticated communication

The facilities mentioned above are concerned with the adjustment of the existing real-time operating system, rather than with additions to it. Sophisticated systems allow the incorporation on-line of new programs and data. Indeed they frequently permit the background compilation, editing and debugging of programs which can then be established on file, and later incorporated into the on-line system. If this background batch processing facility is supplied (see Part 2) the whole job control language associated with such work is available to the operator. The output is a file or files for incorporation in the on-line system, discussed below.

Once a new process is inserted, it must be activated, either directly from the console or by changing the internal routing system so that it becomes the destination for a message from another process.

In the same way that new code may be added to the system, it is possible to add new data in core or on file. If the data already exists on disk it is only a question of giving a 'capability' for it to the relevant processes. This subject will be discussed more fully in Part 2.

CONFIGURING, LINKING AND LOADING

Given the existence of program modules which are to make up a real-time system, some of which may be library modules, some specially written and others standard modules with modifications for the particular application, the problem remains: how are these modules to be placed in store to form a coherent whole? (Here we are only considering core-resident programs without memory protection—i.e. no distinction between virtual and physical address space.)

The creation of a system out of individual modules can be done at the most simple level by coding PCBs and other configuration data by hand and using the standard linkers and loaders supplied for assembler object code; or, at the most sophisticated level, using an interactive system generation program, which enables the user to call in modules from backing store and link them into the system in a conversational mode. Here again the manufacturer is likely to supply much more comprehensive facilities than are necessary for the construction of a one-off system. But, with or without system generation aids, it is in this area of configuring, linking and loading that much confusion often exists in practice. Probably, this is because it is not taken seriously enough, and the user assumes that it will be straightforward without foreseeing any of the problems.

Configuring

If we consider a system composed of modules as outlined in Figure 1, we see that there are five basic types of modules involved. Type 1, the executive, is largely fixed, and only needs configuring to the number of processes and of GSs in the system. The processes are identified by the PCBs, which are chained together, so that the executive needs to know only one member of the chain to be able to find all the others. The GSs are found using a look-up table which translates macro calls and device names as found in the linkblocks, into entry addresses for the GSs. Assuming that the six macro calls, Create, Delete, Open, Close, Read and Write, can handle all devices, then each device in the system will require six entries in this table, which can be conventionally ordered. Thus configuring involves making a $6n$ table for n devices, and incorporating a decoding mechanism for each of the n device names. If application processes or core-resident data bases are also accessed by GSs, the table will have to be suitably extended.

Type 2 modules, application processes, require no configuring unless they have GSs and buffer structures to enable them to be called by other application processes. In this case the basic structure already exists in the code, and all that is required is to allocate the number of buffers in the pool and possibly the size of each. This is best done by arranging that buffer creation is actually performed at initialization time, so that all that is inserted at configuration time are appropriate numbers. When the process is initialized it uses these as parameters on calling a global buffer creation subroutine, which can conveniently use up all the free store in the system, in particular by over-writing the loader.

Type 3 modules are the configuration data for application processes. They include the PCBs and their dual input queue structure, one queue handling buffers from I/O devices, the other being for messages from other application processes. These queues will be initialized as empty, so configuring essentially involves filling in any data in the PCB which is not already there or going to be satisfied by the linking procedure. Such data might include the priority being allocated to the process, the size and possibly the initial address of the stack-space for it.

Type 4 modules, which are the I/O drivers and associated GSs, require some thought. Configuring here is concerned with allocating buffer space. If we consider a filing system, such as described in the fifth section, two types of buffers have to be made available. One type is for FHBs, the other for data. These buffers are going to be used by all processes which access the filing system, so the number allocated must be related to the number of such processes in the system and the maximum demands likely to be made by such processes. For example, if n processes can access the filing system and each is allowed to have m open files at a time, nm FHB buffers should be created. If no process is allowed to have

more than p data buffers at a time, provided some sort of bankers' algorithm is implemented, then p data buffers are the minimum to be supplied. Some figure between p and np is what will probably be provided in practice. As stated earlier, buffer set up is best done at initialization time.

Type 5 modules, configuration data for I/O drivers, include the PCBs, the DCBs and the input queues for those drivers. A PCB must be created for each device capable of being driven in parallel with other devices, and work- or stack-space for the process must be indicated in it. Similarly, the DCB must include the specific I/O addresses for this device.

It will be seen, below, that modules types 1, 3 and 5 contain pointers to other modules

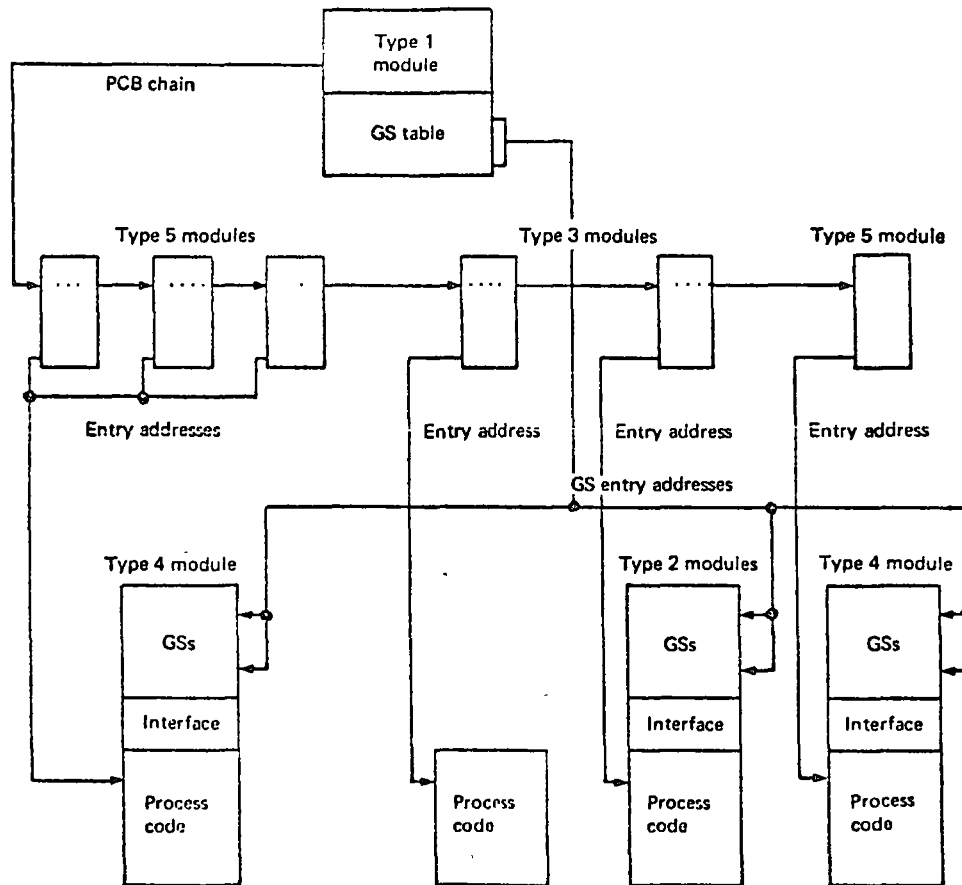


Figure 5. Linking the modules

which will be resolved by linking. These symbolic pointers, typically the names of entry points to GSs and processes, must also be filled in at configuration time. Configuring can obviously be done manually, the result being a series of modules to be presented to the linker, but it can also be done interactively. A special configuring program, run off-line, can accept each module, recognize it for what it is and ask questions of the user. Thus, when confronted with module type 1, it would ask the user for the names of the devices to be supported, and continue creating the entry-to-GS table, until the user said there were no more devices. Similarly, a basic type 5 module would be filled in conversationally, and indeed repeated if there are several similar devices, for example teletypes, in the system.

Linking

A basic link-editor's function is to link together modules which are simultaneously allocated core store, either by the link-editor itself or by the operator. Figure 5 illustrates the links which must be made. They are:

(1) The links between the executive GS table, set up while configuring, and the GSs' entry addresses.

(2) The links between the executive and the PCBs, chaining them together to form a basic schedule.

(3) The links between the PCBs and the code of the associated processes. Again, these are initial addresses. If a process' code is being used re-entrantly then several PCBs will point to the one piece of code. The product of the link editor is a binary file for the loader.

As in the case of configuring, a more sophisticated approach is possible, in which a special-purpose linker recognizes the modules for what they are. Indeed configuring and linking can be made one operation. As an example one can consider the chaining of link-blocks in application code, to enable device independence. Normally, linking involves matching a reference in module 1 to an external symbol which appears as a label in module 2. To avoid saving the complete symbol table for module 2, it is possible to declare a list of external symbols (those referred to from outside) at the head of module 2, and this list alone is considered by the linker. However, this method does imply that the coder of module 1 knows the symbol in module 2. In our example, he may not. All he knows is that his link-blocks are to be chained to module 2's linkblocks. (A similar situation exists for chaining PCBs.)

The solution is to declare, under the title 'Linkblocks', a list of each one, at the head of a type 2 module. The special configurer/linker recognizes the declaration and chains the items to any previously processed linkblocks. The operator can also be asked by the configurer/linker to fill in actual device names in the linkblocks.

Loading

If the above procedures have been followed, loading is a trivial operation. A binary core image is loaded directly to its final location. In general, the modules presented to the linker before address allocation, are all relocatable with the exception of type 1 modules. Type 1 modules include interrupt and trap locations and other dedicated addresses. When linked, therefore, the total system is no longer relocatable, but hopefully will allow the includion of 'gaps' in the contiguous load module, which will permit the system to be made up of more than one such module, interleaved, if so required. This facility can be very important on systems where there is a distinction between virtual and physical address space. Either the linker or the loader should be used to provide a free store pointer, which is the base from which buffers are created at initialization. This free store pointer is adjusted as each buffer pool is formed.

Adding to the system on-line

If a conversational configuring program is provided there is no reason why it should not run on-line. A new module, e.g. a driver and GSs for a new peripheral, can be configured and then linked to the system. Linking involves extending the GS entry table in the executive, extending the PCB chain and allocating store for the code and work-space.

If the symbol table from the initial linking is still available, an on-line linker can use it to perform its functions. Store can be found using the free store pointer, although this would mean that entities like the GS table were not a contiguous whole. The on-line linker works on disk files using the filing system, and an on-line loader can finally load the binary file to store. Such a facility can be provided for basic core-resident systems, but it is likely to be clumsy; it is usually only supplied in the circumstances when a full overlay and core allocation system is already available for use by it.

Summary

Building the complete system involves selecting the necessary modules, filling in data internal to them and cross-references between them, allocating store to each and inserting them into it. Well organized, this repetitive task, performed module by module, is susceptible to automation—and one has a system generation program. Badly organized, it can become a nightmare of unresolved external references, store allocation and relocation problems.

REFERENCES

1. A. St. Johnson, 'The MACE technique for real time programming in small computers', *Software World* (1973).
2. E. W. Dijkstra, *Co-operating Sequential Processes* (Ed. F. Genuys), Academic Press, London, 1968.
3. D. W. Barron, *Computer Operating Systems*, Chapman and Hall, London, 1971.
4. W. F. C. Purser, 'Software protection in multiprogramming on small computers', *Management Informatics*, **1**, No. 5, 203–210 (1972).
5. J. M. Taylor *et al.*, *HIVE—A High Integrity Virtual Machine for Multiprocessor Systems*, Signals Research Development Establishment, Christchurch, Hants., 1972.

DIRECTORIO DE ASISTENTES AL CURSO DE APLICACION DE MINICOMPUTADORAS
(28 Y 29 DE NOVIEMBRE 5,6,12 Y 13 DE DICIEMBRE DE 1975)

NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
1. ING. JOSE H. BERLANGA OCHOA Edif. "A" - 9-401 Col. Torres de Mixcoac México 19, D. F. Tel: 5-93-42-52	SECRETARIA DE RECURSOS HIDRAULICOS Viena 20 Desp. 301 Col. Juárez México 6, D. F. Tel: 5-92-35-68
2. SR. FLORENCIO BOBADILLA MENDOZA Adrian Brouwer 233-3 Col. Mixcoac México 19, D. F. Tel: 5-98-08-03	DEPARTAMENTO DEL DISTRITO FEDERAL Plaza de la Constitución México 1, D. F. Tel: 5-21-12-39
3. ING. CARLOS BREÑA Merida 21-7 Col. Roma México 7, D. F. Tel: 5-11-50-82	CONSULTORIA ELECTRONICA Y ADMINISTRATIVA, S.C. P. Antonio de los Santos No. 70 Col. Tacubaya México 18, D. F. Tel: 5-16-99-20
4. ARQ. MAURICIO EPELBAUM Sierra Fria No. 505 Lomas de Chapultepec México, D. F.	INMOBILIARIA CISA Havre No. 30-101 Col. Juárez México 6, D. F. Tel: 5-33-60-45
5. SR. JOSE ANTONIO ESPINOSA Palenque 302-Bis. Col. Narvarte México 12, D. F. Tel: 5-23-24-27	DESPACHO PARTICULAR Av. Cuauhtémoc 1486-301 Col. Santa Cruz Atoyac México 13, D. F. Tel: 5-34-88-55
6. SR. DANIEL FISCHER DUBSON Fte. del Rey No. 94 Tecamachalco México 10, D. F. Tel: 5-89-33-58	

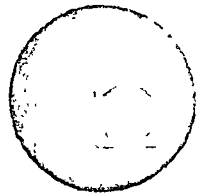
DIRECTORIO DE ASISTENTES AL CURSO DE APLICACION DE MINICOMPUTADORAS
(28 Y 29 DE NOVIEMBRE 5,6,12,Y 13 DE DICIEMBRE DE 1975)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
7.	ARQ. JORGE FLORES Salvador Díaz Mirón No.354-7 San Jacinto México 17, D. F. Tel: 5-41-53-24	INMOBILIARIA CISA. Havre No. 30-101 Col. Juárez México 6, D. F. Tel: 5-33-60-45
8.	ING. MARIANO GARCIA MALO Y G. Retorno Prolong. Moctezuma Oriente No. 3 Col. Romero de Terreros México 21, D. F. Tel: 5-54-72-76	SECRETARIA DE OBRAS PUBLICAS Niño Perdido y Av. Fernando 268 Col. Alamos México, D. F.
9.	ING. ROBERTO GARIBAY SOLORIO Santa Ana No. 10 Fracc. Capistrano Atizapan de Zaragoza Edo. de México Tel: 3-97-66-90	SECRETARIA DE OBRAS PUBLICAS Fernando No. 268 Col. Alamos México 13, D. F. Tel: 5-90-83-52
10.	ING. IGNACIO HOLTZ HALE Angel del Campo No. 3 Satélite Novelistas Edo. de México Tel: 5-72-19-14	DESPACHO CALCULO ESTRUCTURAL Hamburgo No. 172-4o. Piso Col. Juárez México 6, D. F. Tel: 5-33-57-26
11.	ING. RENE MIJANGOS NUÑEZ Retorno 24 No. 35 Col. Jardín Balbuena México 9, D. F. Tel: 5-71-08-21	ECTRO, S. A. Av. Chapultepec 153-106 Col. Nueva Campestre Churubusco México, D. F. Tel: 5-71-08-21
12.	ING. BULMARO MORALES LEON Miraflores 137-A-15 Col. del Valle México 12, D. F. Tel: 5-43-08-07	SECRETARIA DE OBRAS PUBLICAS Xola y Av. Universidad Col. Narvarte México 12, D. F. Tel: 5-19-27-70

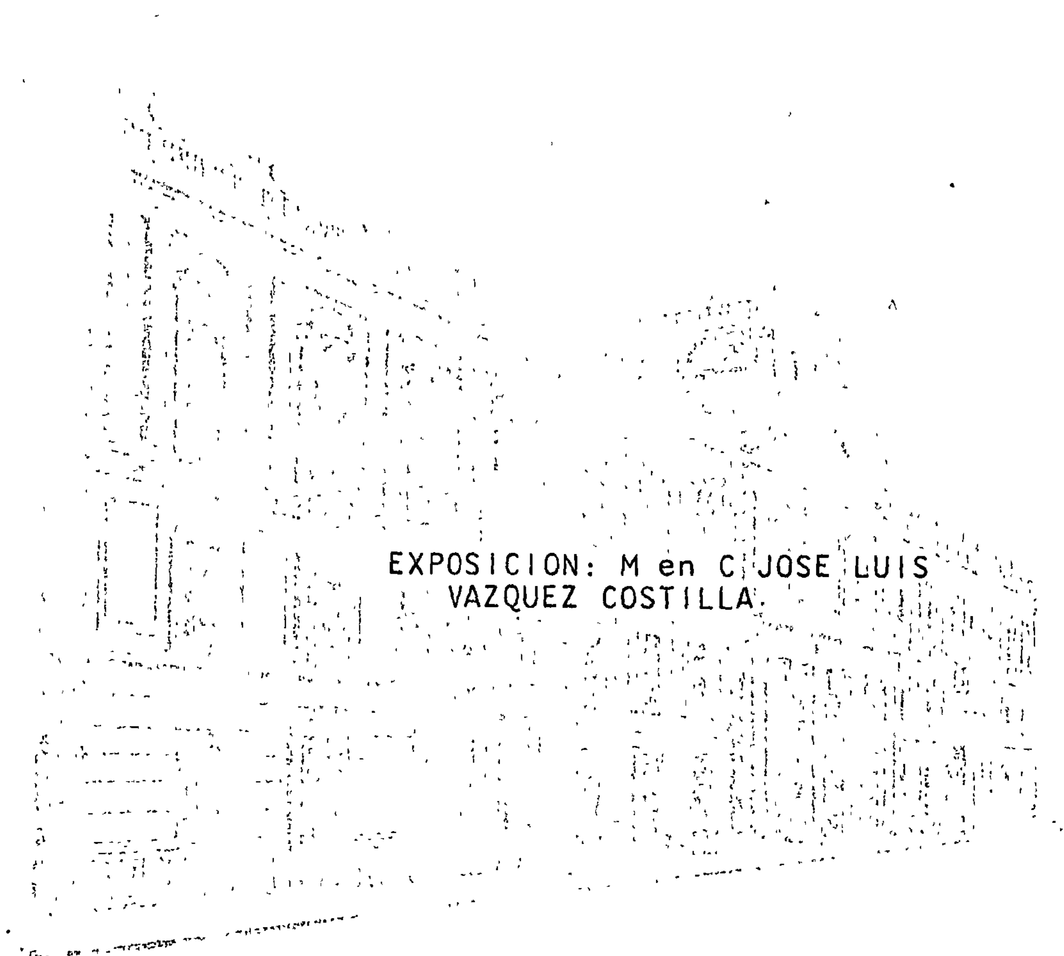
DIRECTORIO DE ASISTENTES AL CURSO DE APLICACION DE MINICOMPUTADORAS
(28 Y 29 DE NOVIEMBRE 5,6,12 Y 13 DE DICIEMBRE DE 1975)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
13.	ARQ. ERNESTO MURRIETA NECOECHEA Cda. Tlalpan 3061 Sta. Ursula Coapa México 22, D. F. Tel: 5-49-23-29	DEPARTAMENTO DEL DISTRITO FEDERAL Delicias No. 36 México 1, D. F. Tel: 5-85-59-77
14.	SR. GUSTAVO A. OROPEZA GARCIA Av. Tasqueña 1247-12 Col. Campestre Churubusco México 21, D. F.	FACULTAD DE INGENIERIA,UNAM Ciudad Universitaria México 20, D. F. Tel: 5-50-00-40
15.	ING. JOSE LUIS RAMIREZ CASTRO Rep. del Brasil 21-304 México 1, D. F.	SECRETARIA DE OBRAS PUBLICAS Av. Fernando No. 268 Col. Alamos México 13, D. F. Tel: 5-90-83-52
16.	ARQ. DAVID RAMIREZ SOUBERVIELLE Calz. Chabacano No. 132 Col. Paulino Navarro México 8, D. F. Tel: 5-30-86-48	DEPARTAMENTO DEL DISTRITO FEDERAL Av. Chapultepec 466-3er. Piso Col. Roma México 7, D. F. Tel: 5-53-48-60
17.	ING. DANIEL GILIBERT Ave. Río Blanco No. 182 Col. Industrial México, D. F. Tel: 5-81-12-50	CONSTRUCTORA GYS, S. A. La Fragua 4-110 México, D. F. Tel: 5-63-82-22

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam



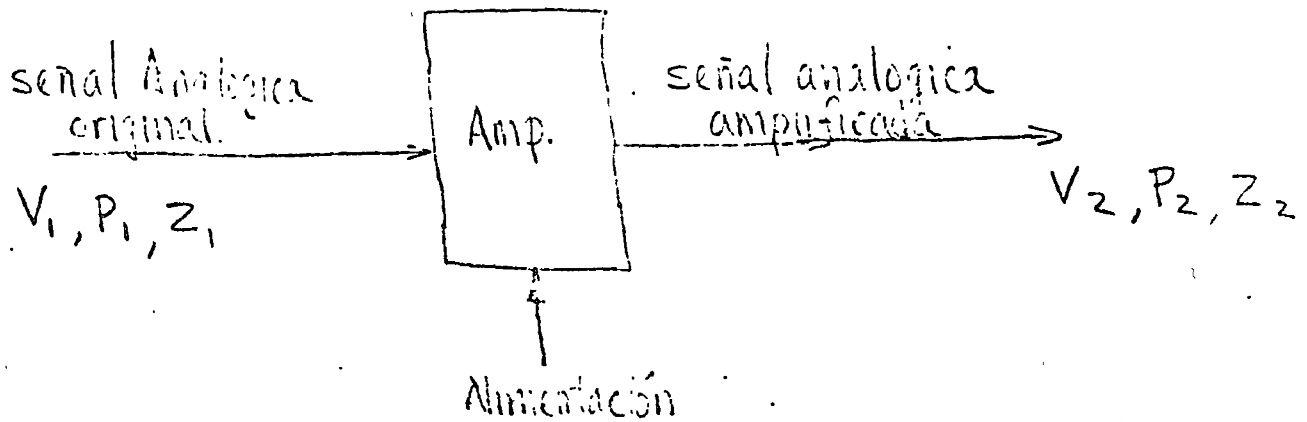
APLICACION DE MINICOMPUTADORAS



EXPOSICION: M en C JOSE LUIS VAZQUEZ COSTILLA

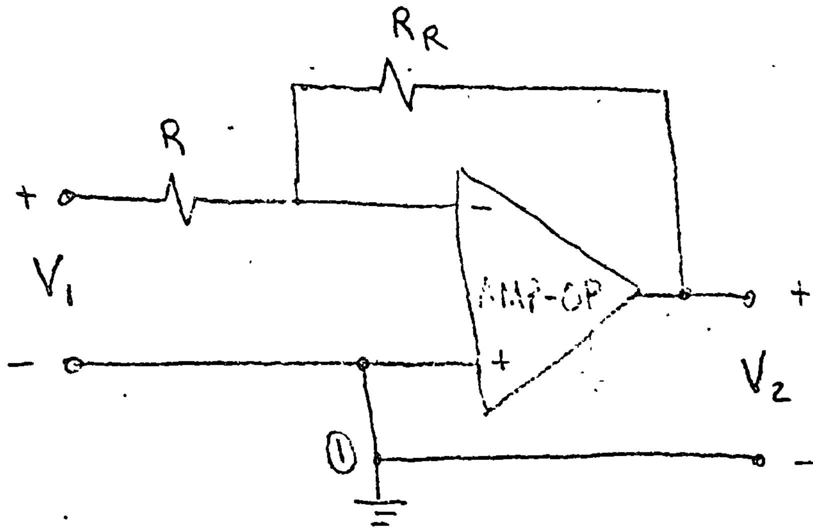
PALACIO DE MINERIA
Tacuba 5, primer piso. México 1, D. F.
TELEFONOS: 513-27-95
512-31-23 521-73-35

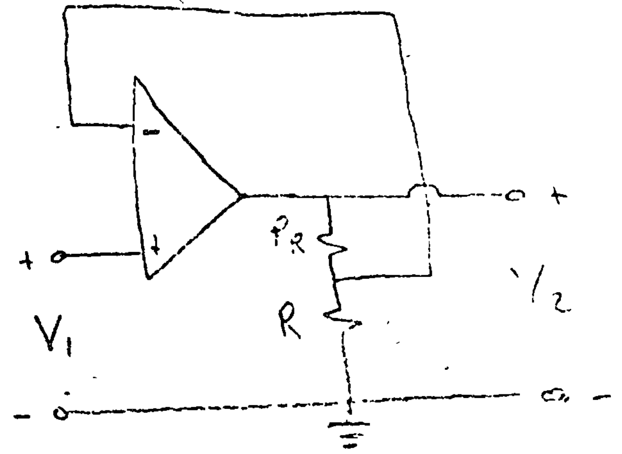
AMPLIFICADORES



$$\left.\begin{array}{l} Z_1 > Z_2 \\ V_1 < V_2 \\ P_1 < P_2 \end{array}\right\} \text{Características usuales}$$

CONFIGURACION BASICA



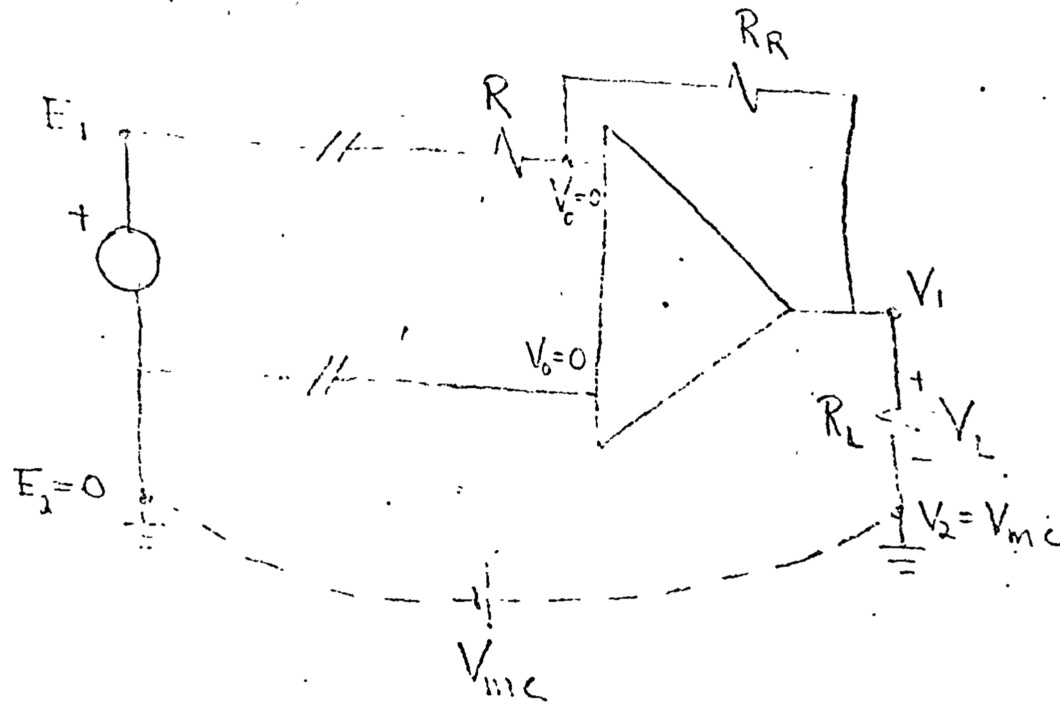


$$V_2 = -\frac{R_R}{R} V_1$$

$$V_2 = \frac{R_R + R}{R} V_1$$

La tierra ①. debe ser la tierra de la fuente del o[illegible]
(Para su justificación ver pp

VOLTAJE DE MODO COMUN



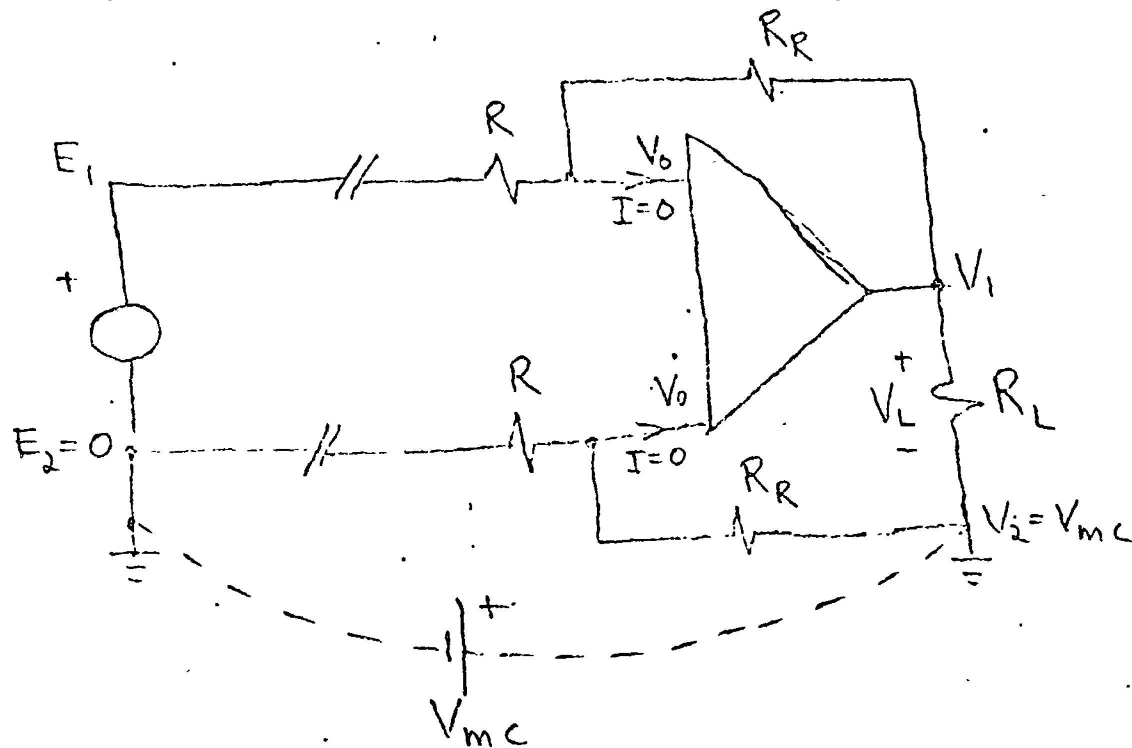
$$\frac{V_1}{R_R} = \frac{-E_1}{R} \quad y \quad V_L = V_1 - V_{mc}$$

$$\Rightarrow \boxed{V_L = -\frac{R_R}{R} E_1 - V_{mc}}$$

V_{mc} aparece como un error en V_L

SOLUCIÓN: AMP. DIFERENCIAL

AMP. DIFERENCIAL



$$\frac{E_1-V_0}{R}=\frac{V_0-V_1}{R_R} \quad y \quad \frac{0-V_0}{R}=\frac{V_0-V_{mc}}{R_R}$$

$$\Rightarrow \quad \frac{E_1}{R}-\frac{V_0}{R}+\frac{V_0}{R}=\frac{V_0}{R_R}-\frac{V_1}{R_R}-\frac{V_0}{R_R}+\frac{V_{mc}}{R_R}$$

$$\Rightarrow \quad \frac{E_1}{R}-\frac{V_{mc}}{R_R}=-\frac{V_1}{R_R}$$ además $V_L=V_1-V_{mc}$ $$\Rightarrow \boxed{V_L=-\frac{R_R}{R}E_1}$$

$$\text{Relación de rechazo al voltaje de modo común} = \frac{\text{Ganancia al voltaje diferencial}}{\text{Ganancia al voltaje de modo común}}$$

(Common mode rejection Ratio: CMRR)

CMRR → Se expresa comunmente en decibeles

$$X \text{ en decibeles} = 20 \log_{10} X$$

...plo

Señal de entrada E_1

$(E_1)_{diferencial} = 1\ V.$

$(E_1)_{Modo\ Común} = 1\ V$

si $CMRR = 80\ db$
$= 10^4$

Señal de salida E_2

$(E_2)_{diferencial} = ?$

$(E_2)_{Modo\ Común} = ?$

solución:

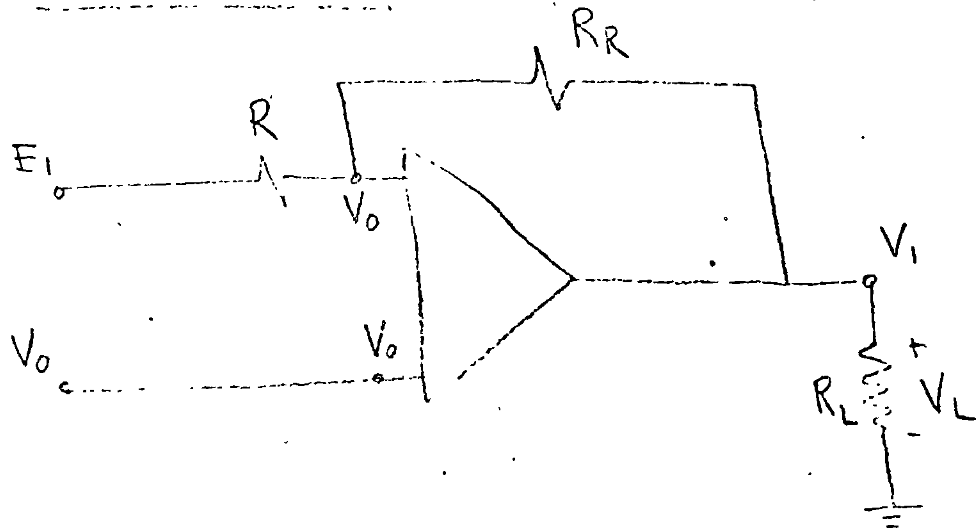
$(E_2)_{dif.} = 1 \times G_{dif.}$

$(E_2)_{m.c.} = 1 \times G_{m.c}$

$$\Rightarrow \frac{(E_2)_{m.c}}{(E_2)_{dif}} = \frac{G_{mc}}{G_{dif.}} = \frac{1}{CMRR.}$$

$$\frac{(E_2)_{m.c}}{(E_2)_{dif}} = \frac{1}{10^4} = 10^{-4}$$

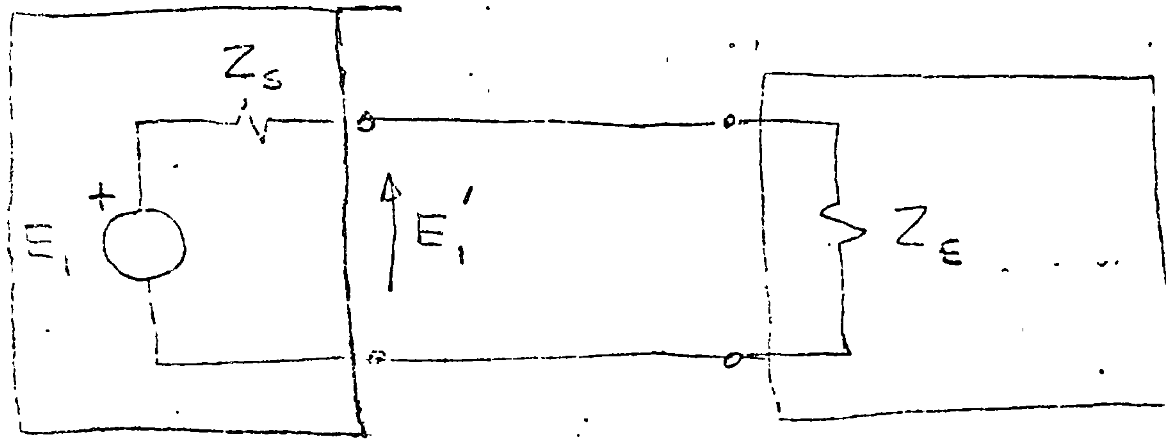
El Amp. Inversor con entrada flotante NO FUNCIONA COMO AMP. DIFERENCIAL



$$\frac{E_1 - V_0}{R} = \frac{V_0 - V_1}{R_R} \quad y \quad V_L = V_1$$

$$\Rightarrow \frac{R_R}{R}(E_1 - V_0) = V_0 - V_L \quad \Rightarrow \quad \boxed{V_L = -\frac{R_R}{R}(E_1 - V_0) + \underbrace{V_0}_{V_{mc.}}}$$

... IAS DE ENTRADA Y SALIDA



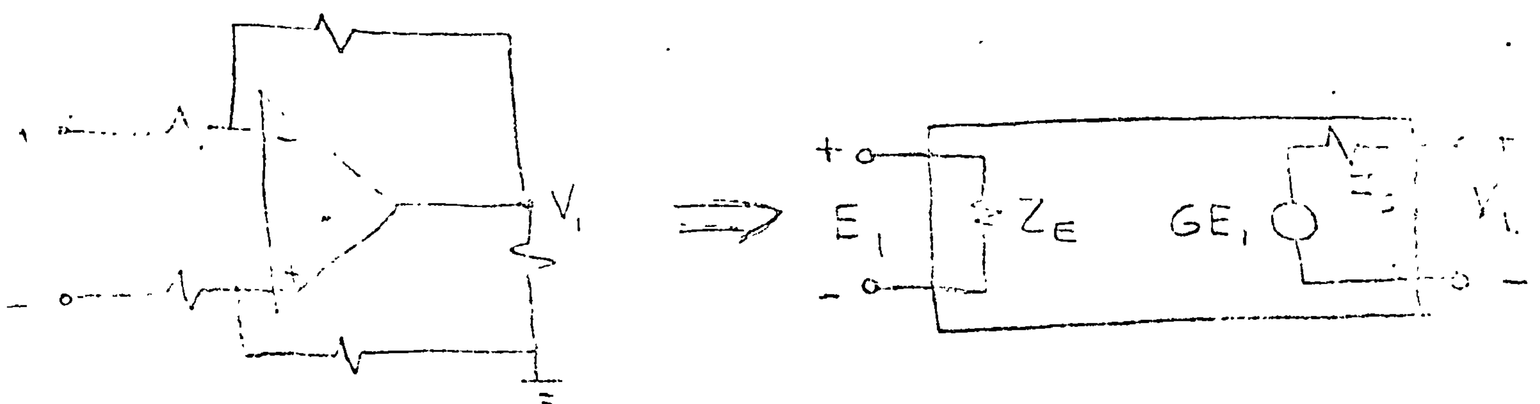
Transductor

- Graficador
- Computadora
- Controlador

$(E_1 - E_1')$ es despreciable $\Longleftrightarrow$ $Z_s \ll Z_E$

$\Rightarrow$

Si $Z_s \not\ll Z_E$ una solución es utilizar un amplificador



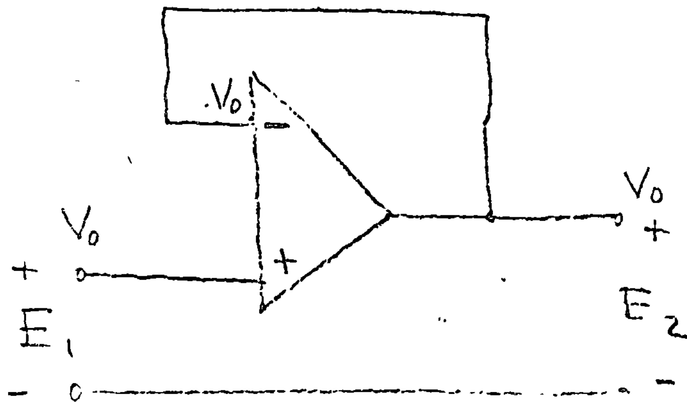
Ejemplo de Valores de Z_E y Z_S

Amplificador Honeywell Accudata 122

$Z_E = 15\ M\Omega$

$Z_S = 1\ \Omega$

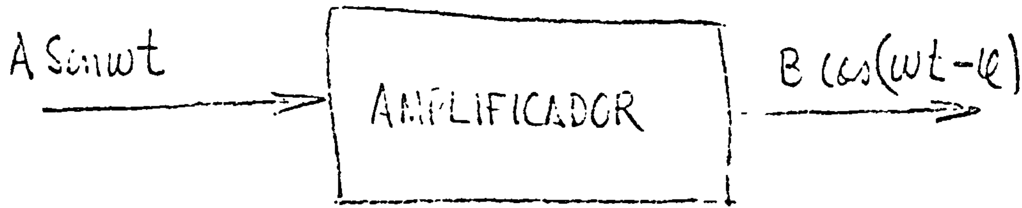
Amplificador transformador de Impedancia

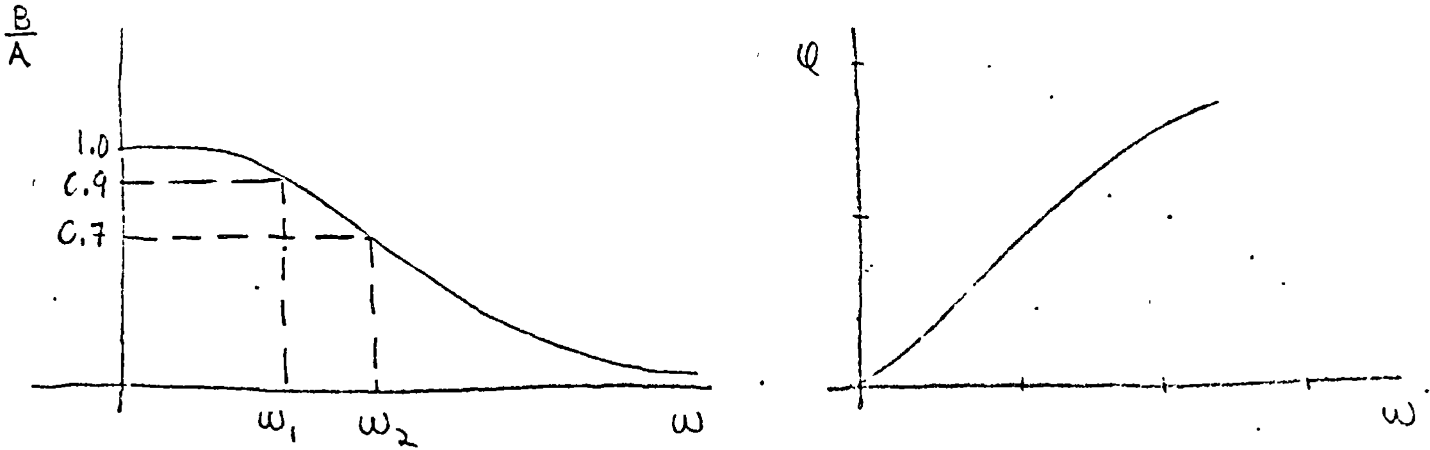


Nótese que $E_1 = E_2$

Pero $Z_E >> Z_S$

RESPUESTA EN FRECUENCIA





GRAFICAS DE RESPUESTA EN FREC. DEL AMP.

ANCHO DE BANDA

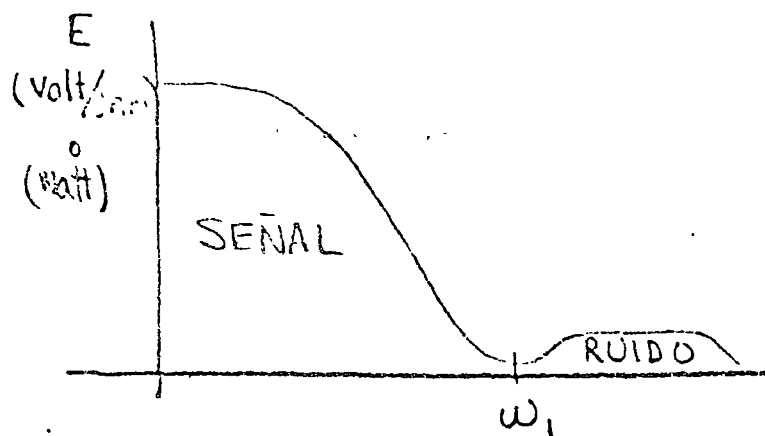
EXISTEN VARIAS CONVENCIONES PARA DEFINIR EL ANCHO DE BANDA DE UN INSTRUMENTO.

a) Ancho de Banda $= \omega\Big|_{B/A=0.9} = \omega_1$

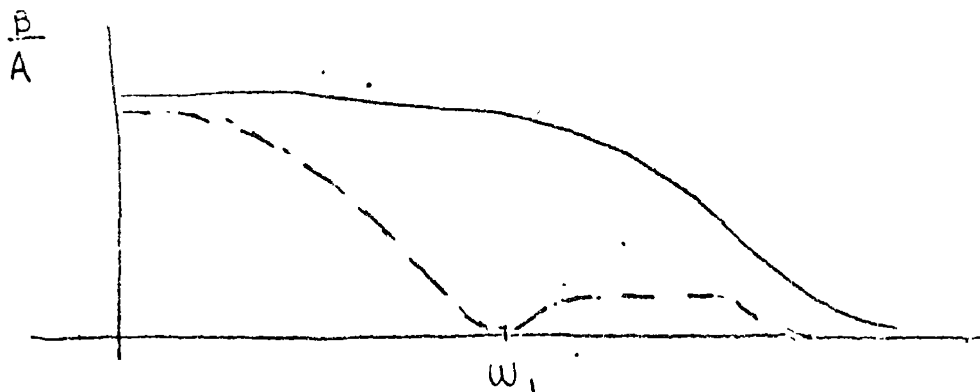
b) Ancho de Banda $= \omega\Big|_{B/A=0.707} = \omega_2$

PREFILTRADO DE SEÑALES

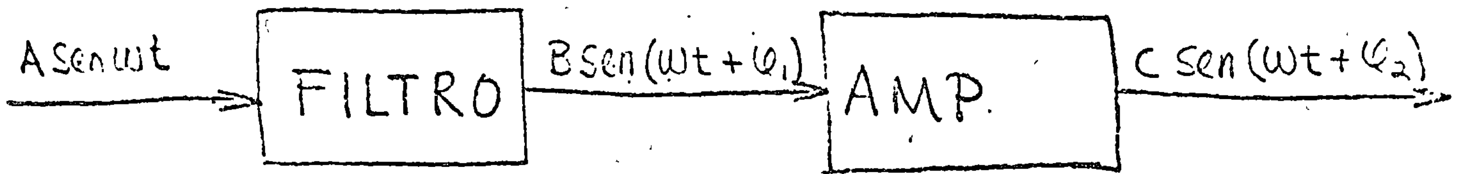
Considere una señal con el siguiente espectro:

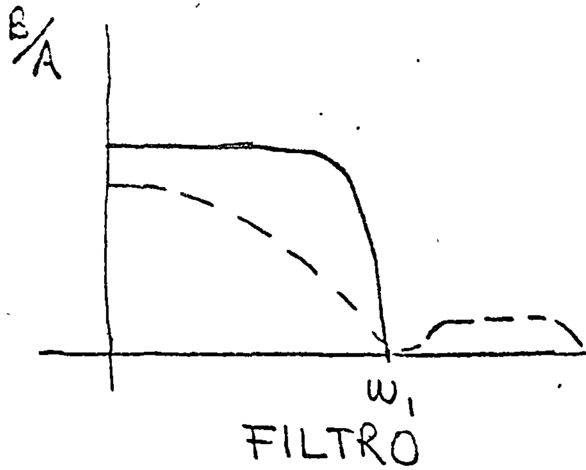


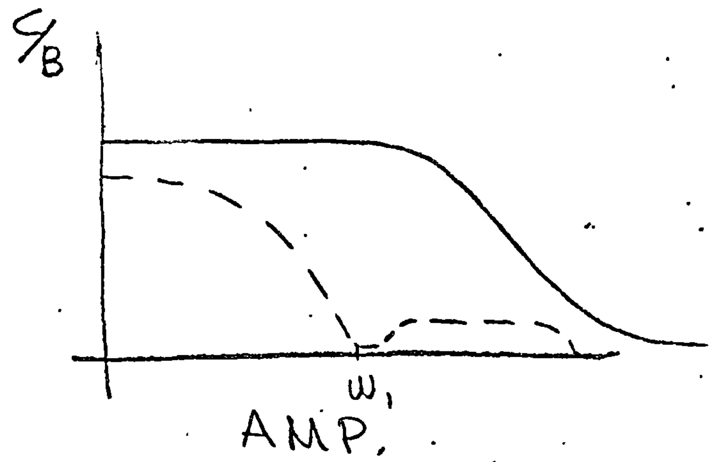
y que se alimenta a un amplificador cuya resp. en frec. es:

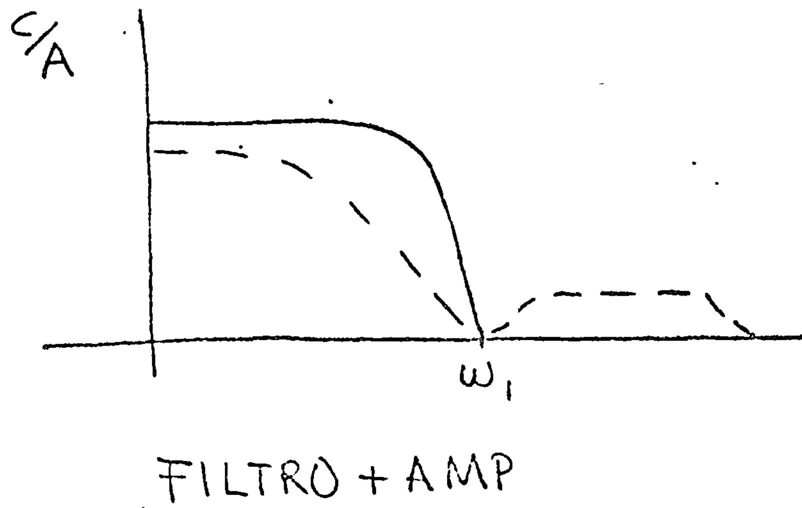


LA SOLUCIÓN PARA NO AMPLIFICAR EL RUIDO ES: PREFILTRAR.



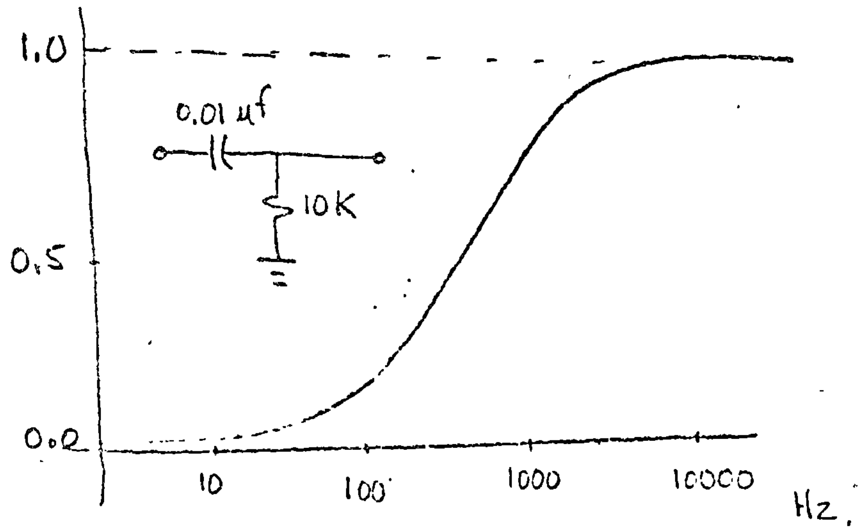


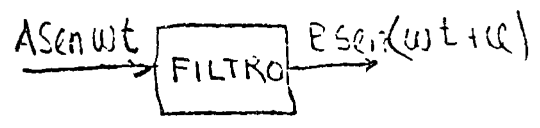




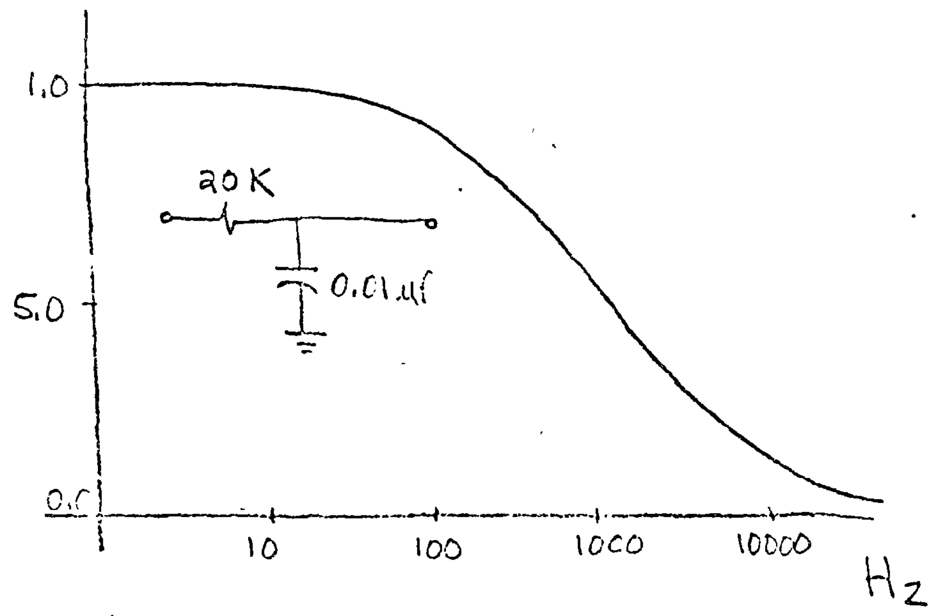
FILTRO + AMP

ALGUNOS FILTROS

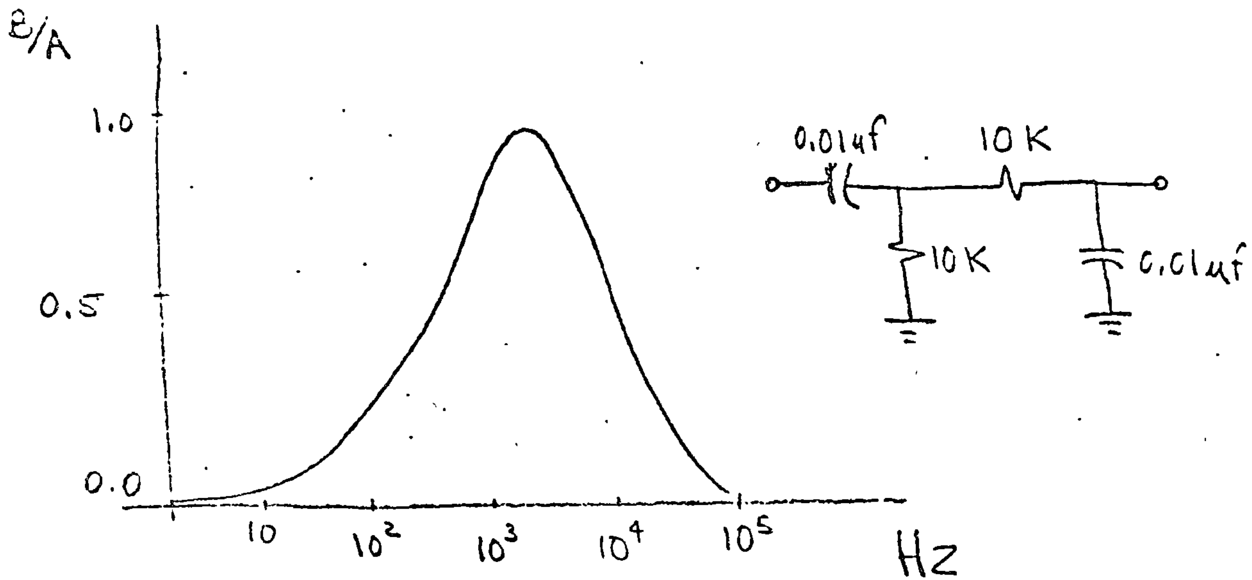




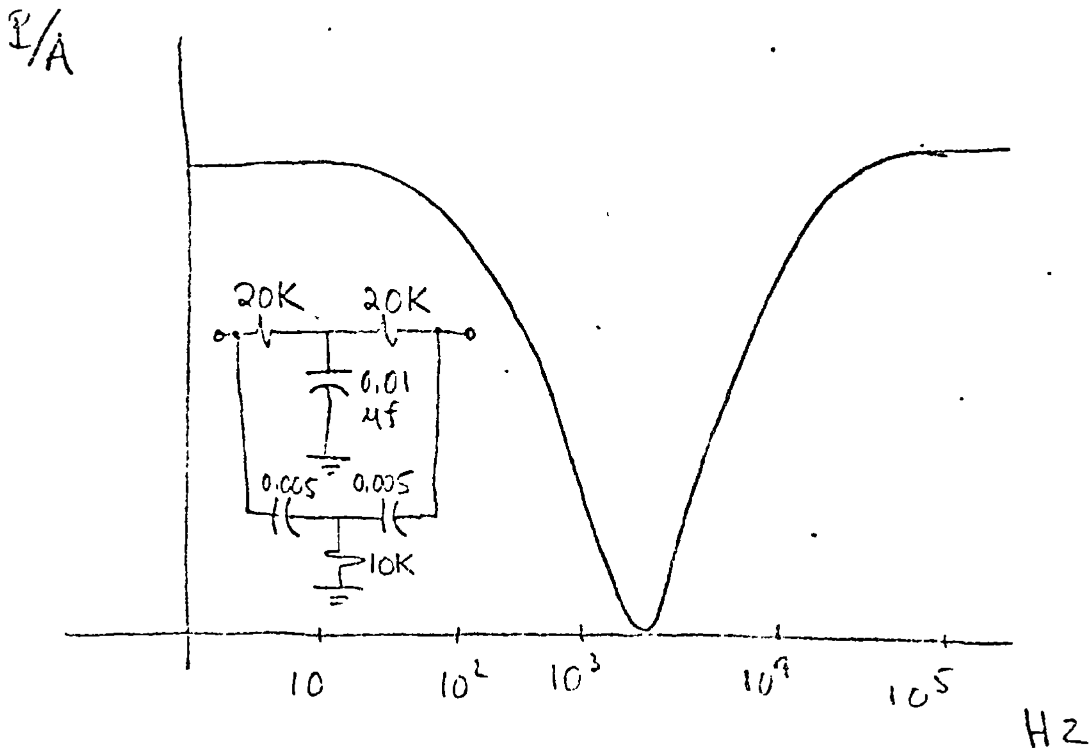
FILTRO PASA-ALTAS



FILTRO PASA-BAJAS

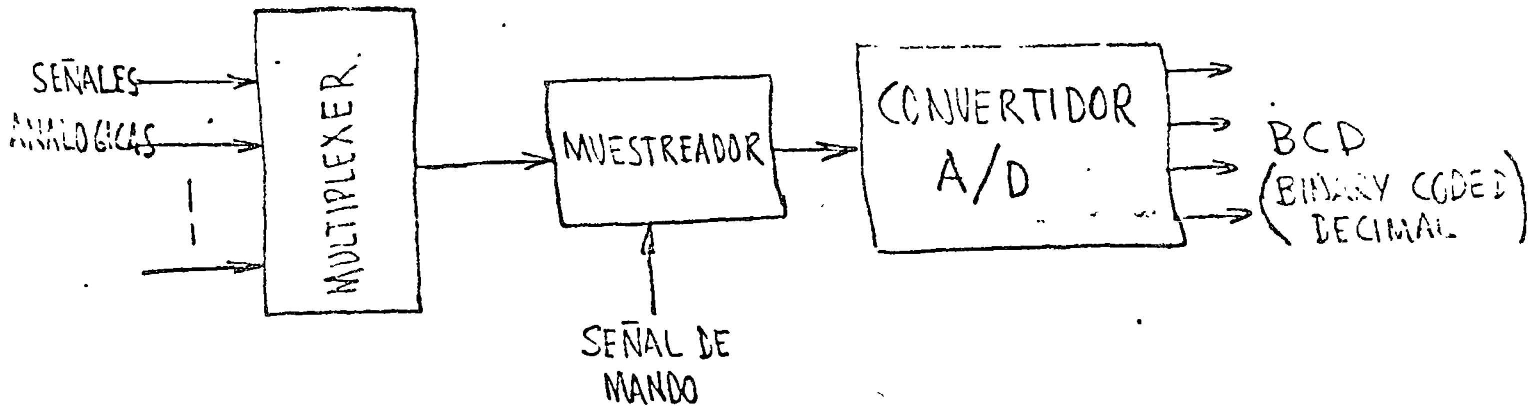


FILTRO PASA-BANDA



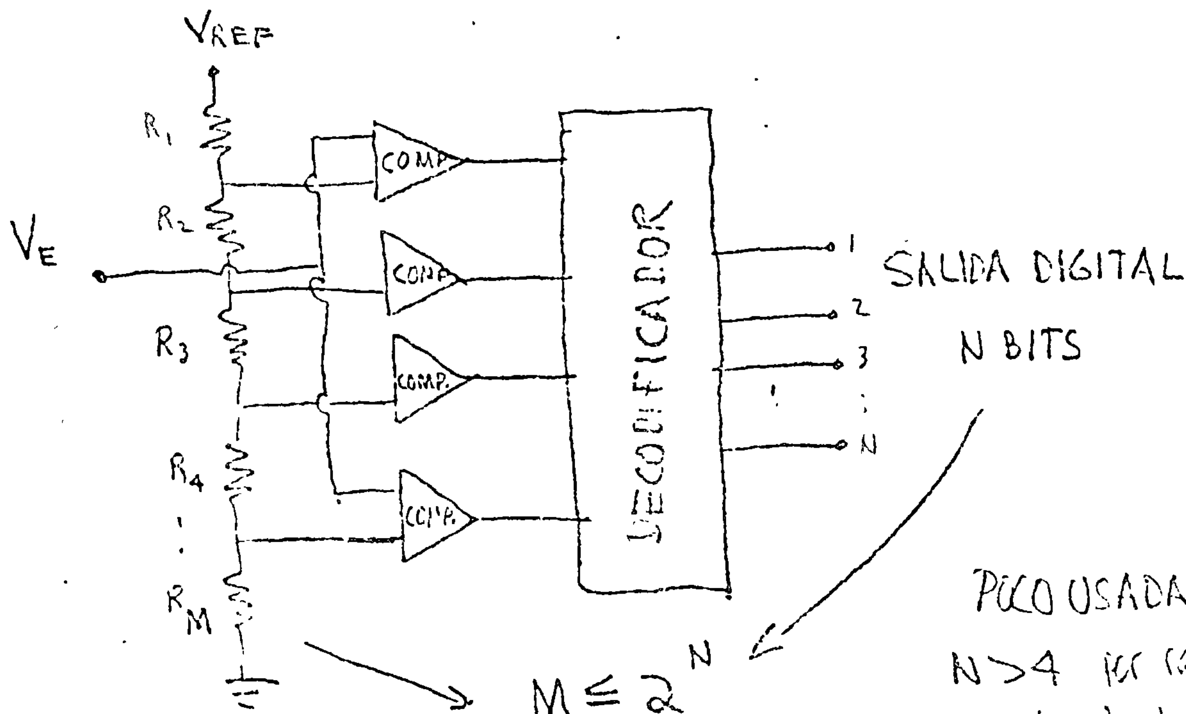
FILTRO NOTCH

CONVERSION ANALOGICA DIGITAL



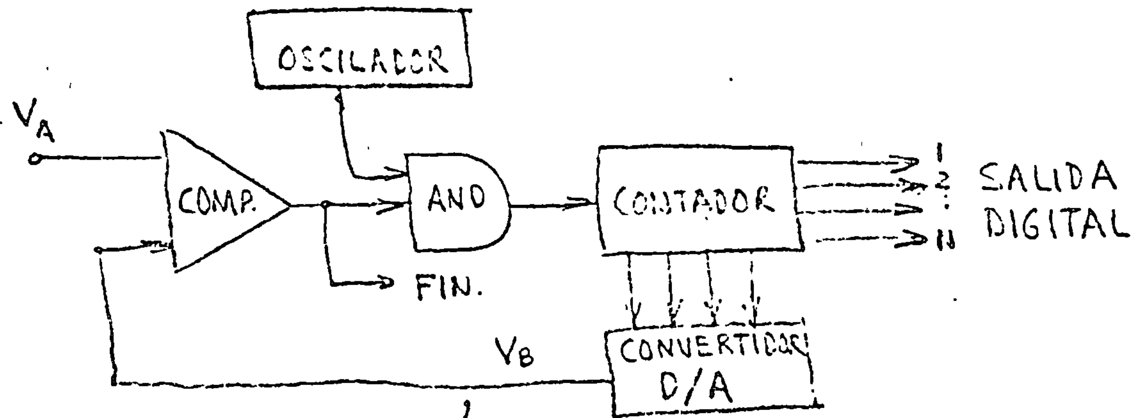
TECNICAS DE CONVERSION

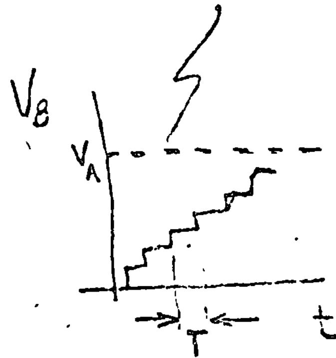
CONVERTIDOR EN PARALELO :



POCO USADA PARA
N>4 por razones de [illegible]
y complejidad [illegible]

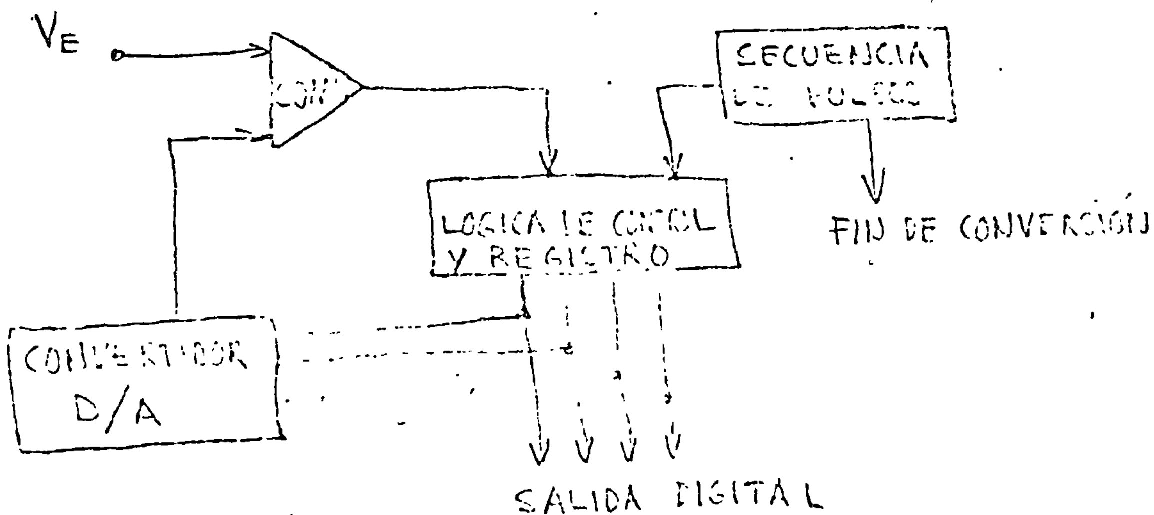
CONVERTIDOR DE CONTEO



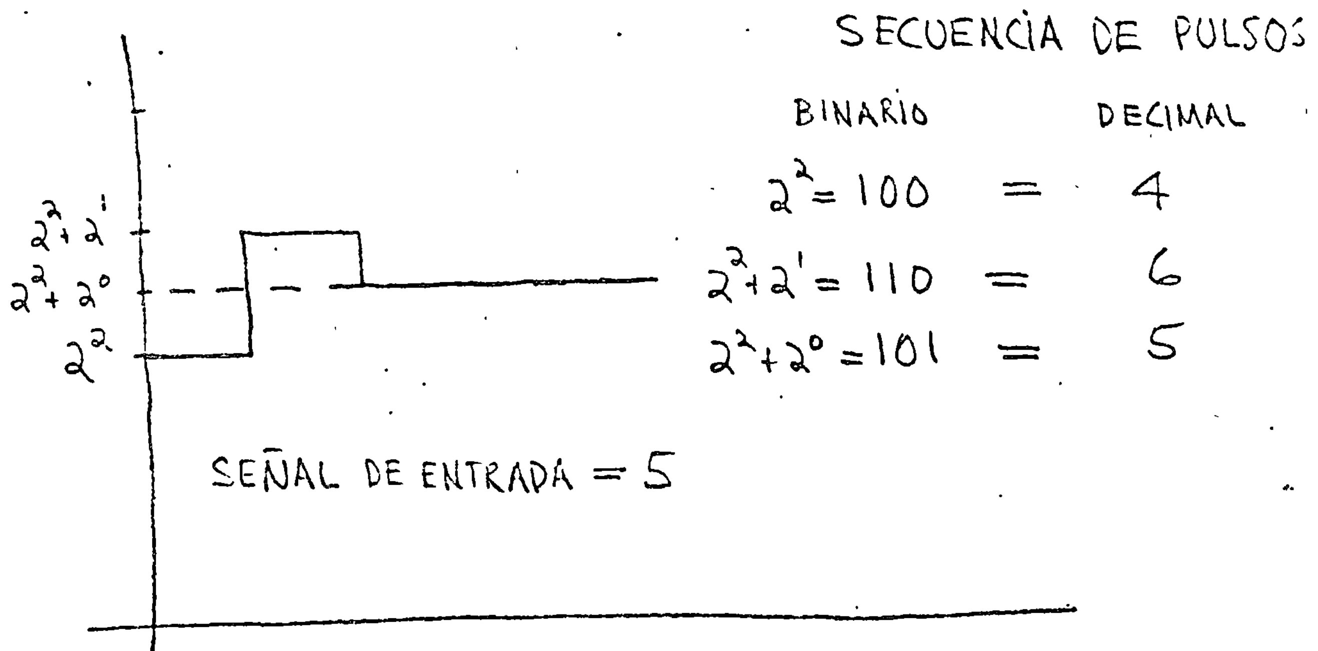


$$V_A\big|_{max} = 2^N \quad \therefore \text{ tiempo maximo de conversion} = 2^N T$$

CONVERTIDOR DE APROXIMACIONES SUCESIVAS



PROCESO DE CONVERSION POR APROX. SUCESIVAS EN UN CONVERTIDOR DE 3 BITS.



CARACTERISTICAS DE LOS CONVERTIDORES TIPO CONTEO Y APROX SUCESIVA

- SU EXACTITUD DEPENDE PPALMENTE DE LAS CARACTERISTICAS DEL CONVERTIDOR D/A.

CONVERTIDORES

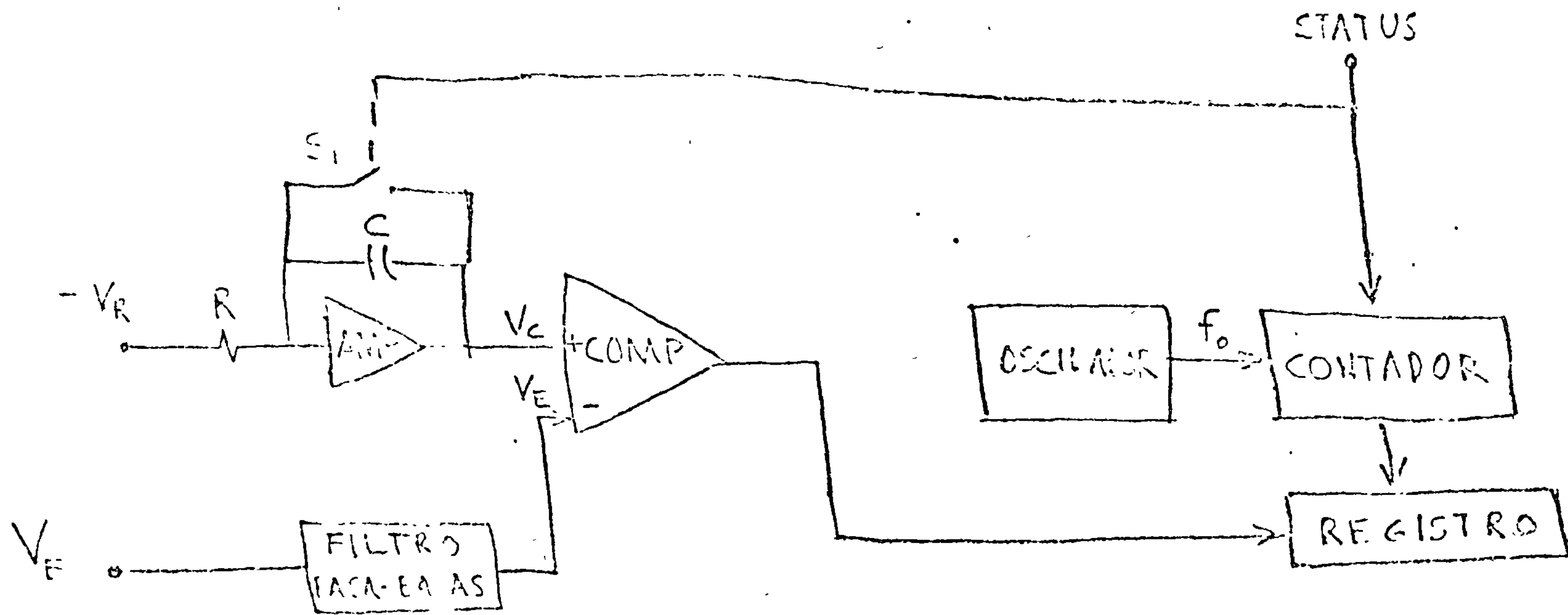
CARACTERISTICAS DEL CONV. A/D TIPO RAMPA

- SIMPLICIDAD
- EXACTITUD DE 0.1 a 1% DEBIDO PPALMENTE A VARIACIONES EN C, R y f_0 OCACIONADAS POR CAMBIOS DE TEMPERATURA, ENVEJECIMIENTO Y VOLTAJE DE ALIMENTACION.

CARACTERISTICAS DEL CONV. A/D TIPO DOBLE RAMPA

- SIMPLICIDAD
- INMUNE A VARIACIONES EN C, R, f_0
- INMUNE A RUIDO DE LINEA (Normalmente 60 Hz)

CONVERTIDOR A/D TIPO RAMPA



AL INICIAR EL CONTEO:

a) EL CONTADOR SE INICIA EN CERO

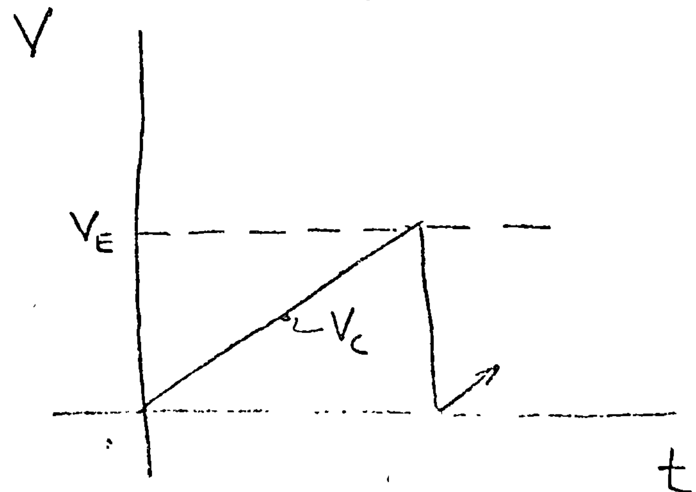
b) S_1 ABRE

CUANDO $V_c = V_E$:

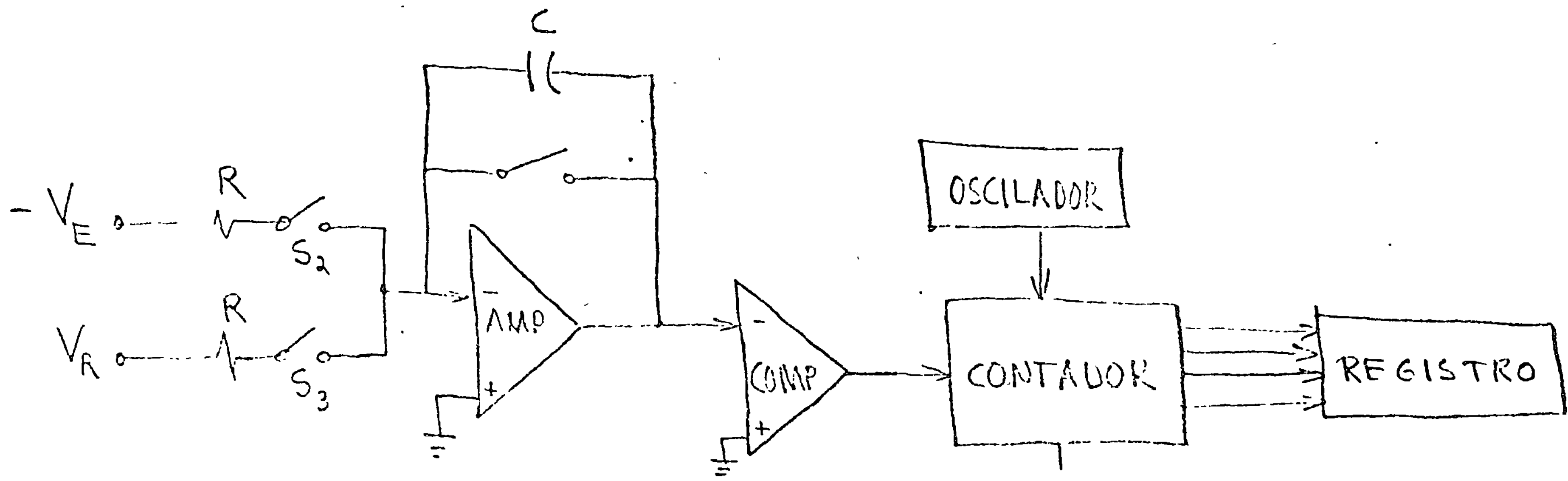
a) EL REGISTRO TOMA LA INFORMACIÓN DEL CONTADOR

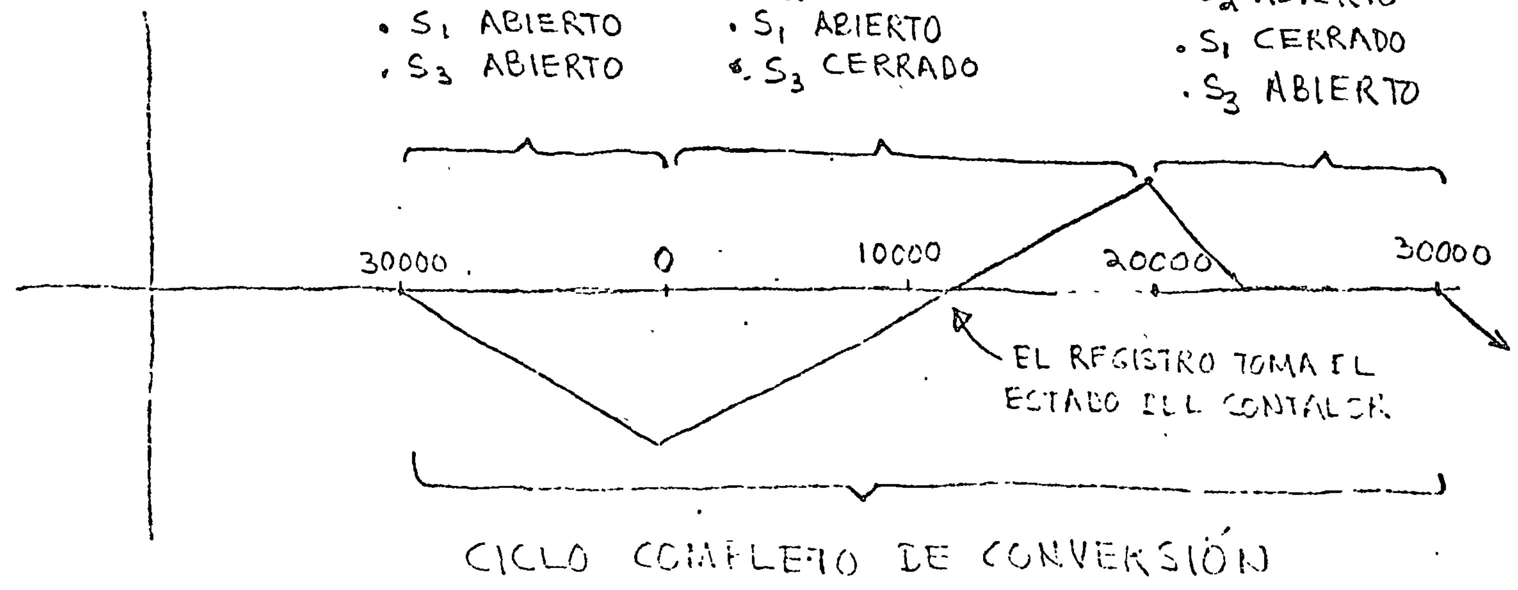
b) CIERRA $S_1 \Rightarrow V_c \to 0$

c) FIN DEL CICLO



CONVERTIDOR A/D TIPO DOBLE RAMPA





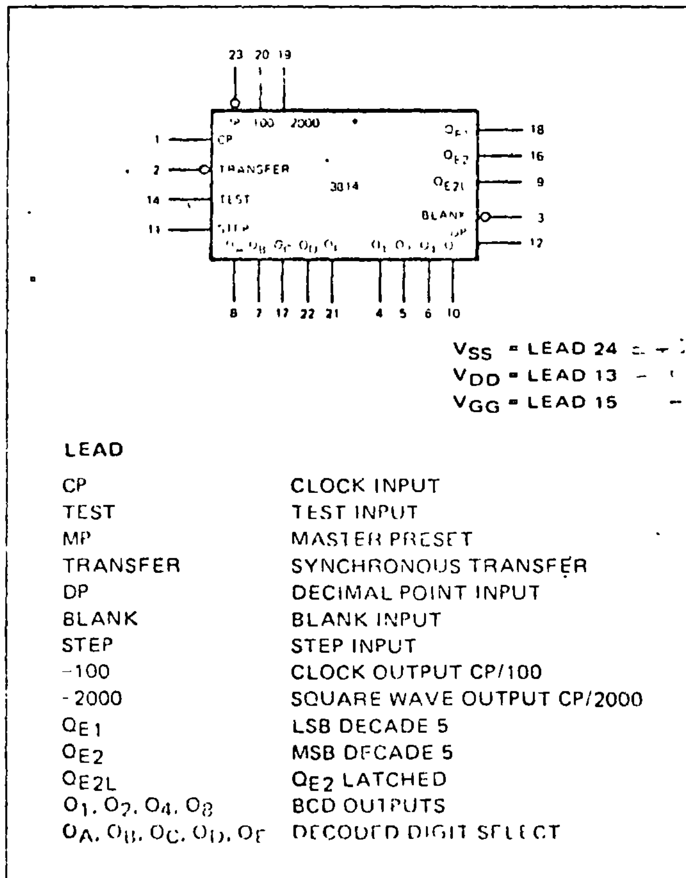
CICLO COMPLETO DE CONVERSIÓN

... DIGITAL VOLTMETER LOGIC ARRAY

R Whicker and [illegible] Trueblood

INTRODUCTION

Fairchild Semiconductor has developed a silicon gate device that contains most of the logic required for a 4 1/2 decade digital voltmeter. All necessary BCD counters, latches and the display multiplexing logic is on chip. In addition, the control signals necessary for dual slope integration techniques are generated by the 3814 device. The BCD outputs can directly drive a 9315 BCD to 1-of-10 decoder or 9307 BCD to 7-segment decoder. Zero suppression is generated on chip by feeding back the DIGIT SELECT output. Outputs are also provided for indicating over range and under range. A unique feature of the 3814 is the incorporation of a 10-count pause at the start of an integration cycle to mask noise generated when switching the external analog circuits, such as the reference current source.



LEAD	
CP	CLOCK INPUT
TEST	TEST INPUT
MP	MASTER PRESET
TRANSFER	SYNCHRONOUS TRANSFER
DP	DECIMAL POINT INPUT
BLANK	BLANK INPUT
STEP	STEP INPUT
÷100	CLOCK OUTPUT CP/100
÷2000	SQUARE WAVE OUTPUT CP/2000
O_{E1}	LSB DECADE 5
O_{E2}	MSB DECADE 5
O_{E2L}	O_{E2} LATCHED
O_1, O_2, O_4, O_8	BCD OUTPUTS
O_A, O_B, O_C, O_D, O_E	DECODED DIGIT SELECT

Fig 1. Logic symbol and lead designation

DUAL SLOPE INTEGRATION

A reliable and accurate DVM circuit must be insensitive to long term changes of such factors as supply voltage, time base, and passive and active component values. For the short term, it must be able to reject 60 Hz line perturbations. Dual Slope Integration achieves a high degree of accuracy by causing the effect of changes in these parameters to cancel.

One method of Dual Slope Integration involves integrating a current directly related to the unknown voltage (V_x) for a fixed period of time, followed by the integration of a standard current (I_S) until the integrator output returns to zero. The amount of time required to null the integrator is directly proportional to the ratio of I_x to I_S and therefore to V_x. Since the same system power supply, time base and components are used for integrating the known and unknown currents their absolute values are not extremely critical.

3814 BLOCK DIAGRAM

The 3814 provides 4 1/2 decades of BCD counters, with a modulus of 40,000, clocked by the CP input. (Figure 2) The CP input is TTL compatible and requires a low time of 300 ns minimum and a high time of 500 ns minimum. The counters change state on the low to-high CP transition.

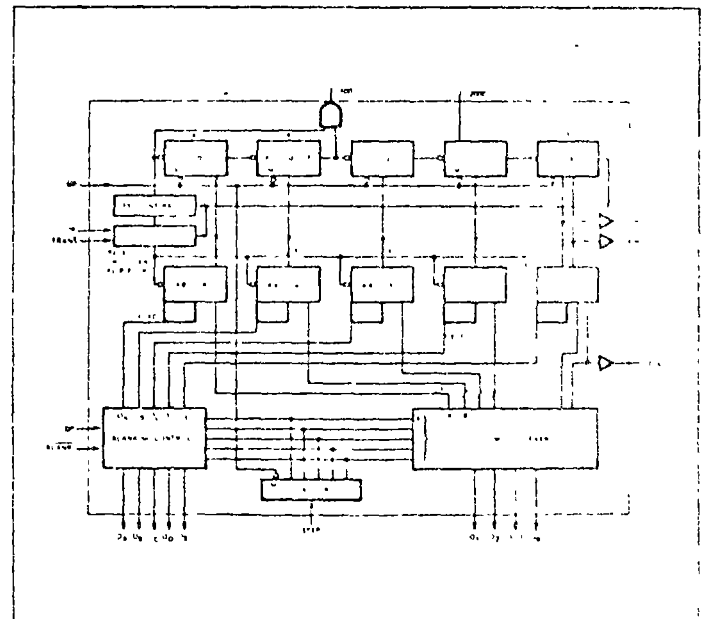


Fig 2 Block diagram

Fairchild cannot assume responsibility for use of any circuitry described other than circuitry entirely embodied in a Fairchild product. No other circuit pa[illegible] licenses [illegible]

The second decade output is gated with the clock and brought off chip to provide a clock pulse output for every 100 input clocks ("÷100 output"). The output of the first flip-flop of the thousands decade is buffered and brought directly off chip as "÷2000". The outputs of both flip-flops of the fifth "half decade" are available as outputs Q_{E1} and Q_{E2}. The latched state of the most significant bit (Q_{E2L}) is also brought off chip.

A TRANSFER input causes the data in the counters to be stored in the latches. This input is edge sensitive and is synchronized internally with the clock. When the TRANSFER input changes from high to low the circuit is enabled, so that when CP goes low, data is stored. A TRANSFER command is permitted only once during a count cycle by an internal flip-flop which is set when the first transfer occurs, and remains set until the next terminal count (counters advancing from 39,999 to 00,000). Transfers are ignored when this flip-flop is set.

The latched state of each decade is multiplexed out as BCD data on outputs O_1, O_2, O_4 and O_8. The multiplexer is driven by a scanner counter with outputs O_A, O_B, O_C, O_D and O_E available. This counter is clocked by the STEP input causing the stored data to appear, decade by decade, on the O_1, O_2, O_4 and O_8 outputs. One of the O_A through O_E outputs will be high indicating the decade displayed. The BLANK input (active low) causes all five of these outputs to go low. Because outputs O_A through O_E can drive display lamps in a multiplexed system the display will blank when they are low.

FUNCTIONAL DESCRIPTION

Figure 3 is a simplified schematic of a digital voltmeter using a 3814 with an integrator and comparator front-end. Figure 4 shows the output waveforms of the integrator and comparator for a typical cycle when some unknown voltage is applied to the input of the integrator (V_x). During the time interval from 30,000 to 39,999, a current I_x ($\approx V_x/R_x$) is integrated. When the counter reaches 40,000 (or 00,000), the input to the integrator is switched to integrate the standard current I_S. Integration of the standard current continues until the integrator crosses the null point. At this time, the comparator output switches states to a LOW and this signal is used to cause a transfer of the count into the latches in the 3814. The stored count is then directly proportional to the ratio I_x/I_S. When the count reaches 20,000, the analog circuitry can be reset to zero.

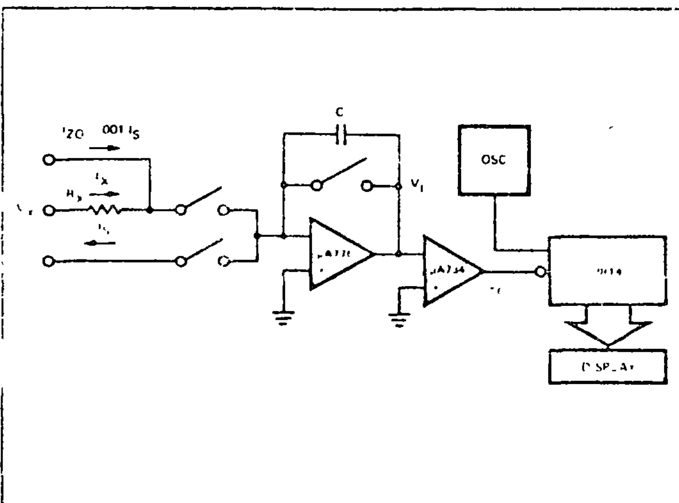


Fig. 3. Simplified schematic of front end.

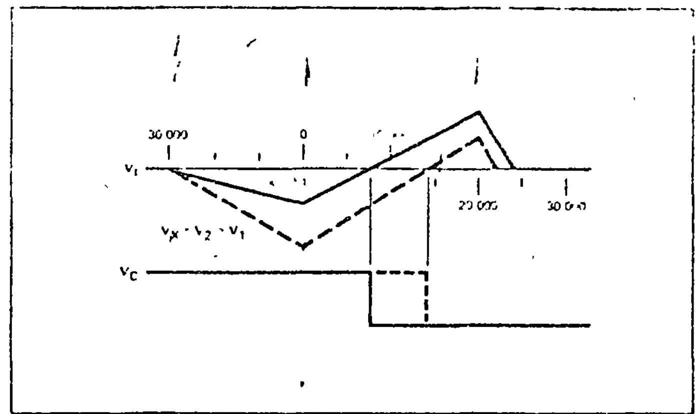


Fig. 4. Integrator and comparator waveforms.

A glitch may occur in the null detector output when it changes state or during an analog reset. To prevent false loading of a count into the storage latches, the transfer input is accepted only once during the interval 00,000 to 39,999. Any transfer commands after the first one are ignored.

The control for the various periods of integration can be obtained from the Q_{E1} and Q_{E2} outputs of the 3814. A "11" state means the unknown should be integrated, an "XO" means the standard should be integrated and a "01" means the reset period is present. (The circuitry need not be reset here if a DVM capable of displaying 0, 1 or 2 in the most significant decade is desired. In this case, the analog signals must be rapidly reset at count = 30,000.) If only a 0 or a 1 in the most significant decade is required, a transfer occuring during the interval 20,000 to 30,000 indicates overflow. This can be detected by the Q_{E2L} output, which will be high if a count greater than 19,999 was stored in the latches. This can be used to indicate overflow, to automatically up-range, and to blank the display by means of the BLANK input. Forcing MASTER PRESET (MP) sets the internal counters to 30,000 on the next clock pulse.

The ÷2000 output can be used to down range in an autoranging instrument. If this output has not yet gone high when the transfer is received, then the count is less than 10% of full scale (count <1000).

The data outputs are designed to drive a multiplexed display system. A single decoder/driver (such as the 9315 BCD to 1-of-10, or the 9327 BCD to 7-segment) is connected to the O_1, O_2, O_4 and O_8 outputs. The outputs of the decoder drive *all* the display devices. The O_A, O_B, O_C, O_D and O_E outputs drive transistors which select the desired digit. The BCD will have one quarter unit load left over which can be used to drive a low power TTL latch (93L00) or register if parallel BCD data is desired as an additional output of the DVM.

The STEP input, which drives the output multiplexer, can be driven directly from the ÷100 output or can be driven by a separate oscillator.

ZERO OFFSET

An analog glitch also occurs when the integrator input is switched from I_x to I_S. If the input voltage is near zero, the integrator output is close to the null line at the end of the integration. The glitch might cause false triggering of the null detector. For this reason I_{ZO} adds a quantity of charge equal to ten counts of I_S during the integration of the unknown voltage V_x as shown in Figure 3 and Figure 5. This fixed offset guarantees that the integrator output moves away

from the null even if the input voltage is zero. When the counter on the 3814 reaches 40,000 (00,000) the counter remains at zero for ten counts, thus subtracting out the extra ten units of current added during the integration of the unknown voltage.

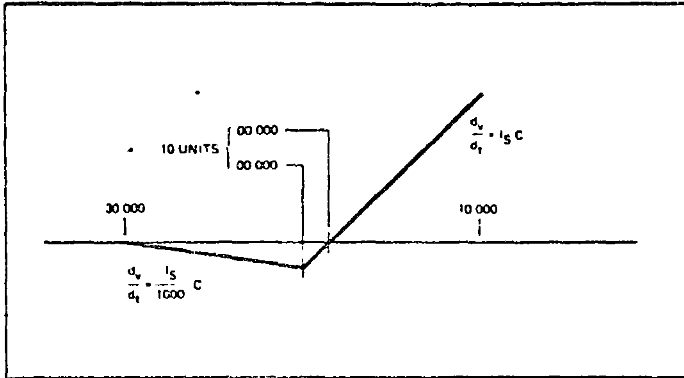


Fig. 5. Integrator output when $V_X = 0$ showing desired zero offset.

LINE REJECTION

Line rejection is important for measuring voltages from high impedance sources. As shown in Figure 6, any 60 Hz components contained in the unknown voltage have an average voltage of zero only if the sample period is some harmonic of 60 Hz. Since the sample period of the 3814 is 10,000 clocks, a suitable clock frequency would be 600,000 Hz for good line rejection. This would give a sample rate of 15 Hz and an output multiplex rate of 6 kHz if the ÷100 output is used as the STEP input.

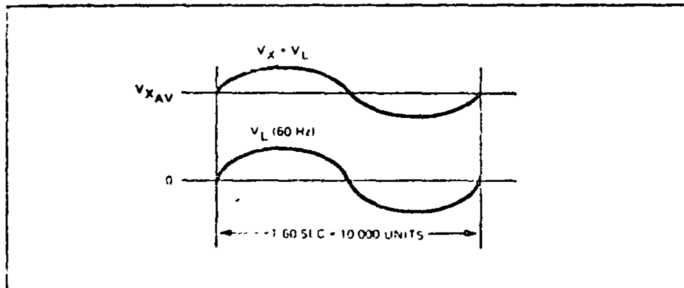


Fig. 6 Line rejection

DECIMAL POINT

The position of the decimal point in the display is selected by an external switch. For zero suppression, the DP input of the 3814 must be driven by one of the five DIGIT SELECT outputs. This feedback inhibits the zero suppression for that decade and all remaining decades to the right. If the DP input is tied to V_{SS}, all digits are displayed. Tying DP to V_{DD} has the same effect as tying it to output O_E. The different possibilities are shown in Figure 7.

DIGIT FED BACK	COUNT	DISPLAY *
A, or DP - V_{SS}	00000	0
A, or DP - V_{SS}	00120	120
B	00120	12 0
C	00120	1.20
D	00120	0 120
E, or DP = V_{DD}	00120	0.0120
		E DCBA

*The decimal point in the display is not controlled by the 3814

Fig 7. Decimal point positioning.

EXPANSION TO 5 1/2 DECADES

To increase accuracy, additional decades can be added using standard MSI components. Figure 8 shows the extra devices needed to realize a 5 1/2 digit system.

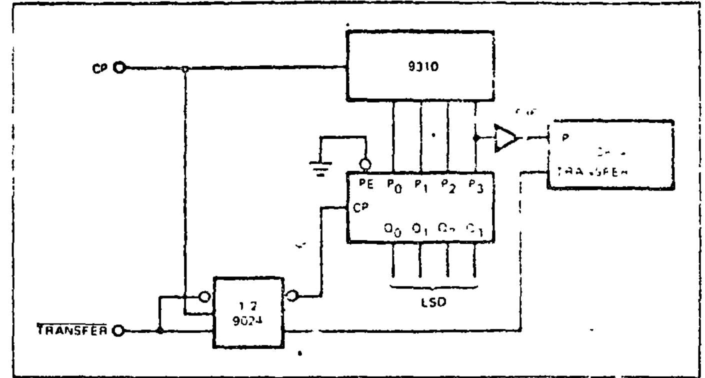


Fig. 8. Modification for 5-1/2 decades.

UNDER RANGE INDICATOR

For input voltages less than 10% of full scale an under range signal can be generated by gating O_{E1}, O_{E2}, the ÷2000 output and the analog comparator output. Figure 9 shows circuitry required for a constant output which can be used to turn on a Light Emitting Diode or as a control signal for auto-ranging circuitry. A pulsed output is shown in Figure 10.

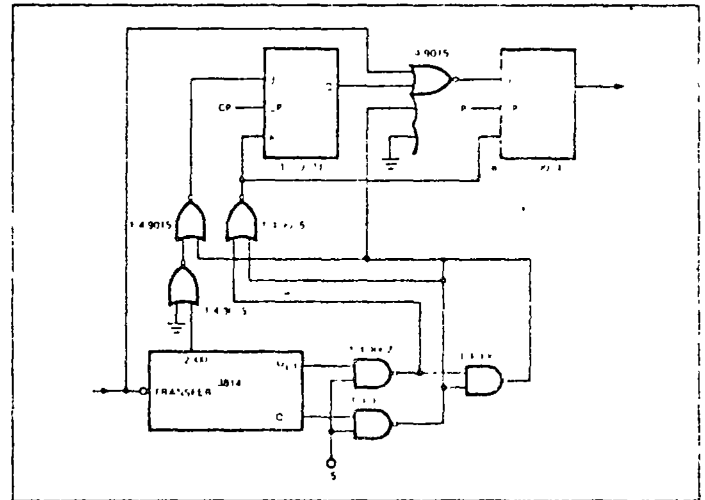


Fig. 9. Under range indicator.

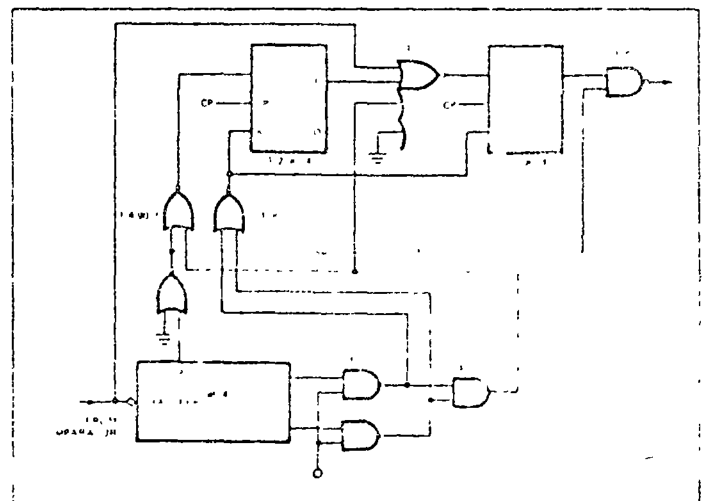


Fig. 10. Blinking under range indicator.

BLANKING BETWEEN DIGITS

While the outputs are multiplexed to the various digits, interference may appear depending on the type of display and the multiplex rate. Figure 11 shows a self-driven blanking scheme to assure that the segment select inputs are stable prior to digit select.

GAS DISCHARGE DISPLAY

The BCD outputs of the 3814 will directly drive the 9315 1-of-10 decoder for systems using NIXIE* display tubes. Figure 12 shows the circuit required. For detailed information, see Fairchild Application Note 212 on digital display systems.

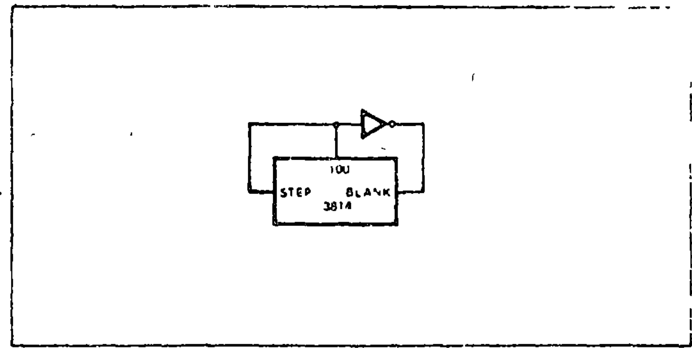


Fig. 11. Self-driven multiplexer with blanking between digits.

Fig. 12 NIXIE* output.

*NIXIE is a registered trade

LOW COST DVM

Figure 13 shows one version of the basic DVM shown in Figure 3. An input buffer, such as μA776, could have been added to boost the input resistance to 400 MΩ and provide isolation of the unknown from the action of the current sources. If source resistance is low, the buffer may not be needed. The I_{ZO} and I_S current sources have been implemented with discrete components. Also, temperature compensation has been added to the I_S circuit as this is most critical to system accuracy. As designed, only positive inputs are properly integrated. If negative input capability is also desired, additional current sources and gating are needed.

Ideally, SW_1 and SW_2 have zero resistance when on, infinite resistance when off and no offset voltage. For an accurate system then, bipolar transistors cannot be used because of offset. P-Channel or N-Channel FETs ably satisfy all three of the switch criteria. To avoid gate-to-source debiasing, P-Channel devices should be used for negative input voltages and N-Channel devices for positive inputs. The versatile μA776 is used again as the integrator amplifier, the μA734 comparator receives the integrator signal and upon a null crossing, generates the transfer command to the 3814. All gating for mode control to SW_1, SW_2 and SW_3 is obtained from the Q_{E1} and Q_{E2} output. The FND21 LED Display Module and the associated decoder and drivers are also shown. For flexibility of decimal point location and zero suppression, a five position SPST switch has been added to appropriately gate the DIGIT SELECT outputs to the DP inputs.

This DVM can be built with a total of *only six integrated circuits, seven* if input buffering is required.

Fig. 13

AMPLIFICADORES, CONVERTIDORES A/D y D/A
BIBLIOGRAFIA

BANNON.- OPERATIONAL AMPS.: THEORY & SERVICING.
RESTON PUBLISHING CO, 1975.

HONEYWELL.- ACCUDATA 122 D.C. AMP. TECHNICAL MANUAL 1975

FINKEL JULES.- INTERFACING TO MINICOMPUTERS.
WILEY.- 1975

KORN.- MINICOMPUTERS FOR ENGINEERS & SCIENTISTS
McGRAW-HILL 1973

PEATMAN.- THE DESIGN OF DIGITAL SYSTEMS
McGRAW-HILL 1972

MALMSTADT-ENKE-CROUCH.- DIGITAL & ANALOG DATA CONVERSIONS
W.A. BENJAMIN INC. 1973

KOHONEN.- DIGITAL CIRCUITS AND DEVICES
PRENTICE HALL 1972