DIRECTORIO DE PROFESORES DEL CURSO INTRODUCCION A LAS MINICOMPUTADORAS 1980.

1. ING. LUIS CORDERO BORBOA
Jefe de Laboratorio de Computación
Facultad de Ingeniería
U N A M
México 20, D.F.
Tel. 550.52.15 Ext. 4750

2. DR. VICTOR GEREZ GREISER
Shakespeare No. 6-5° Piso
México, D.F.
Tel. 511.20.99

3. DR. ADOLFO GUZMAN ARENAS
Investigador
IIMAS
UNAM
México 20, D.F.
Tel. 550.52.15 Ext. 4585

4. M. EN C. MARCIAL PORTILLA ROBERTSON
Jefe de la División de Computación
Facultad de Ingeniería
U N A M
México 20, D.F.
Tel. 550.52.15 Ext. 3746

5. ING. DANIEL RIOS ZERTUCHE ORTUÑO
Coordinador de Procesos Electrónicos
Coordinación Informática
Dirección General de Planeación Hacendaria
S HCP
F. T. de Mier No. 198 P.B.
México, D.F.
Tel. 522.10.32

6. ING. JOSE ARMANDO TORRES FENTANES
Investigador y Coordinador
Sistemas de Circuitos Electromecánicos
División de Ingeniería Mecánica y Eléctrica
Facultad de Ingeniería
UNAM
México 20, D.F.
Tel. 550.52.15 Ext. 3752

INTRODUCCION A LAS MINICOMPUTADORAS 1980

Fecha	Tema	Hora	Profesor
Junio 20	ELEMENTOS DE UNA COMPUTADORA	17 a 21 h	M. en C. Marcial Portilla Robertson
" 21	ARQUITECTURA PDP 11	9 a 14 h	" " "
" 27	MODOS DE DIRECCIONAMIENTO	17 a 21 h	Ing. Luis Cordero Borboa
" 28	CONJUNTO INSTRUCCIONES	9 a 14 h	M. en C. Marcial Portilla Robertson
Julio 4	EDI, TKB, PIP, MAC, ETC.	17 a 21 h	M. en C. Armando Torre Fentanes.
" 5	MANEJO SUBRUTINAS Y PROGRAMAS	9 a 14 h	Ing. Luis Cordero Borboa
" 11	ENTRADA SALIDA	17 a 21 h	Dr. Adolfo Guzmán Arenas.
" 12	LABORATORIO (Fac. de Ing.)	9 a 14 h	Ing. Luis Cordero Borboa
" 18	SISTEMA OPERATIVO	17 a 21 h	Ing. Daniel Ríos Zertuche
" 19	LABORATORIO (Fac. de Ing.)	9 a 14 h	Ing. Luis Cordero Borboa
" 25	APLICACIONES	17 a 21 h	Dr. Víctor Gerez Greise
	CLAUSURA		

'edcs.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

ELEMENTOS DE UNA COMPUTADORA

M. EN C. MARCIAL PORTILLA ROBERTSON

JUNIO, 1980

1 SYSTEMS PROGRAMMING

JOHN J. DONOVAN

background

This book has two major objectives: to teach procedures for the design of software systems and to provide a basis for judgement in the design of software. To facilitate our task, we have taken specific examples from systems programs. We discuss the design and implementation of the major system components.

What is systems programming? You may visualize a computer as some sort of beast that obeys all commands. It has been said that computers are basically people made out of metal or, conversely, people are computers made out of flesh and blood. However, once we get close to computers, we see that they are basically machines that follow very specific and primitive instructions.

In the early days of computers, people communicated with them by *on* and *off* switches denoting primitive instructions. Soon people wanted to give more complex instructions. For example, they wanted to be able to say $X = 30 * Y$; given that $Y = 10$, what is X? Present day computers cannot understand such language without the aid of systems programs. Systems programs (e.g., compilers, loaders, macro processors, operating systems) were developed to make computers better adapted to the needs of their users. Further, people wanted more assistance in the mechanics of preparing their programs.

Compilers are systems programs that accept people-like languages and translate them into machine language. Loaders are systems programs that prepare machine language programs for execution. Macro processors allow programmers to use abbreviations. Operating systems and file systems allow flexible storing and retrieval of information (Fig. 1.1).

There are over 100,000 computers in use now in virtually every application. The productivity of each computer is heavily dependent upon the effectiveness, efficiency, and sophistication of the systems programs.

In this chapter we introduce some terminology and outline machine structure and the basic tasks of an operating system.

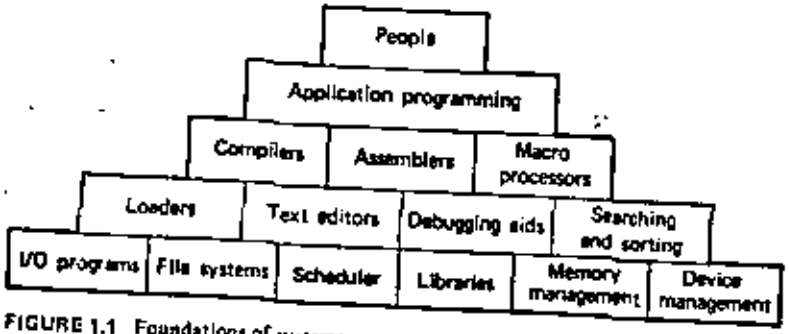


FIGURE 1.1 Foundations of systems programming

1.1 MACHINE STRUCTURE

We begin by sketching the general hardware organization of a computer system (Fig. 1.2).

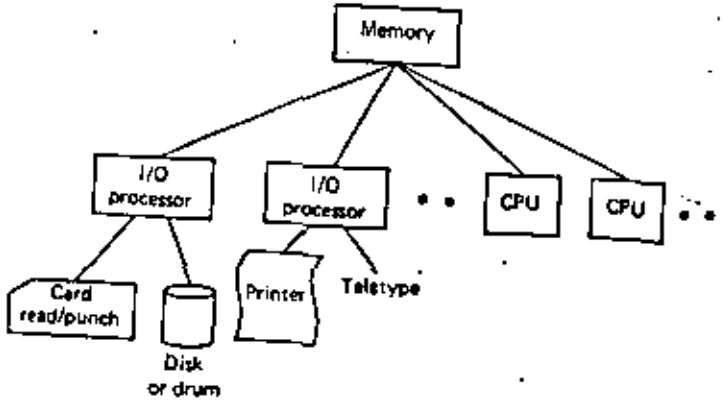


FIGURE 1.2 General hardware organization of a computer system

Memory is the device where information is stored. *Processors* are the devices that operate on this information. One may view information as being stored in the form of ones and zeros. Each one or zero is a separate binary digit called a *bit*. Bits are typically grouped in units that are called words, characters, or *bytes*. Memory locations are specified by *addresses*, where each address identifies a specific byte, word, or character.

The contents of a word may be interpreted as *data* (values to be operated on) or *instructions* (operations to be performed). A processor is a device that performs a sequence of operations specified by instructions in memory. A *program* (or procedure) is a sequence of instructions.

Memory may be thought of as mailboxes containing groups of ones and zeros. Below we depict a series of memory locations whose addresses are 10,000 through 10,002.

Address	*Contents*
10,000	0000 0000 0000 0001
10,001	0011 0000 0000 0000
10,002	0000 0000 0000 0100

An IBM 1130 processor treating location 10,001 as an instruction would interpret its contents as a "halt" instruction. Treating the same location as numerical data, the processor would interpret its contents as the binary number 0011 0000 0000 0000 (decimal 12,288). Thus instructions and data share the same storage medium.

Information in memory is coded into groups of bits that may be interpreted as characters, instructions, or numbers. A *code* is a set of rules for interpreting groups of bits, e.g., codes for representation of decimal digits (BCD), for characters (EBCDIC, or ASCII), or for instructions (specific processor operation codes). We have depicted two types of processors: *Input/Output* (I/O) processors and *Central Processing Units* (CPUs). The I/O processors are concerned with the transfer of data between memory and peripheral devices such as disks, drums, printers, and typewriters. The CPUs are concerned with manipulations of data stored in memory. The I/O processors execute I/O instructions that are stored in memory; they are generally activated by a command from the CPU. Typically, this consists of an "execute I/O" instruction whose argument is the address of the start of the I/O program. The CPU interprets this instruction and passes the argument to the I/O processor (commonly called I/O channels).

The I/O instruction set may be entirely different from that of the CPU and may be executed *asynchronously* (simultaneously) with CPU operation. Asynchronous operation of I/O channels and CPUs was one of the earliest forms of *multiprocessing*. Multiprocessing means having more than one processor operating on the same memory simultaneously.

Since instructions, like data, are stored in memory and can be treated as data, by changing the bit configuration of an instruction – adding a number to it – we may change it to a different instruction. Procedures that modify themselves are

called *impure* procedures. Writing such procedures is poor programming practice. Other programmers find them difficult to read, and moreover they cannot be shared by multiple processors. Each processor executing an impure procedure modifies its contents. Another processor attempting to execute the same procedure may encounter different instructions or data. Thus, impure procedures are not readily reusable. A *pure* procedure does not modify itself. To ensure that the instructions are the same each time a program is used, pure procedures (*re-entrant code*) are employed.

1.2 EVOLUTION OF THE COMPONENTS OF A PROGRAMMING SYSTEM

1.2.1 Assemblers

Let us review some aspects of the development of the components of a programming system.

At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and zeros (machine language), place them into the memory of the machine, and press a button, whereupon the computer would start to interpret them as instructions.

Programmers found it difficult to write or read programs in machine language. In their quest for a more convenient language, they began to use a *mnemonic* (symbol) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an *assembly language*. Programs known as *assemblers* were written to automate the translation of assembly language into machine language. The input to an assembler program is called the *source program*; the output is a machine language translation (*object program*).

1.2.2 Loaders

Once the assembler produces an object program, that program must be placed into memory and executed. It is the purpose of the loader to assure that object programs are placed in memory in an executable form.

The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be executed.

However, this would waste core[1] by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome the problems of wasted translation time and wasted memory, systems programmers developed another component, called the loader.

A *loader* is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary storage device and a loader is placed in core. The loader places into memory the machine language version of the user's program and transfers control to it. Since the loader program is much smaller than the assembler, this makes more core available to the user's program.

The realization that many users were writing virtually the same programs led to the development of "ready-made" programs (packages). These packages were written by the computer manufacturers or the users. As the programmer became more sophisticated, he wanted to mix and combine ready-made programs with his own. In response to this demand, a facility was provided whereby the user could write a main program that used several other programs or subroutines. A *subroutine* is a body of computer instructions designed to be used by other routines to accomplish a task. There are two types of subroutines: closed and open subroutines. An *open subroutine* or *macro definition* is one whose code is inserted into the main program (flow continues). Thus if the same open subroutine were called four times, it would appear in four different places in the calling program. A *closed subroutine* can be stored outside the main routine, and control transfers to the subroutine. Associated with the closed subroutine are two tasks the main program must perform: transfer of control and transfer of data.

Initially, closed subroutines had to be loaded into memory at a specific address. For example, if a user wished to employ a square root subroutine, he would have to write his main program so that it would transfer to the location assigned to the square root routine (SQRT). His program and the subroutine would be assembled together. If a second user wished to use the same subroutine, he also would assemble it along with his own program, and the complete machine language translation would be loaded into memory. An example of core allocation under this inflexible loading scheme is depicted in Figure 1.3, where core is depicted as a linear array of locations with the program areas shaded.

---

[1] Main memory is typically implemented as magnetic cores; hence *memory* and *core* are used synonymously.

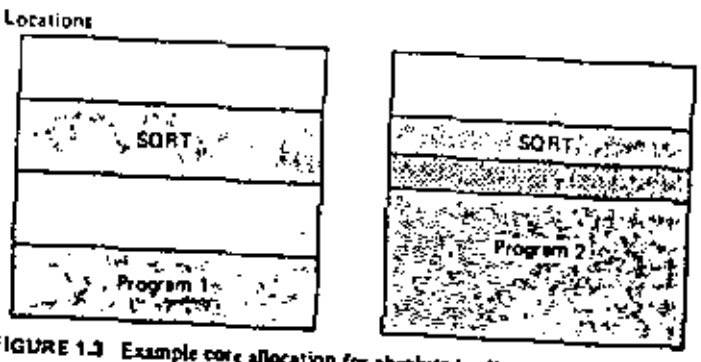


FIGURE 1.3 Example core allocation for absolute loading

Note that program 1 has "holes" in core. Program 2 *overlays* and thereby destroys part of the SQRT subroutine.

Programmers wished to use subroutines that referred to each other symbolically and did not want to be concerned with the address of parts of their programs. They expected the computer system to assign locations to their subroutines and to substitute addresses for their symbolic references.

Systems programmers noted that it would be more efficient if subroutines could be translated into an object form that the loader could "relocate" directly behind the user's program. The task of adjusting programs so they may be placed in arbitrary core locations is called *relocation*. *Relocating* loaders 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*).

The various types of loaders that we will discuss ("compile-and-go," absolute, relocating, direct-linking, dynamic-loading, and dynamic-linking) differ primarily in the manner in which these four basic functions are accomplished.

The period of execution of a user's program is called *execution time*. The period of translating a user's source program is called *assembly* or *compile time*. *Load time* refers to the period of loading and preparing an object program for execution.

1.2.3 Macros

To relieve programmers of the need to repeat identical parts of their program, operating systems provide a macro processing facility, which permits the programmer to define an abbreviation for a part of his program and to use the abbreviation in his program. The macro processor treats the identical parts of the program defined by the abbreviation as a *macro definition* and saves the definition. The macro processor substitutes the definition for all occurrences of the abbreviation (macro call) in the program.

In addition to helping programmers abbreviate their programs, macro facilities have been used as general text handlers and for specializing operating systems to individual computer installations. In specializing operating systems (systems generation), the entire operating system is written as a series of macro definitions. To specialize the operating system, a series of macro calls are written. These are processed by the macro processor by substituting the appropriate definitions, thereby producing all the programs for an operating system.

1.2.4 Compilers

As the user's problems became more categorized into areas such as scientific, business, and statistical problems, specialized languages (*high level languages*) were developed that allowed the user to express certain problems concisely and easily. These high level languages – examples are FORTRAN, COBOL, ALGOL, and PL/I – are processed by compilers and interpreters. A *compiler* is a program that accepts a program written in a high level language and produces an object program. An *interpreter* is a program that appears to execute a source program as if it were machine language. The same name (FORTRAN, COBOL, etc.) is often used to designate both a compiler and its associated language.

Modern compilers must be able to provide the complex facilities that programmers are now demanding. The compiler must furnish complex accessing methods for pointer variables and data structures used in languages like PL/I, COBOL, and ALGOL 68. Modern compilers must interact closely with the operating system to handle statements concerning the hardware interrupts of a computer (e.g. conditional statements in PL/I).

1.2.5 Formal Systems

A formal system is an uninterpreted calculus. It consists of an alphabet, a set of words called axioms, and a finite set of relations called rules of inference. Examples of formal systems are: set theory, boolean algebra, Post systems, and Backus Normal Form. Formal systems are becoming important in the design, implementation, and study of programming languages. Specifically, they can be

used to specify the *syntax* (form) and the semantics (meaning) of programming languages. They have been used in syntax-directed compilation, compiler verification, and complexity studies of languages.

1.3 EVOLUTION OF OPERATING SYSTEMS

Just a few years ago a FORTRAN programmer would approach the computer with his source deck in his left hand and a green deck of cards that would be a FORTRAN compiler in his right hand. He would:

1. Place the FORTRAN compiler (green deck) in the card hopper and press the load button. The computer would load the FORTRAN compiler.
2. Place his source language deck into the card hopper. The FORTRAN compiler would proceed to translate it into a machine language deck, which was punched onto red cards.
3. Reach into the card library for a pink deck of cards marked "loader," and place them in the card hopper. The computer would load the loader into its memory.
4. Place his newly translated object deck in the card hopper. The loader would load it into the machine.
5. Place in the card hopper the decks of any subroutines which his program called. The loader would load these subroutines.
6. Finally, the loader would transfer execution to the user's program, which might require the reading of data cards.

This system of multicolored decks was somewhat unsatisfactory, and there was strong motivation for moving to a more flexible system. One reason was that valuable computer time was being wasted as the machine stood idle during card-handling activities and between jobs. (A *job* is a unit of specified work, e.g., an assembly of a program.) To eliminate this waste, the facility to *batch* jobs was provided, permitting a number of jobs to be placed together into the card hopper to be read. A *batch operating system* performed the task of batching jobs. For example the batch system would perform steps 1 through 6 above retrieving the FORTRAN compiler and loader from secondary storage.

As the demands for computer time, memory, devices, and files increased, the efficient management of these resources became more critical. In Chapter 9 we discuss various methods of managing them. These resources are valuable, and inefficient management of them can be costly. The management of each resource has evolved as the cost and sophistication of its use increased.

In simple batched systems, the memory resource was allocated totally to a single program. Thus, if a program did not need the entire memory, a portion of that resource was wasted. Multiprogramming operating systems with *partitioned core memory* were developed to circumvent this problem. *Multiprogramming* allows multiple programs to reside in separate areas of core at the same time. Programs were given a fixed portion of core (*Multiprogramming with Fixed Tasks* (MFT)) or a varying-size portion of core (*Multiprogramming with Variable Tasks* (MVT)).

Often in such partitioned memory systems some portion could not be used since it was too small to contain a program. The problem of "holes" or unused portions of core is called *fragmentation*. Fragmentation has been minimized by the technique of relocatable partitions (Burroughs 6500) and by paging (XDS 940, HIS 645). *Relocatable partitioned core* allows the unused portions to be condensed into one continuous part of core.

Paging is a method of memory allocation by which the program is subdivided into equal portions or pages, and core is subdivided into equal portions or *blocks*. The pages are loaded into blocks.

There are two paging techniques: simple and demand. In *simple paging* all the pages of a program must be in core for execution. In *demand paging* a program can be executed without all pages being in core, i.e., pages are fetched into core as they are needed (demanded).

The reader will recall from section 1.1 that a system with several processors is termed a multiprocessing system. The *traffic controller* coordinates the processors and the processes. The resource of processor time is allocated by a program known as the *scheduler*. The processor concerned with I/O is referred to as the *I/O processor*, and programming this processor is called *I/O programming*.

The resource of files of information is allocated by the *file system*. A *segment* is a group of information that a user wishes to treat as an entity. *Files* are segments. There are two types of files: (1) directories and (2) data or programs. *Directories* contain the locations of other files. In a hierarchical file system, directories may point to other directories, which in turn may point to directories or files.

Time-sharing is one method of allocating processor time. It is typically characterized by interactive processing and time-slicing of the CPU's time to allow quick response to each user.

A *virtual memory* (*name space*, *address space*) consists of those addresses that may be generated by a processor during execution of a computation. The *memory space* consists of the set of addresses that correspond to physical memory locations. The technique of *segmentation* provides a large name space and a good

protection mechanism. Protection and sharing are methods of allowing controlled access to segments.

1.4 OPERATING SYSTEM USER VIEWPOINT: FUNCTIONS

From the user's point of view, the purpose of an operating system (monitor) is to assist him in the *mechanics* of solving problems. Specifically, the following functions are performed by the system:

1. Job sequencing, scheduling, and traffic controller operation
2. Input/output programming
3. Protecting itself from the user; protecting the user from other users
4. Secondary storage management
5. Error handling

Consider the situation in which one user has a job that takes four hours, and another user has a job that takes four seconds. If both jobs were submitted simultaneously, it would seem to be more appropriate for the four-second user to have his run go first. Based on considerations such as this, job scheduling is automatically performed by the operating system. If it is possible to do input and output while simultaneously executing a program, as is the case with many computer systems, all these functions are scheduled by the traffic controller.

As we have said, the I/O channel may be thought of as a separate computer with its own specialized set of instructions. Most users do not want to learn how to program it (in many cases quite a complicated task). The user would like to simply say in his program, "Read," causing the monitor system to supply a program to the I/O channel for execution. Such a facility is provided by operating systems. In many cases the program supplied to the I/O channel consists of a sequence of closely interwoven interrupt routines that handle the situation in this way: "Hey, Mr. I/O Channel, did you receive that character?" "Yes, I received it." "Are you sure you received it?" "Yes, I'm sure." "Okay, I'll send another one." "Fine, send it." "You're sure you want me to send another one?" "*Send* it!"

An extremely important function of an operating system is to protect the user from being hurt, either maliciously or accidentally, by other users; that is, protect him when other users are executing or changing their programs, files, or data bases. The operating system must insure inviolability. As well as protecting users from each other, the operating system must also protect itself from users who, whether maliciously or accidentally, might "crash" the system.

Students are great challengers of protection mechanisms. When the systems

programming course is given at M.I.T., we find that due to the large numb students participating it is very difficult to personally grade every program on the machine problems. So for the very simple problems – certainly the problem which may be to count the number of A's in a register and the answer in another register – we have written a grading program that cluded as part of the operating system. The grading program calls the stuc program and transfers control to it. In this simple problem the student's prc processes the contents of the register, leaves his answer in another register returns to the grading program. The latter checks to find out if the c number has been left in the answer register. Afterwards, the grading prc prints out a listing of all the students in the class and their grades. For exa

```
VITA KOHN          — CORRECT
RACHEL BUXBAUM     — CORRECT
JOE LEVIN          — INCORRECT
LOFTI ZADEH        — CORRECT
```

On last year's run, the computer listing began as follows:

```
JAMES ARCHER       — CORRECT
ED MCCARTHY        — CORRECT
ELLEN NANGLE       — INCORRECT
JOHN SCHWARTZ      — MAYBE
```

(We are not sure how John Schwartz did this; we gave him an A in the cr

Secondary storage management is a task performed by an operating sys conjunction with the use of disks, tapes, and other secondary storage for s programs and data.

An operating system must respond to errors. For example, if the progr should overflow a register, it is not economical for the computer to simpl and wait for an operator to intervene. When an error occurs, the operating must take appropriate action.

1.5 OPERATING SYSTEM USER VIEWPOINT: BATCH CONTROL LANGUAGE

Many users view an operating system only through the batch system cards by which they must preface their programs. In this section we will a simple monitor system and the control cards associated with it. Othe complex monitors are discussed in Chapter 9.

Monitor is a term that refers to the control programs of an operating system. Typically, in a batch system the jobs are stacked in a card reader, and the monitor system sequentially processes each job. A job may consist of several separate programs to be executed sequentially, each individual program being called a *job step*. In a *batch monitor system* the user communicates with the system by way of a control language. In a simple batch monitor system we have two classes of control cards: execution cards and definition cards. For example, an execution card may be in the following format:

```
// step name  EXEC   name of program to be executed, Argument 1, Argument 2
```

The job control card, a definition card, may take on the following format:

```
// job name  JOB      (User name, identification, expected time use, lines to
                      be printed out, expected number of cards to be printed
                      out)
```

Usually there is an end-of-file card, whose format might consist of /*, signifying the termination of a collection of data. Let us take the following example of a FORTRAN job.

```
//EXAMPLE   JOB     DONOVAN,    T168,1,100,0
//STEP1     EXEC    FORTRAN, NOPUNCH
          READ 9100,N
          DO 100 I = 1,N
          I2 = I**I
          I3 = I**I**I
      100 PRINT 9100, I, I2, I3
     9100 FORMAT (3I10)
          END
/*
//STEP2     EXEC LOAD
/*
//STEP3     EXEC OBJECT
       10
/*
```

The first control card is an example of a definition card. We have defined the user to be Donovan. The system must set up an accounting file for the user, noting that he expects to use one minute of time, to output a hundred lines of output, and to punch no cards. The next control card, EXEC FORTRAN, NOPUNCH, is an example of an execution card; that is, the system is to execute the program FORTRAN, given one argument – NOPUNCH. This argument allows the monitor system to perform more efficiently; since no cards are to be punched, it need not utilize the punch routines. The data to the compiler is the FORTRAN program shown, terminated by an end-of-file card /*.

The next control card is another example of an execution card and in this

case causes the execution of the loader. The program that has just been compiled will be loaded, together with all the routines necessary for its execution, whereupon the loader will "bind" the subroutines to the main program. This job step is terminated by an end-of-file card. The EXEC OBJECT card is another execution card, causing the monitor system to execute the object program just compiled. The data card, 10, is input to the program and is followed by the end-of-file card.

The simple loop shown in Figure 1.4 presents an overview of an implementation of a batch monitor system. The monitor system must read in the first card, presumably a job card. In processing a job card, the monitor saves the user's name, account number, allotted time, card punch limit, and line print limit. If the next control card happens to be an execution card, then the monitor will load the corresponding program from secondary storage and process the job step by transferring control to the executable program. If there is an error during processing, the system notes the error and goes back to process the next job step.

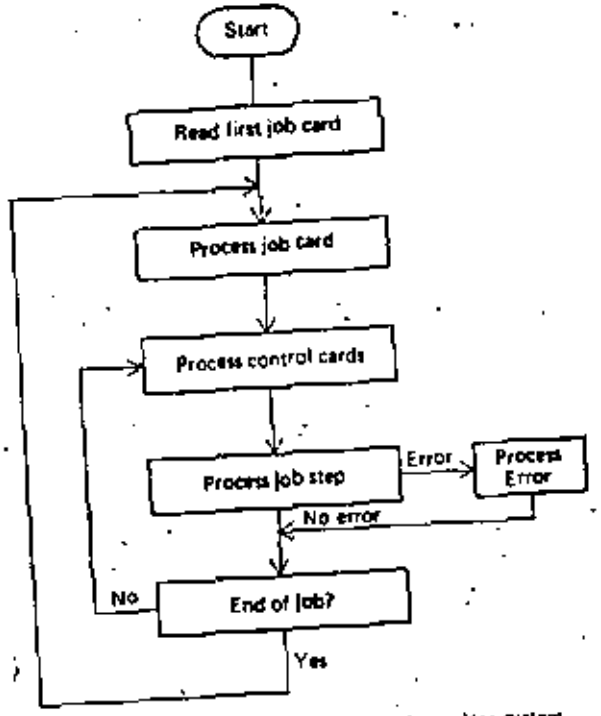


FIGURE 1.4. Main loop of a simple batch monitor system

1.6 OPERATING SYSTEM USER VIEWPOINT: FACILITIES

For the applications-oriented user, the function of the operating system is to provide facilities to help solve problems. The questions of scheduling or protection are of no interest to him; what he is concerned with is the available software. The following facilities are typically provided by modern operating systems:

1. Assemblers
2. Compilers, such as FORTRAN, COBOL, and PL/I
3. Subroutine libraries, such as SINE, COSINE, SQUARE ROOT
4. Linkage editors and program loaders that bind subroutines together and prepare programs for execution
5. Utility routines, such as SORT/MERGE and TAPE COPY
6. Application packages, such as circuit analysis or simulation
7. Debugging facilities, such as program tracing and "core dumps"
8. Data management and file processing
9. Management of system hardware

Although this "facilities" aspect of an operating system may be of great interest to the user, we feel that the answer to the question, "How many compilers does that operating system have?" may tell more about the orientation of the manufacturer's marketing force than it does about the structure and effectiveness of the operating system.

1.7 SUMMARY

The major components of a programming system are:

1. Assembler

Input to an assembler is an *assembly language program*. Output is an object program plus information that enables the loader to prepare the object program for execution.

2. Macro Processor

A *macro call* is an abbreviation (or name) for some code. A *macro definition* is a sequence of code that has a name (macro call). A *macro processor* is a program that substitutes and specializes macro definitions for macro calls.

3. Loader

A loader is a routine that loads an object program and prepares it for execution.

There are various loading schemes: absolute, relocating, and direct-linking. In general, the loader must *load*, *relocate*, and *link* the object program.

4. Compilers

A compiler is a program that accepts a source program "in a high-level language" and produces a corresponding object program.

5. Operating Systems

An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a *traffic controller*, a *scheduler*, *memory management module*, *I/O programs*, and a *file system*.

Input-Output Devices

DIGITAL PRINCIPLES AND APPLICATIONS

10

MALVIO/LEACH

In any digital system it is necessary to have a link of communication between man and machine. This communication link is often called the "man-machine interface" and it presents a number of problems. Digital systems are capable of operating on information at speeds much greater than man's, and this is one of their most important attributes. For example, a large-scale digital computer is capable of performing more than 500,000 additions per second.

The problem here is to provide data input to the system at the highest possible rate. At the same time, there is the problem of accepting data output from the system at the highest possible rate. The problem is further magnified since most digital systems do not speak English, or any other language for that matter, and some system of symbols must therefore be used for communication (there is at present a considerable amount of research in this area, and some systems have been developed which will accept spoken commands and give oral responses on a limited basis).

Since digital systems operate in a binary mode, a number of code systems which are binary representations have been developed and are being used as the language of communication between man and machine. In this chapter we discuss a number of these codes and, at the same time, consider the necessary input-output equipment.

The primary objective of this chapter is to acquire the ability to

1. Explain how Hollerith code and ASCII code are used in input/output media.
2. Discuss techniques for magnetic recording of digital information, including RZ, RZI, and NRZI.
3. Describe the limitations of a number of different digital input/output units.
4. Draw the logic diagrams for a simple tree decoder and a balanced multiplicative decoder.

10-1 PUNCHED CARDS

One of the most widely used media for entering data into a machine, or for obtaining output data from a machine, is the punched card. Some common examples of these cards are college registration cards, government checks, monthly oil company statements, and bank statements. It is quite simple to use this medium to represent binary information, since only two conditions are required. Typically, a hole in the card represents a 1 and the absence of a hole represents a 0. Thus, the card provides the means of presenting information in binary form, and it is only necessary to develop the code.

The typical punched card used in large-scale data-processing systems is 7⅜ in long, 3¼ in wide, and 0.007 in thick. Each card has 80 vertical columns, and there are 12 horizontal rows, as shown in Fig. 10-1. The columns are numbered 1 through 80 along the bottom edge of the card. Beginning at the top of the card, the rows are designated 12, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The bottom edge of the card is the 9 edge, and the top edge is the *12 edge*. Holes in the 12, 11, and 0 rows are called *zone punches*, and holes in the 0 through 9 rows are called *digit punches*. Notice that row 0 is both a zone-and a digit-punch row. Any number, any letter in the alphabet, or any of several special characters can be represented on the card by punching one or more holes in any one column. Thus, the card has the capacity of 80 numbers, letters, or combinations.

Probably the most widely used system for recording information on a punched card is the *Hollerith code*. In this code the numbers 0 through 9 are represented by a single punch in a vertical column. For example, a hole punched in the fifth row of column 12 represents a 5 in that column. The letters of the alphabet are represented by two punches in any one column. The letters A through I are represented by a zone punch in row 12 and a punch in rows 1 through 9. The letters J through R are represented by a zone punch in row 11 and a punch in rows 1 through 9. The letters S through Z are represented by a zone punch in row 0 and a punch in rows 2 through 9. Thus, any of the 10 decimal digits and any of the 26 letters of the alphabet can be represented in a binary fashion by punching the proper holes in the card. In addition, a number of special characters can be represented by punching combinations of holes in a column which are not used for the numbers or letters of the alphabet. These characters are shown with the proper punches in Fig. 10-1.

An easy device for remembering the alphabetic characters is the phrase "JR. is 11." Notice that the letters J through R have an 11 punch, those before have a 12 punch, and those after have a 0 punch. It is also necessary to remember that S begins on a 2 and not a 1.

Example 10-1

Decode the information punched in the card in Fig. 10-2.

Solution

Column 1 has a 2 punch in row 0 and a punch in row 3. It is therefore the letter T. Column 2 has a zone punch in row 12 and another punch in row 8. It is

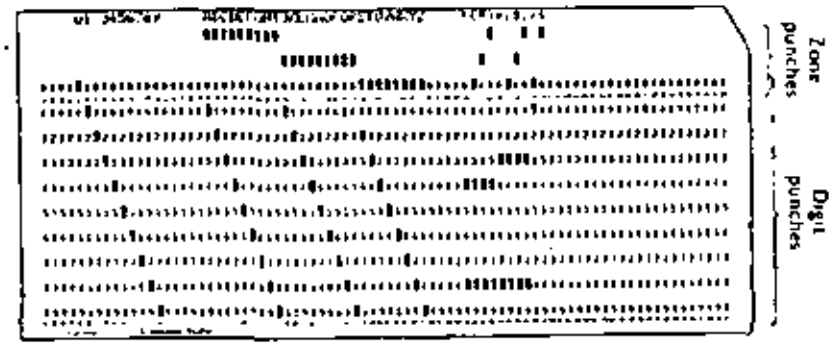


Fig. 10-1. Standard punched card using Hollerith code.

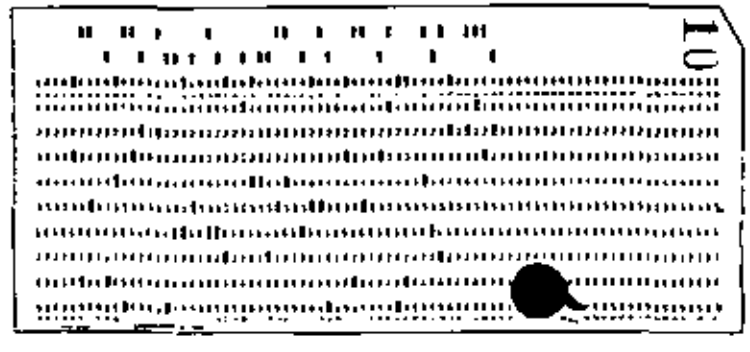
therefore the letter H. Continuing in this fashion, you should see that the complete message reads, "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOGS BACK."

With this card code, any *alphanumeric* (alphabetic and numeric) information can be used as input to a digital system. On the other hand, the system is capable of delivering alphanumeric output information to the user. In scientific disciplines, the information might be missile flight number, location, or guidance information such as pitch rate, roll rate, and yaw rate. In business disciplines, the information could be account numbers, names, addresses, monthly statements, etc. In any case, the information is punched on the card with one character per column, and the card is then capable of containing a maximum of 80 characters.

Each card is considered as one block or unit of information. Since the machine operates on one card at a time, the punched card is often referred to as a "unit record." Moreover, the digital equipment used to punch cards, read cards into a system, sort cards, etc., is referred to as "unit-record equipment."

Occasionally, the information used with a digital system is entirely numeric; that is, no alphabetic or special characters are required. In this case, it is possible to input the information to the system by punching the cards in a straight binary fashion. In this system, the absence of a punch is a binary 0, and a punch is a

Fig. 10-2. Example 10-1.



binary 1. It is then possible to punch 80 × 12 = 960 bits of binary information on one card.

Many large-scale data-processing systems use binary information in blocks of 36 bits. Each block of 36 bits is called a "word." You will recall from the previous chapter that a register capable of storing a 36-bit word must contain 36 flip-flops. There is nothing magical about the 36-bit word, and there are in fact other systems which operate with other word lengths. Even so, let's see how binary information arranged in words of 36 bits might be punched on cards.

There are two methods. The first method stores the information on the card horizontally by punching across the card from left to right. The first 36-bit word is punched in row 9 in columns 1 through 36. The second word is also in row 9, in columns 37 through 72. The third word is in row 8, columns 1 through 36, and so on. Thus a total of twenty-four 36-bit words can be punched in the card in straight binary form. It is then possible to store 864 bits of information on the card.

The second method involves punching the information vertically in columns rather than rows. Beginning in row 12 of column 1, the first 12 bits of the word are punched in rows 12, 11, 0, . . . , 9. The next 12 bits are punched in column 2, and the remaining 12 bits are punched in column 3. Thus, a 36-bit word can be punched in every three columns. The card is then capable of containing twenty-six 36-bit words.

The most common method of entering information into punched cards initially is by means of the key-punch machine. This machine operates very much the same as a typewriter, and the speed and accuracy of the operation depend entirely on the operator. The information on the punched cards can then be read into the digital system by means of a card reader. The information can be entered into the system at the rate of 100 to 1,000 cards per minute, depending on the type of card reader used.

The basic method for changing the punched information into the necessary electrical signals is shown in Fig. 10-3. The cards are stacked in the *read* hopper and are drawn from it one at a time. Each card passes under the *read* heads, which are either brushes or photocells. There is one *read* head for each column on the card, and when a hole appears under the *read* head an electrical signal is generated.

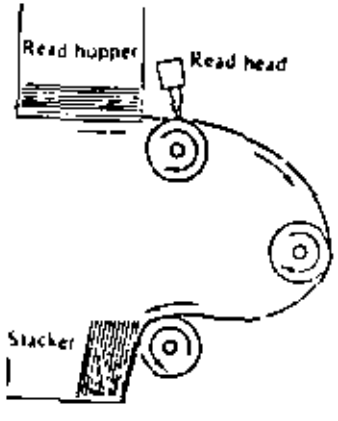


Fig. 10-3. Card-reading operation.

Thus, each signal from the *read* heads represents a binary 1, and this information can be used to set flip-flops which form the input storage register. The cards then pass over other rollers and are placed in the stacker. There is quite often a second *read* head which reads the data a second time to provide a validity check on the reading process.

Example 10-2

Suppose a deck of cards has binary data punched in them. Each card has twenty-four 36-bit words. If the cards are read at a rate of 600 cards per minute, what is the rate at which data are entering the system?

Solution

Since each card contains 24 words, the data rate is 24 × 600 = 14,400 words per minute. This is equivalent to 36 × 14,400 = 518,400 bits per minute, or 518,400/60 = 8,640 bits per second.

Punched cards can also be used as a medium for accepting data output from a digital system. In this case, a stack of blank cards (having no holes punched in them) are held in a hopper in a card punch which is controlled by the digital system. The blank cards are drawn from the hopper one at a time and punched with the proper information. They are then passed under *read* heads, which check the validity of the punching operation, and stacked in an output hopper. Card punches are capable of operating at 100 to 250 cards per minute, depending on the system used.

Punched cards present a number of important advantages, the first of which is the fact that the cards represent a means of storing information permanently. Since the information is in machine code, and since this information can be printed on the top edge of the card, this is a very convenient means of communication between man and machine, and between machine and machine. There is also a wide variety of peripheral equipment which can be used to process information stored on cards. The most common are sorters, collators, calculating punches, reproducing punches, and accounting machines. Moreover, it is very easy to correct or change the information stored, since it is only necessary to remove the desired card(s) and replace it (them) with the corrected one(s). Finally, these cards are quite inexpensive.

10-2 PAPER TAPE

Another widely used input-output medium is punched paper tape. It is used in much the same way as punched cards. Paper tape was developed initially for the purpose of transmitting telegraph messages over wires. It is now used extensively for storing information and for transmitting information from machine to machine. Paper tape differs from cards in that it is a continuous roll of paper; thus, any amount of information can be punched into a roll. It is possible to record any alphabetic or numeric character, as well as a number of special characters, on paper tape by punching holes in the tape in the proper places.

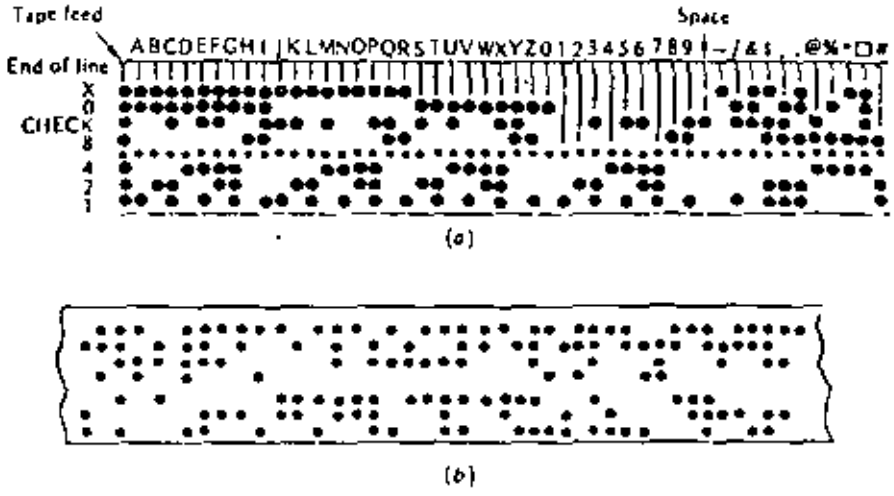


Fig. 10-4. Punched paper tape. (a) Eight-hole code. (b) Example 10-3.

There are a number of codes for punching data in paper tape, but one of the most widely used is the *eight-hole code* in Fig. 10-4a. Holes, representing data, are punched in eight parallel channels which run the length of the tape. (The channels are labeled 1, 2, 4, 8, parity, 0, *X*, and end of line.) Each character,—numeric, alphabetic, or special,—occupies one column of eight positions across the width of the tape.

Numbers are represented by punches in one or more channels labeled 0, 1, 2, 4, and 8, and each number is the sum of the punch positions. For example, 0 is represented by a single punch in the 0 channel; 1 is represented by a single punch in the 1 channel; 2 is a single punch in channel 2; 3 is a punch in channel 1 and a punch in channel 2, etc. Alphabetic characters are represented by a combination of punches in channels *X*, 0, 1, 2, 4, and 8. Channels *X* and 0 are used much as the zone punches in punched cards. For example, the letter *A* is designated by punches in channels *X*, 0, and 1. The special characters are represented by combinations of punches in all channels which are not used to designate either numbers or letters. A punch in the end-of-line channel signifies the end of a block of information, or the end of record. This is the only time a punch appears in this channel.

As a means of checking the validity of the information punched on the tape, the parity channel is used to ensure that each character is represented by an *odd* number of holes. For example, the letter C is represented by punches in channels *X*, 0, 1, and 2. Since an odd number of holes is required for each character, the code for the letter C also has a punch in the parity channel, and thus a total of *five* punches is used for this letter.

Example 10-3

What information is held in the perforated tape in Fig. 10-4*b*?

Solution

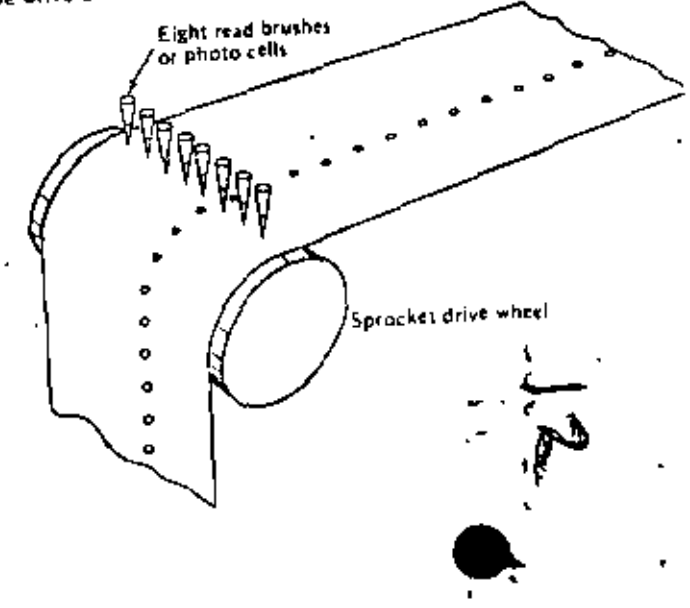
The first character has punches in channels 0, 1, and 2, and this is the letter T. The second character is the letter H, since there are punches in channels *X*, 0, and 8. Continuing, you should see that the message is the same as that punched on the card in Example 10-1.

The row of smaller holes between channels 4 and 8 are guide holes, used to guide and drive the tape under the read positions. The information on the tape can be sensed by brushes or photocells as shown in Fig. 10-5. The method for reading information from the paper tape and inputting it into the digital system is very similar to that used for reading punched cards. Depending on the type of reader used, information can be read into the system at a rate of 150 to 1,000 characters per second. You will notice that this is only slightly faster than reading information from punched cards.

Paper tape can be used as a means of accepting information output from a digital system. In this case the system drives a tape punch which enters the data on the tape by punching the proper holes. Typical tape punches are capable of operating at rates of 15 characters per second, and the data are punched with 10 characters to the inch. The number of characters per inch is referred to as the "data density," and in this case the density is 10 characters per inch. Recording density is one of the important features of magnetic-tape recording which will be discussed in the next session.

Paper tape can also be perforated by a manual tape punch. This unit is very similar to an electric typewriter, and indeed in some cases electric typewriters with special punching units attached are used. The accuracy and speed of this method are again a function of the machine operator. One advantage of this method is tha

Fig. 10-5. Paper-tape drive and reading mechanism.



the typewriter provides a written copy of what is punched into the tape. This copy can be used for verification of the punched information.

10-3 MAGNETIC TAPE

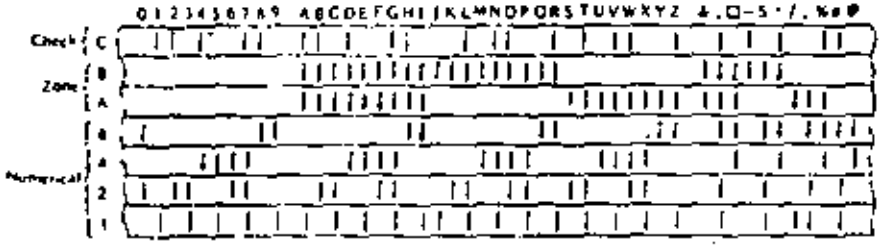
Magnetic tape has become one of the most important methods for storing large quantities of information. Magnetic tape offers a number of advantages over punched cards and punched paper tape. One of the most important is the fact that magnetic tape can be erased and used over and over. Reading and recording are much faster than with either cards or paper tape. However, they require the use of a tape-drive unit which is much more expensive than the equipment used with cards and paper tape. On the other hand, it is possible to store up to 20 million characters on one 2,400-ft reel of magnetic tape, and if a high volume of data is one of the system requirements, the use of magnetic tape is well justified. Most commonly, magnetic tape is supplied on 2,400-ft reels. The tape itself is a ½-in-wide strip of plastic with a magnetic oxide coating on one side.

Data are recorded on the tape in seven parallel channels along the length of the tape. The channels are labeled 1, 2, 4, 8, A, B, and C as shown in Fig. 10-6. Since the information recorded on the tape must be digital in form, that is, there must be two states, it is recorded by magnetizing spots on the tape in one of two directions.

A simplified presentation of the *write* and *read* operations is shown in Fig. 10-7. The magnetic spots are recorded on the tape as it passes over the *write* head as shown in Fig. 10-7*a*. If a positive pulse of current is applied to the *write*-head coil, as shown in the figure, a magnetic flux is set up in a clockwise direction around the *write* head. As this flux passes through the record gap, it spreads slightly and passes through the oxide coating on the magnetic tape. This causes a small area on the tape to be magnetized with the polarity shown in the figure. If a current pulse of the opposite polarity is applied, the flux is set up in the opposite direction, and a spot magnetized in the opposite direction is recorded on the tape. Thus, it is possible to record data on the tape in a digital fashion. The spots shown in the figure are greatly exaggerated in size to show the direction of magnetization clearly.

In the *read* operation shown in Fig. 10-7*b*, a magnetized spot on the tape sets up a flux in the *read* head as the tape passes over the *read* gap. This flux induces a small voltage in the *read*-head coil which can be amplified and used to set or reset a flip-flop. Spots of opposite polarities on the tape induce voltages of opposite

Fig. 10-6. Magnetic-tape code.



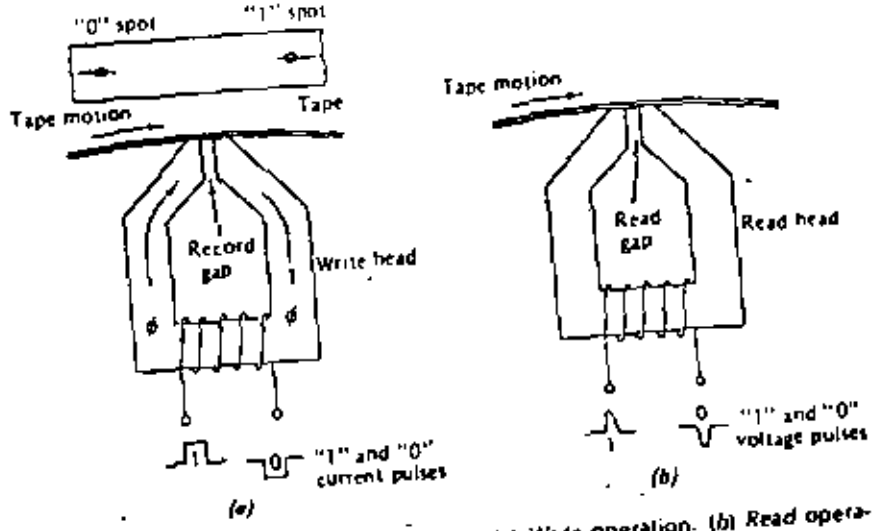


Fig. 10-7. Magnetic-tape recording and reading. (a) *Write* operation. (b) *Read* operation.

polarities in the *read* coil, and thus both 1s and 0s can be sensed. There is one *read/write* head for each of the seven channels on the tape. Typically, *read/write* heads are constructed in pairs as shown in Fig. 10-8. Thus, the *write* operation can be set up as a self-checking operation. That is, data recorded on tape are immediately read as they pass over the *read* gap and can be checked for validity.

A coding system similar to that used to punch data on cards is used to record alphanumeric information on tape. Each character occupies one column of seven bits across the width of the tape. The code is shown in Fig. 10-6. There are two independent systems for checking the validity of the information stored on the tape.

The first system is a vertical parity bit which is written in channel C of the tape. This is called a "character-check bit" and is written in channel C to ensure that all characters are represented by an even number of bits. For example, the letter A is

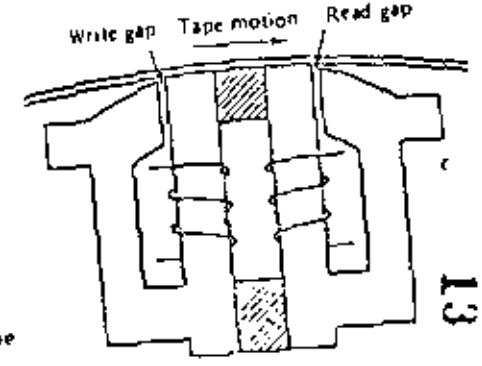


Fig. 10-8. Magnetic-tape *read/write* heads.

represented by spots in channels 1, *A*, and *B*. Since this is only three spots, an additional spot is recorded in channel *C* to maintain even parity for this character.

The second system is the *horizontal parity-check bit*. This is sometimes referred to as the longitudinal parity bit, and it is written, when needed, at the end of a block of information or record. The total number of bits recorded in each channel is monitored, and at the end of a record, a parity bit is written if necessary to keep the total number of bits an even number. These two systems form an even-parity system. They could, of course, just as easily be implemented to form an odd-parity system. Information can also be recorded on the tape in straight binary form. In this case, a 36-bit word is written across the width of the tape in groups of six bits. Thus it requires six columns to record one 36-bit word.

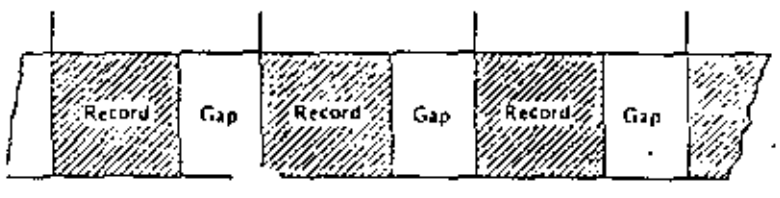
The vertical spacing between the recorded spots on the tape is fixed by the positions of the *read/write* heads. The horizontal spacing is a function of the tape speed and the recording speed. Tape speeds vary from 50 to 200 in/s, but 75 and 112.5 in/s are quite common.

The maximum number of characters recorded in 1 in of tape is called the "recording density," and it is a function of the tape speed and the rate at which data are supplied to the *write* head. Typical recording densities are 200, 556, and 800 bits per inch. Thus it can be seen that a total of $800 \times 2{,}400 \times 12 = 23.02 \times 10^6$ characters can be stored on one 2,400-ft reel of tape. This would mean that the data would have to be stored with no gaps between characters or groups of characters.

For purposes of locating information on tape, it is most common to record information in groups or blocks called "records." In between records there is a blank space of tape called the "interrecord gap." This gap is typically a 0.75-in space of blank tape, and it is positioned over the *read/write* heads when the tape stops. The interrecord gap provides the space necessary for the tape to come up to the proper speed before recording or reading of information can take place. The total number of characters recorded on a tape is then also a function of the record length (or the total number of interrecord gaps, since they represent blank space on the tape).

The data as recorded on the tape, including records (actual data) and interrecord gaps, can be represented as shown in Fig. 10-9. If there were no interrecord gaps, the total number of characters recorded could be found by multiplying the length of the tape in inches by the recording density in characters per inch. If the record were exactly the same length as the interrecord gap, the total storage would be cut in half. Thus, it is desirable to keep the records as long as possible in order to use the tape most efficiently.

Fig. 10-9. Recording data on magnetic tape.



Given any one tape system and the recording density, it is a simple matter to determine the actual storage capacity of the tape. Consider the length of tape composed of one record and one record gap as shown in Fig. 10-9. This length of tape is repeated over and over down the length of the tape. The total number of characters that could be stored in this length of tape is the sum of the characters in the record R and the characters which could be stored in the record gap. The number of characters which could be stored in the gap is equal to the recording density D multiplied by the gap length C. Thus the total number of characters which could be stored in this length of tape is given by $R + CD$. The ratio of the characters actually recorded R to the total possible could be called a tape-utilization factor F and is given by

$$F = \frac{R}{R + CD} \tag{10-1}$$

Examination of the tape-utilization factor shows that if the total number of characters in the record is equal to the number of characters which could be stored in the gap, the utilization factor reduces to 0.5. This utilization factor can be used to determine the total storage capacity of a magnetic tape if the recording density and the record length are known. Thus the total number of characters stored on a tape *CHAR* is given by

$$CHAR = LDF \tag{10-2}$$

where L = length of tape, in
D = recording density, characters per inch.

For a standard 2,400-ft reel of tape having a 0.75-in record gap, the formula in Eq. (10-2) reduces to

$$CHAR = \frac{2{,}400 \times 12 \times DR}{R + 0.75D} \tag{10-3}$$

Example 10-4

What is the total storage capacity of a 2,400-ft reel of magnetic tape if data are recorded at a density of 556 characters per inch and the record length is 100 characters?

Solution

The total number of characters can be found using Eq. (10-3).

$$CHAR = \frac{2{,}400 \times 12 \times 556 \times 100}{100 + (0.75 \times 556)} = 3.10 \times 10^6$$

This result can be checked by calculating the tape-utilization fact

$$F = \frac{100}{100 + (0.75 \times 556)} = \frac{1}{5.17} = 0.19$$

The maximum number of characters that can be stored on the tape is $2{,}400 \times 12 \times 556 = 16.0128 \times 10^6$. Multiplying this by the utilization factor gives

$$CHAR = 16.0128 \times 10^6 \times \frac{1}{5.17} = 3.10 \times 10^6$$

10-4 DIGITAL RECORDING METHODS

There are a number of methods for recording data on a magnetic surface. The methods fall into two general categories, called "return-to-zero" and "non-return-to-zero," and they apply to magnetic-tape recording as well as recording on magnetic disk and drum surfaces (magnetic-disk and magnetic-drum storage will be discussed in a later chapter).

In the previous section, it was stated that digital information could be recorded on magnetic tape by magnetizing spots on the tape with opposite polarities. This type of recording is known as return-to-zero, or RZ for short, recording. The technique for recording data on tape using this method is to apply a series of current pulses to the write-head winding as shown in Fig. 10-10. The current pulses set up corresponding fluxes in the write head, as shown in the figure. The spots magnetized on the tape have polarities corresponding to the direction of the flux waveform, and it is only necessary to change the direction of the input current to write 1s or 0s. Notice that the input current and the flux waveform return to a zero reference level between individual bits. Thus the term "return to zero."

When it is desired to read the recorded information from the tape, the tape is passed over the *read* heads and the magnetized spots induce voltages in the read-coil winding as shown in the figure. Notice that there is somewhat of a problem here, since all the pulses have both positive and negative portions. One method of detecting these levels properly is to strobe the output waveform. That is, the output-

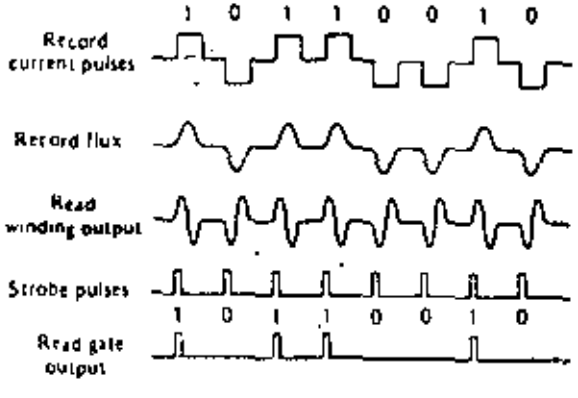


Fig. 10-10. Return-to-zero recording and reading.

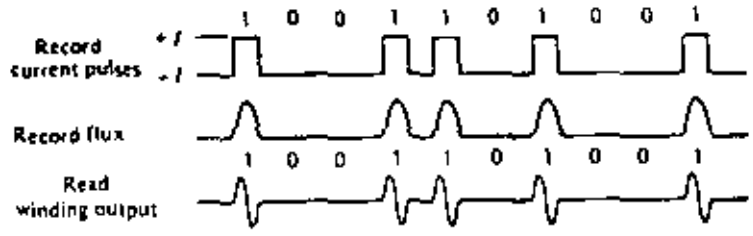


Fig. 10-11. Biased return-to-zero recording and reading.

voltage waveform is applied to one input of an AND gate (after being amplified), and a clock or strobe pulse is applied to the other input to the gate. The strobe pulse must be very carefully timed to ensure that it samples the output waveform at the proper time. This is one of the major difficulties of this type of recording, and it is therefore seldom used except on magnetic drums. On a magnetic drum, the strobe waveform can be recorded on one track of the drum, and thus the proper timing is achieved.

A second difficulty with this type of recording is the fact that between bits there is no record current, and thus between the spots on the tape the magnetic surface is randomly oriented. This means that if a new recording is to be made over old data, the new data have to be recorded precisely on top of the old data. If they are not, the old data will not be erased, and the tape will contain a conglomeration of information. The tape could be erased by installing another set of erase heads, but this is costly and unnecessary.

A method for curing these problems is to bias the record head with a current which will saturate the tape in either one direction or the other. In this system, a current pulse of positive polarity is applied only when it is desired to write a 1 on the tape as shown in Fig. 10-11. At all other times the flux in the write heads is sufficient to magnetize the entire track in the 0 direction. Now, recording data over old data is not a problem since the tape is effectively erased as it passes over the record heads. Moreover, the timing is not so critical since it is not necessary to record exactly over the previous data. When data are recorded in this fashion and then played back, a pulse appears at the output of the *read* winding only when a 1 has been recorded on the tape. This makes reading the information from the tape much simpler.

The *non-return-to-zero*, or NRZ, recording technique is a variation of the RZ technique where the write current pulses do not return to some reference level between bits. The NRZ recording technique can be best explained by examining the record-current waveform shown in Fig. 10-12. Notice that the current is at $+I$ while recording 1s and at $-I$ while recording 0s. Since the current levels are always at either $+I$ or $-I$, the recording problems of the first RZ system do not exist here.

Notice that the voltage at the *read*-winding output has a pulse only when the recorded data change from a 1 to a 0 or vice versa. Therefore, some means of sensing the recorded data is necessary for the *read* operation. If the read-winding voltage is amplified and used to set or reset a flip-flop as shown in the figure, the A side of the flip-flop is high during each time that a 1 is being read. It is low during

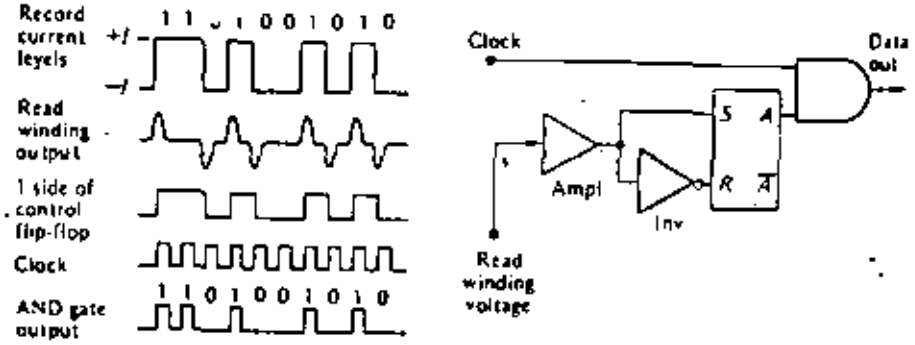


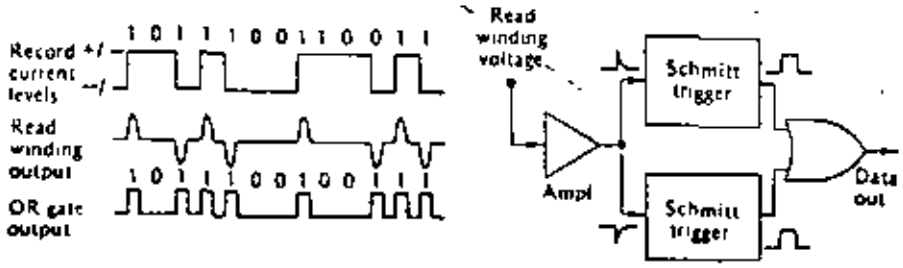
Fig. 10-12. NRZ recording and reading.

any time when the data being read is a 0. Thus if the *A* output of the flip-flop is used as a control signal at one input of an AND gate, while the other input is a clock, the output of the AND gate is an exact replica of the digital data being read. Notice that the clock must be carefully synchronized with the data train from the *read*-head winding. Notice also that the maximum rate of flux changes occurs when recording (or reading) alternate 1s and 0s.

In comparing this with the RZ recording methods, you can see that the NRZ method offers the distinct advantage that the maximum rate of flux changes is only one-half that for RZ recording. Thus the *read/write* heads and associated electronics can have reduced requirements for operation at the same rates, or they are capable of operating at twice the rate for the same specifications.

A variation on this basic form of NRZ recording is shown in Fig. 10-13. This technique is quite often called "non-return-to-zero-inverted," NRZI, since both 1s and 0s are recorded at both the high and low saturation-current levels. The key to this method of recording is that a 1 is sensed whenever there is a flux change, whether it be positive or negative. If the *read*-winding output voltage is amplified and presented to the OR gate as shown in the figure, the output of the gate will be the desired data train. The upper Schmitt trigger is sensitive only to positive pulses, while the lower one is sensitive only to negative pulses. Both outputs of the Schmitt triggers are low until a pulse arrives. At this time the output goes positive for a fixed duration and generates the desired output pulse.

Fig. 10-13. NRZI recording and reading.



10-5 OTHER PERIPHERAL EQUIPMENT

A wide variety of peripheral equipment has been developed for use with digital systems. Only a cursory description of some of the various equipment will be given here, and the reader is encouraged to study equipment of particular interest by consulting the data manuals of the various manufacturers.

One of the simplest means of inputting information into a digital system is by the use of switches. These switches could be push-button, toggle, etc., but the important thing is the fact that they are capable of representing binary information. A row of 10 switches could, for example, be switched to represent the 10 binary bits in a 10-bit word.

Similarly, one of the simplest means of reading data out of a digital system is to put lights on the outputs of the flip-flops in a storage register. Admittedly, this is a rather slow means of communication, since the operator must convert the displayed binary data into something more meaningful. Nevertheless, this represents an inexpensive and practical means of communication between man and machine.

A much more sophisticated method for reading data out of a digital system is by means of a cathode-ray tube. One type of cathode-ray tube used is very similar to the tube used in oscilloscopes, and the operation of the tube is nearly the same. The unit is generally used to display curves representing information which has been processed by the system, and a camera can be attached to some units to photograph the display for a permanent record. The information displayed might be the transient response of an electrical network or a guided-missile trajectory.

A second type of cathode-ray tube for display is called a "charactron." It has the ability to display alphanumeric characters on the face of the screen. This tube operates by shooting an electron beam through a matrix (mask) which has each of the characters cut in it. As the beam passes through the matrix it is shaped in the form of the character through which it passes, and this shaped beam is then focused on the face of the screen. Since the operation of the electron beam is very fast, it is possible to write information on the face of the tube, and the operator can then read the display.

Some tubes of this type which are used in large radar systems have matrices with the proper characters to display map coordinates, friendly aircraft, unfriendly aircraft, etc. The operator thus sees a display of the surrounding area complete with all aircraft, properly designated, in the vicinity. These systems usually have an additional accessory called a "light pen" which enables the operator to input information into the digital system by placing the light pen on the surface of the tube and activating it. The operator can do such things as expand an area of interest, request information on an unidentified flying object, and designate certain aircraft as targets.

A somewhat more common piece of equipment, but nevertheless useful when large quantities of data are being handled, is the printer. Printers are available which will print the output data in straight binary form, octal form, or all the alphanumeric characters. The typical printer has the ability to print information on a 120-space line at rates from a few hundred lines up to over 1,200 lines per minute. The simplest printers are converted, or specially made, electric typewriters

known as "character-at-a-time printers." They are relatively slow and operate at speeds of 10 to 30 characters per second.

A more sophisticated printer is known as the "line-at-a-time printer" since an entire line of 120 characters is printed in one operation. This type of printer is capable of operation at rates of around 250 lines per minute.

Somewhat faster operation is possible with machines which use a print wheel. The print-wheel printer is composed of 120 wheels, one for each position on the line to be printed. These wheels rotate continuously, and when the proper character is under the print position a hammer strikes an inked ribbon against the paper, which contacts the raised character on the print wheel. Wheel printers are capable of operation at the rate of 1,250 lines per minute and have a maximum capacity of 160 characters per line.

One other very important piece of peripheral equipment is the digital plotter. These units are being used more and more in a wide variety of tasks, including automatic drafting, numerical control, production artwork masters (used to manufacture integrated circuits), charts and graphs for management information, maps and contours, biomedical information, and traffic analysis, as well as a host of other applications. A somewhat hybrid form of digital plotting is used when the digital output of a system is converted to analog form (digital-to-analog conversion is the subject of the next chapter) to drive servomotors which position a cursor or pen. A piece of graph paper is positioned on a flat plotting surface, and the pen is caused to move across the paper in response to numbers received from the digital system.

Another digital plotting system, which is more truly a digital plotter, makes use of bidirectional stepping motors to position the pen and thus plot the information on graph paper. In this system, which is known as a "digital incremental plotter," the necessity for digital-to-analog conversion is eliminated, and these systems are usually less expensive and smaller in size. Digital incremental plotters are capable of plotting increments as small as 0.0025 in and offer much greater accuracies than the hybrid model. Furthermore, these plotters are capable of plotting at the rate of 4½ in/s and providing a complete system of annotation and labeling.

10-6 TELETYPEWRITER TERMINALS

The teletypewriter (TTY) is presently one of the most popular *input/output* units. A TTY is an important and versatile link between man and computer, whether the computer is of the small-scale general-purpose type, or a large-scale model used on a time-share basis. It is common practice to use a TTY as a *remote* terminal connected to a large-scale general-purpose computer via telephone lines. The two binary logic levels (1 and 0) used in the TTY and the computer can be represented as two distinct audio frequencies which are then transmitted over telephone lines. An *acoustic tone coupler* is used in conjunction with the TTY to translate data from audio frequencies to logic levels, and vice versa. The central computer can be placed in a convenient site, and access to the computer via a TTY terminal is limited only by the requirement for a telephone line.

A TTY console consists of a basic keyboard for typing in information, and a printing mechanism for printing information output from the computer. Many TTYs are also equipped with a paper-tape punch, and thus either input data or output data can be recorded on punched paper tape.

Most modern TTYs use an eight-hole punched paper tape. There has been an attempt to standardize on an alphanumeric code, and the American Standard Code for Information Interchange (ASCII) is widely used. An eight-hole code has $2^8 = 256$ combinations, sufficient to provide for both uppercase and lowercase alphabets, the 10 numerals, and a number of special characters and control signals. The ASCII code is shown in Table 10-1.

10-7 ENCODING AND DECODING MATRICES

Encoding and decoding matrices are often used to alter the form of the data being entered into or taken out of a system. A decoding matrix is used to decode the binary information in a digital system by changing it into some other number system. For example, in a previous chapter the binary output of a register was decoded into decimal form by means of AND gates, and the decoded output was used to drive nixie tubes. Encoding information is just the reverse process and could, for example, involve changing decimal signals into equivalent binary signals for entry into a digital system.

The most straightforward way of decoding information is simply to construct the necessary AND gates, as was done for the nixie tubes. Decoding in this fashion is quite simple and is most easily accomplished by using the truth table or waveforms for the signals involved. The decoding of a four-flip-flop counter would, for example, require 16 four-input AND gates, since there are 16 possible states determined by the four flip-flops. This type of decoding then requires $n \times 2^n$ diodes, where n is the number of flip-flops, for the complete decoding network.

Example 10-5

Draw the 16 gates necessary to decode a four-flip-flop counter.

Solution

The necessary gates can best be implemented by using a truth table to determine the necessary gate connections. The gates are shown in Fig. 10-14.

There is a second method of decoding which can be used to realize a savings in diodes. This method is referred to as "tree decoding," and it results in a reduction of the number of required diodes by grouping the states to be decoded. Decoding of the four-flip-flop counter discussed in the previous example can be accomplished by separating the counts into four groups. These groups are 0,1,2,3; 4,5,6,7; 8,9,10,11; and 12,13,14,15. Notice that the first group can be distinguished by an AND gate whose output is $\overline{D}\overline{C}$, the second group by $\overline{D}C$, the third group by $D\overline{C}$, and the last group by DC. Each of these four groups can then be divided in half by using B or $\overline{B}$. These eight subgroups can then be further divided into the 16 counts by using A and $\overline{A}$. The complete decoding network is shown in Fig. 10-15.

Table 10-1
THE AMERICAN STANDARD CODE FOR INFORMATION EXCHANGE*

	000	001	010	011	100	101	110	111
0000	NULL	① DC_0	b	0	@	P		
0001	SOM	DC_1	!	1	A	Q		
0010	EOA	DC_2	"	2	B	R		
0011	EOM	DC_3	#	3	C	S		
0100	EOT	DC_4 (Stop)	$	4	D	T		
0101	WRU	ERR	%	5	E	U		
0110	RU	SYNC	&	6	F	V		
0111	BELL	LEM	'	7	G	W		
1000	FE_0	S_0	(	8	H	X	Unassigned	
1001	HT/SK	S_1	)	9	I	Y		
1010	LF	S_2	*	:	J	Z		
1011	V_{TAB}	S_3	+	;	K	[		
1100	FF	S_4	,	<	L	\		ACK
1101	CR	S_5	-	=	M	]		②
1110	SO	S_6	.	>	N	↑		ESC
1111	SI	S_7	/	?	O	←		DEL

Example: 100 0001 = A

b_7 ---------- b_1

The abbreviations used in the figure mean:

NULL	Null idle	CR	Carriage return
SOM	Start of message	SO	Shift out
EOA	End of address	SI	Shift in
EOM	End of message	DC_0	Device control ① Reserved for data Link escape
EOT	End of transmission	$DC_1 - DC_3$	Device control
WRU	"Who are you?"	ERR	Error
RU	"Are you . . .?"	SYNC	Synchronous idle
BELL	Audible signal	LEM	Logical end of media
FE	Format effector	$SO_0 - SO_7$	Separator (information)
HT	Horizontal tabulation		Word separator (blank, normally non-printing)
SK	Skip (punched card)	ACK	Acknowledge
LF	Line feed	②	Unassigned control
VTAB	Vertical tabulation	ESC	Escape
FF	Form feed	DEL	Delete idle

* Reprinted from *Digital Computer Fundamentals* by Thomas C. Bartee. Copyright 1960, 1966 by McGraw-Hill, Inc. Used with permission of McGraw-Hill Book Company.

D	C	B	A	Count
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15
0	0	0	0	0

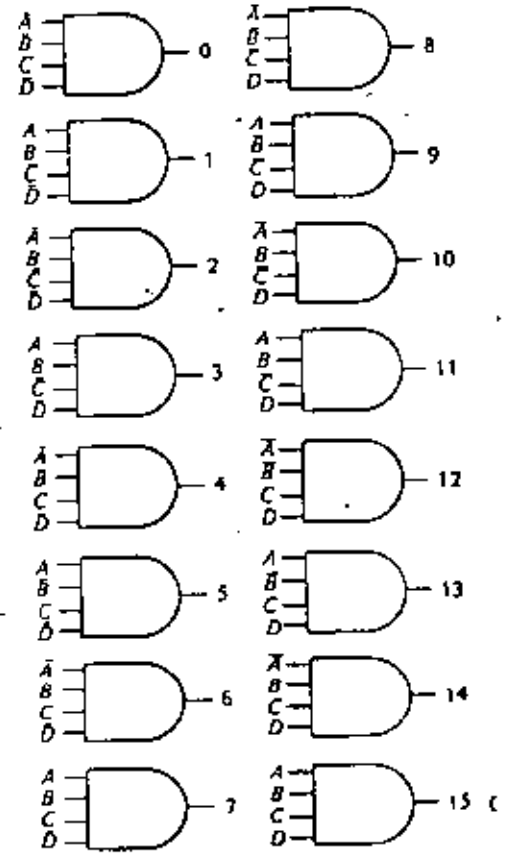


Fig. 10-14. Four-flip-flop counter decoding.

A saving of 8 diodes has been achieved, since the previous decoding scheme required 64 diodes and this method only requires 56. The saving in diodes here is not very spectacular, but the construction of a matrix in this manner to decode five flip-flops would result in a saving of 40 diodes. As the number of flip-flops to be decoded increases, the saving in diodes increases very rapidly.

This type of decoding matrix does have the disadvantage that the decoded signals must pass through more than one level of gates (in the previous method the signal passes through only one gate). The output signal level may therefore suffer considerable reduction in amplitude. Furthermore, there may be a speed limitation due to the number of gates through which the decoded signals must pass.

A third type of decoding network is known as a "balanced multiplicative decoder." This always results in the minimum number of diodes required for the decoding process. The idea is much the same as a tree decoder, since the counts to be decoded are divided into groups. However, in this system the flip-flops to be decoded are divided into groups of two, and the results are then combined to give

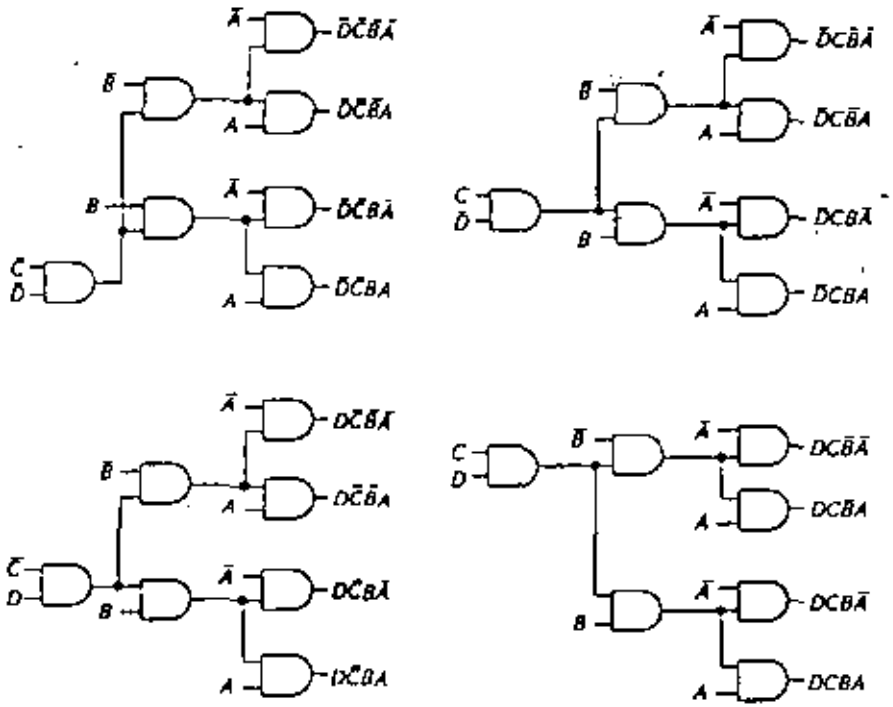
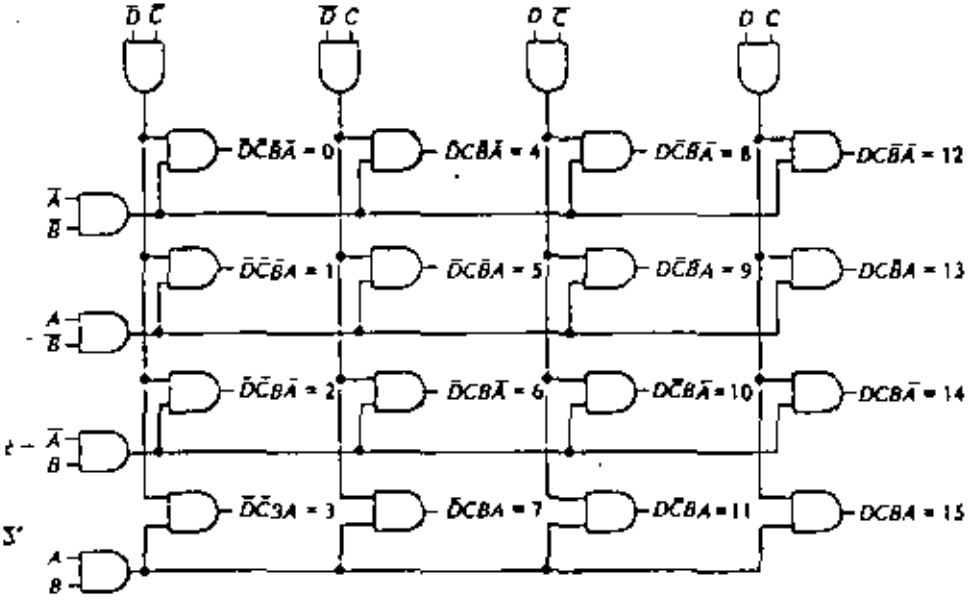


Fig. 10-15. Tree decoding matrix.

Fig. 10-16. Balanced multiplicative decoder.

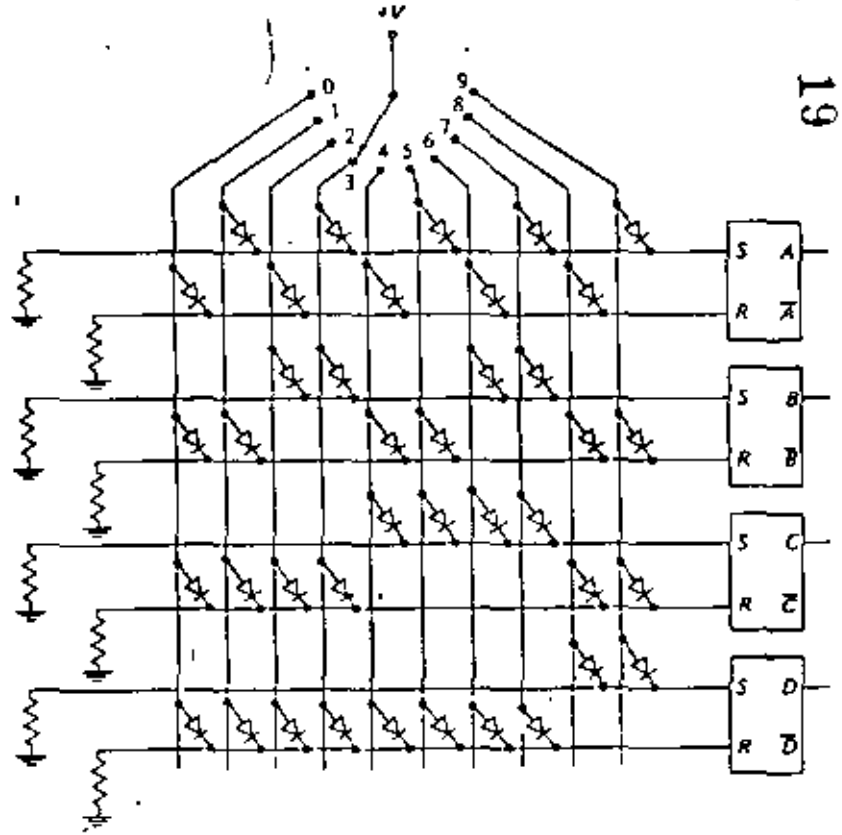


the desired output signals. To decode the four flip-flops discussed previously, four groups are formed by combining flip-flops C and D just as before. In addition, flip-flops B and A are combined in a similar arrangement. The outputs of these eight gates are then combined in 16 AND gates to form the 16 output signals. The results are shown in Fig. 10-16. It can be seen that a total of 48 diodes are required; a saving of 16 diodes is then realized over the first method, while a saving of 8 diodes is realized over the tree method. This scheme again has the same disadvantages of signal-level degradation and speed limitation as the tree decoder.

Encoding a number is just the reverse of decoding. One of the simplest examples of encoding would be the use of a thumb-wheel switch (a 10-position switch) which is used to enter data into a digital system. The operator can set the switch to any one of 10 positions which represent decimal numbers. The output of the switch is then transformed by a proper encoding matrix which changes the decimal number to an equivalent binary number.

An encoding matrix which changes a decimal number to an equivalent binary number and stores it in a register is shown in Fig. 10-17. Setting the switch to a

Fig. 10-17. Decimal encoding matrix.



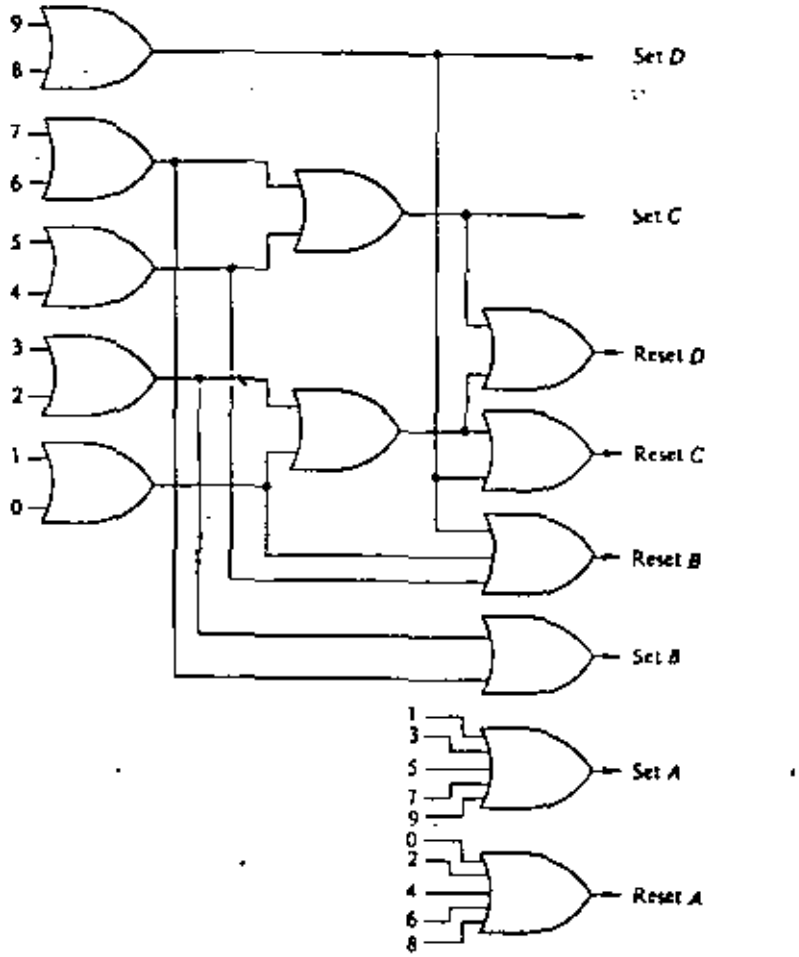


Fig. 10-18. Another decimal encoding matrix.

position places a positive voltage on the line connected to that position. Notice that the R and S input to each flip-flop is essentially the output of an OR gate.

For example, if the switch is set to position 1, the diodes connected to that line have a positive voltage on their plates (they are therefore forward-biased). Thus the set input to flip-flop A goes high while the reset inputs to flip-flops B, C, and D go high. This sets the binary number 0001 in the flip-flops, where A is the least significant bit. Notice that this encoding matrix requires 40 diodes. As might be expected, it is possible to reduce the number of diodes required by combining the input functions as was done with decoding matrices. One method of doing this is shown in Fig. 10-18; it represents a saving of 7 diodes, since this scheme requires only 33 diodes.

Any encoder or decoder can be constructed from basic gates as shown in this section, and when only one or two functions are needed this may provide the best technique. However, as shown in Chap. 3, many of the more common decoding functions are available as MSI ICs. Examples are the 7441 (or 74141) BCD-to-decimal decoder driver, the 7443 excess-3-to-decimal decoder, the 7446 BCD-to-seven-segment decoder driver, and the 74145 1-of-10 decoder driver. There are numerous others, and you are urged to consult manufacturers' data sheets for specific information.

There are also a few encoders available as MSI ICs—for example, the Fairchild 9318 eight-input priority encoder. This unit accepts eight inputs and produces a binary weighted code of the highest-order output. Again, you should consult specific manufacturers' data sheets for detailed information on encoders.

STUDY AIDS

Summary

Punched cards provide one of the most useful and widely used media for storing binary information. Each card is considered as a block or unit of information and is therefore referred to as a "unit record." Furthermore, punched-card equipment (punches, sorters, readers, etc.) is commonly called "unit-record equipment."

Alphanumeric information, as well as special characters, can be punched into cards by means of a code. The most common code in use is the Hollerith code.

A similar medium for information storage is punched paper tape. Alphanumeric and special characters are recorded by perforating the tape according to a code. There are a number of codes, but the one most commonly used is the eight-hole code. A perforated role of paper tape is a continuous record and is thus distinct from the unit record (punched card).

For handling large quantities of information, magnetic tape is a most convenient recording medium. Magnetic tape offers the advantages of much higher processing rate and much greater recording densities. Moreover, magnetic tape can be erased and used over and over.

The three most common methods for recording on magnetic tape are the return-to-zero (RZ), the non-return-to-zero (NRZ), and the non-return-to-zero-inverted (NRZI). The NRZ and NRZI methods effectively erase or clean the tape automatically during the record operation and thus eliminate one of the problems of RZ recording. These two methods also lend themselves to higher recording rates.

Encoding and decoding matrices form an important part of input-output equipment. These matrices are generally used to change information from one form to another, for example, binary to octal, or binary to decimal, or decimal to binary.

There is a wide variety of digital peripheral equipment including unit-record equipment, printers, cathode-ray-tube displays, and plotters. The choice of peripheral equipment to be used with any system is a major engineering decision. The decision involves establishing the system requirements, studying the available equipment, meeting with the equipment manufacturers, and then making the decision based on operational characteristics, delivery time, and cost.

Glossary

alphanumeric information Information composed of the letters of the alphabet, the numbers, and special characters.
bit One binary digit.
character A number, letter, or symbol represented by a combination of bits.
decoding matrix A matrix used to alter the format of information taken from the output of a system.
encoding matrix A matrix used to alter the format of information being entered into a system.
Hollerith code The system for representing information by punching holes in a prescribed manner in a punched card.
interrecord gap A blank piece of tape between recorded information.
NRZ Non-return-to-zero recording.
NRZI Non-return-to-zero inverted recording.
parity The method of using an additional punched hole (or magnetic spot for magnetic recording) to ensure that the total number of holes (or spots) for each character is even or odd.
recording density The number of characters recorded per inch of tape.
tape-utilization factor The ratio of the number of characters actually recorded to the maximum number of characters that could be recorded.
unit record A punched card represents a unit record since each card contains a unit or block of information.

Review Questions

1. Describe some of the problems of the man-machine interface.
2. Describe a typical punched card (size, number of columns, number of rows).
3. Which rows are the zone punches on a punched card?
4. Which rows are the digit punches on a punched card?
5. What is the Hollerith code? What does "JR is 11" signify?
6. How is binary information represented on a card; i.e., what does a hole represent, and what does the absence of a hole represent?
7. What is the meaning of unit record?
8. Name three pieces of unit-record peripheral equipment, and give a brief description of how they are used.
9. Describe the eight-hole code used to punch information into paper tape.
10. Describe how 1s and 0s are recorded on magnetic tape by means of a magnetic record head.
11. How is alphanumeric information recorded on magnetic tape?
12. How is binary information recorded on magnetic tape?
13. Explain the dual-parity system used in magnetic-tape recording.

14. What is the purpose of an interrecord gap on magnetic tape?
15. How can the tape-utilization factor be used to determine the total number of characters stored on a magnetic tape?
16. Describe the operation of the RZ recording method. What are some of the difficulties with this system?
17. Describe the operation of the NRZ recording method. What advantages does this method offer over RZ recording?
18. Describe the NRZI recording technique.
19. Why is a digital incremental plotter a true digital plotting system?
20. What is the difference between an encoding and a decoding matrix?

Problems

10-1. Make a sketch of a punched card and code your name, address, and social security number using the Hollerith code. Use a dark spot to represent a hole.

10-2. Change your social security number to the equivalent binary number. Make a sketch of a punched card, and record this number on the card in the horizontal binary fashion.

10-3. Repeat Prob. 10-2, but record the number on the card in the vertical fashion.

10-4. Assume that alphanumeric information is being punched into cards at the rate of 250 cards per minute. If the cards have an average of 65 characters each, at what rate in characters per second is the information being processed?

10-5. Make a sketch of a length of paper tape. Using the eight-hole code, record your name, address, and social security number on the tape. Use a dark spot to represent a hole.

10-6. What length of paper tape is required for the storage of 60,000 characters of alphanumeric information using the eight-hole code? Assume no record gaps.

10-7. What length of magnetic tape would be required to store the information in Prob. 10-6 if the recording density is 500 bits per inch? Assume no record gaps.

10-8. Assume that data are recorded on magnetic tape at a density of 200 bits per inch. If the record length is 200 characters, and the interrecord gap is 0.75 in, what is the tape-utilization factor? Using this scheme, how many characters can be stored in 1,000 ft of tape?

10-9. Verify the solution to Prob. 10-8 above by using Eq. (10-3). Notice that the 2,400 in the equation must be replaced by 1,000, since this is the tape length.

10-10. Repeat Probs. 10-8 and 10-9 for a density of 800 bits per inch.

10-11. What length of magnetic tape is required to store 10^8 characters recorded at a density of 800 bits per inch with a record length of 500 characters?

11-3. Verify the voltage-output levels for the network of Fig. 11-5 using Millman's theorem. Draw the equivalent circuits.

11-4. Assume the divider in Prob. 11-2 has +10 V full-scale output, and find the following:
(a) The change in output voltage due to a change in the LSB.
(b) The output voltage for an input of 110110.

11-5. A 10-bit resistive divider is constructed such that the current through the LSB resistor is 100 μA. Determine the maximum current that will flow through the MSB resistor.

11-6. What is the full-scale output voltage of a six-bit binary ladder if 0 = 0 V and 1 = +10 V? What is it for an eight-bit ladder?

11-7. Find the output voltage of a six-bit binary ladder with the following inputs:
(a) 101001.
(b) 111011.
(c) 110001.

11-8. Check the results of Prob. 11-7 by adding the individual bit contributions.

11-9. What is the resolution of a 12-bit D/A converter which uses a binary ladder? If the full-scale output is +10 V, what is the resolution in volts?

11-10. How many bits are required in a binary ladder to achieve a resolution of 1 mV if full scale is +5 V?

11-11. How many comparators are required to build a five-bit simultaneous A/D converter?

11-12. Redesign the encoding matrix and read gates of Fig. 11-20 using NAND gates.

11-13. Find the following for a 12-bit counter-type A/D converter using a 1-MHz clock:
(a) Maximum conversion time.
(b) Average conversion time.
(c) Maximum conversion rate.

11-14. What clock frequency must be used with a 10-bit counter-type A/D converter if it must be capable of making at least 7,000 conversions per second?

11-15. What is the conversion time of a 12-bit successive-approximation-type A/D converter using a 1-MHz clock?

11-16. What is the conversion time of a 12-bit section-counter-type A/D converter using a 1-MHz clock? The counter is divided into three equal sections.

11-17. What overall accuracy could you reasonably expect from a 12-bit A/D converter?

11-18. What degree of resolution can be obtained using a 12-bit optical encoder?

11-19. Redesign the Gray-to-binary encoder in Fig. 11-32 using NAND gates.

11-20. Redesign the Gray-to-binary encoder in Fig. 11-32 using exclusive-OR gates.

Magnetic Devices and Memories

12

There is a large class of devices and systems which are useful as digital elements because of their magnetic behavior. A ferromagnetic material can be magnetized in a particular direction by the application of a suitable magnetizing force (a magnetic flux resulting from a current flow). The material remains magnetized in that direction after the removal of the excitation. Application of a magnetizing force of the opposite polarity will switch the material, and it will remain magnetized in the opposite direction after removal of the excitation. Thus the ability to store information in two different states is available, and a large class of binary elements has been devised using these principles. In this chapter we investigate a number of these devices and systems that make use of them.

After studying this chapter you should be able to

1. Illustrate how magnetic cores are used to store binary information.
2. Explain the fundamental principles of a coincident-current memory.
3. Describe the operation of a semiconductor memory using either bipolar or MOS devices.

12-1 MAGNETIC CORES

One of the most widely used magnetic elements is the magnetic core. The typical core is toroidal (doughnut-shaped), as shown in Fig. 12-1, and is usually constructed in one of two ways. The metal-ribbon core is constructed by winding a very thin metallic ribbon on a ceramic-core form. A popular ribbon is ½-mil-thick 4-79 molybdenum-permaloy (known as ultrathin ribbon), and a typical core might consist of 20 turns of this ribbon wound on a 0.2-in-diameter ceramic form.

Ferrite cores are constructed from a finely powdered mixture of magnetite, various bivalent metals such as magnesium or maganese, and a binder material. The powder is pressed into the desired shape and fired. During firing, the powder is fused into a solid, homogeneous, polycrystalline form. Ferrite cores such as this are commonly constructed with 50 mil outside diameters and 30 mil inside diameters.

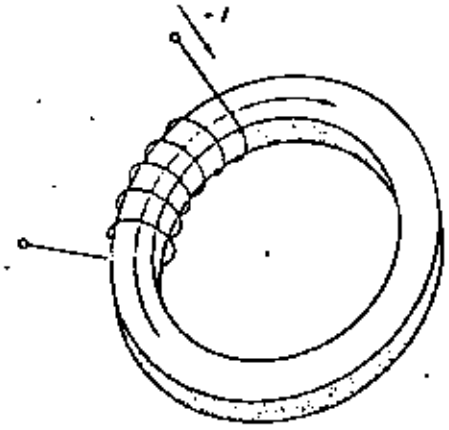


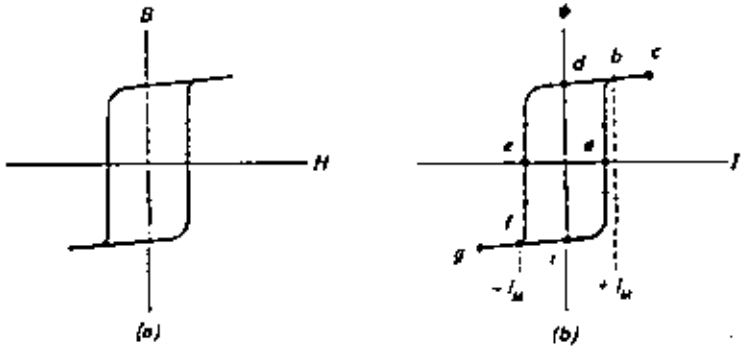
Fig. 12-1. Magnetic core.

Ferrite cores can be constructed in smaller dimensions than metal-ribbon cores and usually have better uniformity and lower cost. Furthermore, ferrite cores typically have resistivities greater than 10^4 Ω-cm, which means eddy-current losses are negligible and thus core heating is reduced. For these reasons, they are widely used as the principal memory or storage elements in large-scale digital computers.

Metal-ribbon cores, on the other hand, have very good magnetic characteristics and generally require a smaller driving current for switching. They are somewhat better for the construction of logic circuits and shift registers.

The binary characteristics of a core can be most easily seen by examining the *hysteresis curve* for a typical core. Hysteresis comes from the Greek word *hysterein*, which means to lag behind. A magnetic core exhibits a lag-behind characteristic in the hysteresis curve shown in Fig. 12-2a. In this figure, the *magnetic flux density* B is plotted as a function of the *magnetic force* H. However, since the flux density B is directly proportional to the flux ϕ, and since the magnetic field H is directly proportional to the current I producing it, a plot of ϕ versus I is a curve of the same

Fig. 12-2 Ferrite-core hysteresis curves. (a) Magnetic flux density B versus magnetic field H. (b) Magnetic flux ϕ versus current I.



general shape. A plot of flux in the core ϕ versus driving current I is shown in Fig. 12-2*b*. We shall base our discussion on this curve since it is generally easier to talk in terms of these quantities.

Now, suppose that a current source is attached to the windings on the core shown in Fig. 12-1, and a positive current is applied (current flows into the upper terminal of the winding). This creates a flux in the core in the clockwise direction shown in the figure (remember the *right-hand rule*). If the drive current is just slightly greater than I_m shown in Fig. 12-2, the operating point of the core is somewhere between points *b* and *c* on the ϕI curve. The magnitude of the flux can then be read from the ϕ axis in this figure.

If the drive current is now removed, the operating point moves along the ϕI curve through point *b* to point *d*. The core is now storing energy with no input signal, since there is a remaining or *remanent* flux in the core at this point. This property is known as *remanence*, and this point is known as a *remanent point*.

The repeated application of positive current pulses simply causes the operating point to move between points *d* and *c* on the ϕI curve. Notice that the operating point always comes to rest at point *d* when all drive current is removed.

If a negative drive current somewhat greater than $-I_m$ is now applied to the winding (in a direction opposite to that shown in Fig. 12-1), the operating point moves from *d* down through *e* and stops at a point somewhere between *f* and *g* on the ϕI curve. At this point the flux has switched in the core and is now directed in a counterclockwise direction in Fig. 12-1. If the drive current is now removed, the operating point comes to rest at point *h* on the ϕI curve of Fig. 12-2. Notice that the flux has approximately the same magnitude but is the negative of what it was previously. This indicates that the core has been magnetized in the opposite direction.

Repeated application of negative drive currents will simply cause the operating point to move between points *g* and *h* on the ϕI curve, but the final resting place with no applied current will be point *h*. Point *h* then represents a second remanent point on the ϕI curve.

By way of summary, a core has two remanent states: point *d* after the application of one or more positive current pulses, point *h* after the application of one or more negative current pulses. For the core in Fig. 12-1, point *d* corresponds to the core magnetized with flux in a clockwise direction, and point *h* corresponds to magnetization with flux in the counterclockwise direction.

Example 12-1

Cores can be magnetized by utilizing the magnetic field surrounding a current-carrying wire by simply *threading* the cores on the wire. For the two possible current directions in the wire shown in Fig. 12-3, what are the corresponding directions of magnetization for the core?

Solution

According to the right-hand rule, a current of $+I$ magnetizes the core with the flux in a clockwise direction around the core. A current of $-I$ magnetizes the core with flux in a counterclockwise direction around the core.

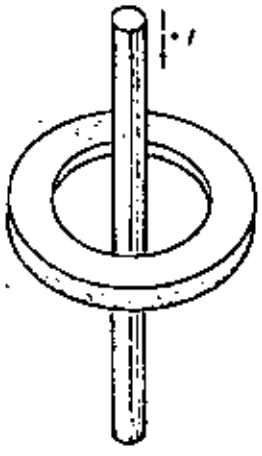


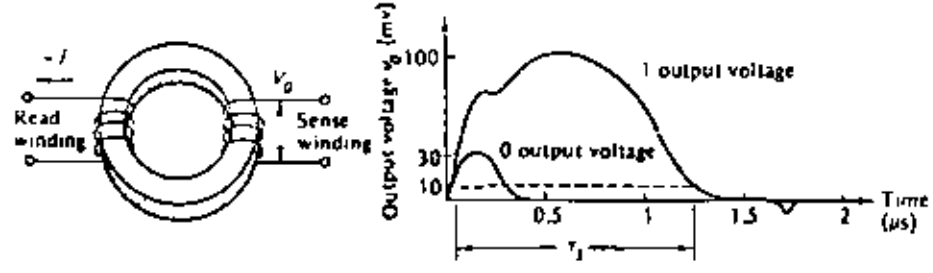
Fig. 12-3.

It is now quite easy to see how a magnetic core is used as a binary storage device in a digital system. The core has two states, and we can simply define one of the states as a 1 and the other state as a 0. It is perfectly arbitrary which is which, but for discussion purposes let us define point *d* as a 1 and point *h* as a 0. This means that a positive current will record a 1 and result in clockwise flux in the core in Fig. 12-1. A negative current will record a 0 and result in a counterclockwise flux in the core.

We now have the means for recording or writing a 1 or a 0 in the core but we do not as yet have any means of detecting the information stored in the core. A very simple technique for accomplishing this is to apply a current to the core which will switch it to a known state and detect whether or not a large flux change occurs. Consider the core shown in Fig. 12-4. Application of a drive current of $-I$ will switch the core to the 0 state. If the core has a 0 stored in it, the operating point will move between points *g* and *h* on the ϕI curve (Fig. 12-2), and a very small flux change will occur. This small change in flux will induce a very small voltage across the sense-winding terminals. On the other hand, if the core has a 1 stored in it, the operating point will move from point *d* to point *h* on the ϕI curve, resulting in a much larger flux change in the core. This change in flux will induce a much larger voltage in the sense winding, and we can thus detect the presence of a 1.

To summarize, we can detect the contents of a core by applying a *read* pulse which resets the core to the 0 state. The output voltage at the sense winding is

Fig. 12-4. Sensing the contents of a core.



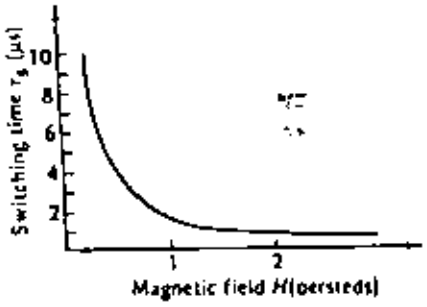


Fig. 12-5. Magnetic-core switching time characteristics.

much greater when the core contains a 1 than when it contains a 0. We can therefore detect a 1 by distinguishing between the two output-voltage signals. Notice that we could set the core by applying a *read* current of $+I$ and detect the larger output voltage at the sense winding as a 0.

The output voltage appearing at the sense winding for a typical core is also shown in Fig. 12-4. Notice that there is a difference of about 3 to 1 in output-voltage amplitude between a 1 and a 0 output. Thus a 1 can be detected by using simple amplitude discrimination in an amplifier. In large systems where many cores are used on common windings (such as the large memory systems in digital computers) the 0 output voltage may become considerably larger because of additive effects. In this case, amplitude discrimination is quite often used in combination with a strobing technique. Even though the amplitude of the 0 output voltage may increase because of additive effects, the width of the output will not increase appreciably. This means that the 0 output-voltage signal will have decayed and will be very small before the 1 output voltage has decayed. Thus if we strobe the *read* amplifiers some time after the application of the *read* pulse (for example, between 0.5 and 1.0 μs in Fig. 12-4), this should improve our detection ability.

The *switching time* of the core is commonly defined as the time required for the output voltage to go from 10 percent up through its maximum value and back down to 10 percent again (see Fig. 12-4). The switching time for any one core is a function of the drive current as shown in Fig. 12-5. It is evident from this curve that an increased drive current results in a decreased switching time. In general, the switching time for a core depends on the physical size of the core, the type of core, and the materials used in its construction, as well as the manner in which it is used. It will be sufficient for our purposes to know that cores are available with switching times from around 0.1 μs up to milliseconds, with drive currents of 100 mA to 1 A.

12-2 MAGNETIC-CORE LOGIC

Since a magnetic core is a basic binary element, it can be used in a number of ways to implement logical functions. Because of its inherent ruggedness, the core is a particularly useful logical element in applications where environmental extremes are experienced, for example, the temperature extremes and radiation exposure experienced by space vehicles.

Since the core is essentially a storage device and its content is ...ed by reset- ...ting the core to the 0 state, any logic system using cores mus... ...ssarily be a

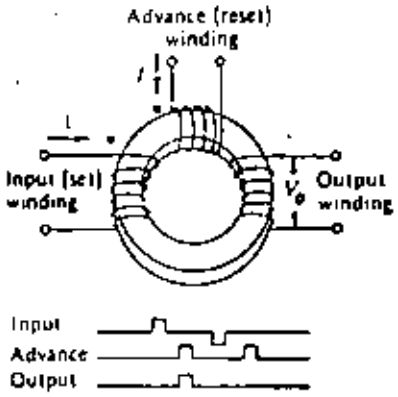


Fig. 12-6. Basic magnetic-core logic element.

dynamic system. The basis for using the core as a logical element is shown in Fig. 12-6. A 1 input to the core is represented by a current of $+I$ at the *input* winding; this sets a 1 in the core (magnetizes it in a clockwise direction). An *advance* pulse occurs sometime after the *input* pulse has disappeared. Logical operations are carried out during the time the *advance* pulse appears at the advance (*reset*) winding. At this time the core is forced into the 0 state and a pulse appears at the output winding only if the core previously stored a 1. The current in the output winding can then be used as the input for other cores or other logical elements.

There is some energy loss in the core during switching. For this reason, the output winding normally has more turns than either the *input* or *advance* windings, so that the output will be capable of driving one or more cores.

Notice that a 0 can be set in the core by application of a current of $-I$ at the *input* winding. Alternatively, a 0 could be stored by a current of $+I$ into the undotted side of the *input* winding. The important thing to notice is that either a 1 or a 0 can be stored in the core by application of a current to the proper terminal of the *input* winding.

To simplify our discussion and the logic diagrams, we shall adopt the symbols for the core and its windings shown in Fig. 12-7. A pulse at the 1 input sets a 1 in the core; a pulse at the 0 input sets a 0 in the core; during the *advance* pulse, a pulse appears at the output only if the core previously held a 1. Let us now consider some of the basic logic functions using the symbol shown in Fig. 12-7*b*.

A method for implementing the OR function is shown in Fig. 12-8*a*. A current pulse at either the X or Y inputs sets a 1 in the core. Sometime after the input pulse(s) have been terminated, an *advance* pulse occurs. If the core has been set to the 1 state, a pulse appears at the output winding. Notice that this is truly an OR function since a pulse at either the X or Y input or both sets a 1 in the core.

The method shown in Fig. 12-8*b* provides the means for obtaining the complement of a variable. The *set input* winding to the core has a 1 input. This means that during the *input* pulse time this winding always has a *set input* current. If there is no current at the X input (signifying $X = 0$), the core is set. Then, when the *advance* pulse occurs, a 1 appears at the output, signifying that $\bar{X} = 1$. On the other hand, if

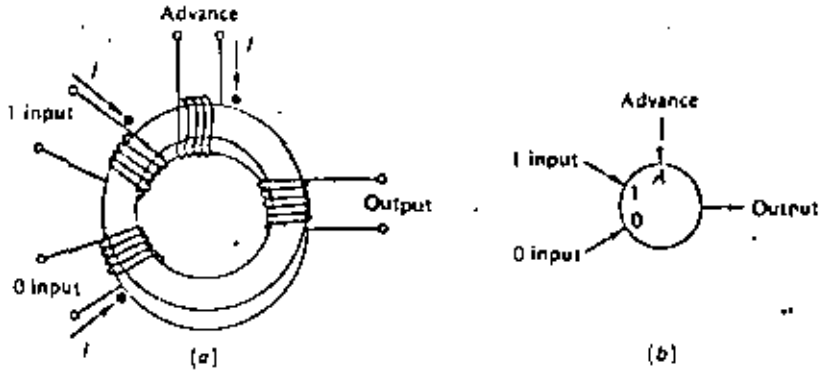
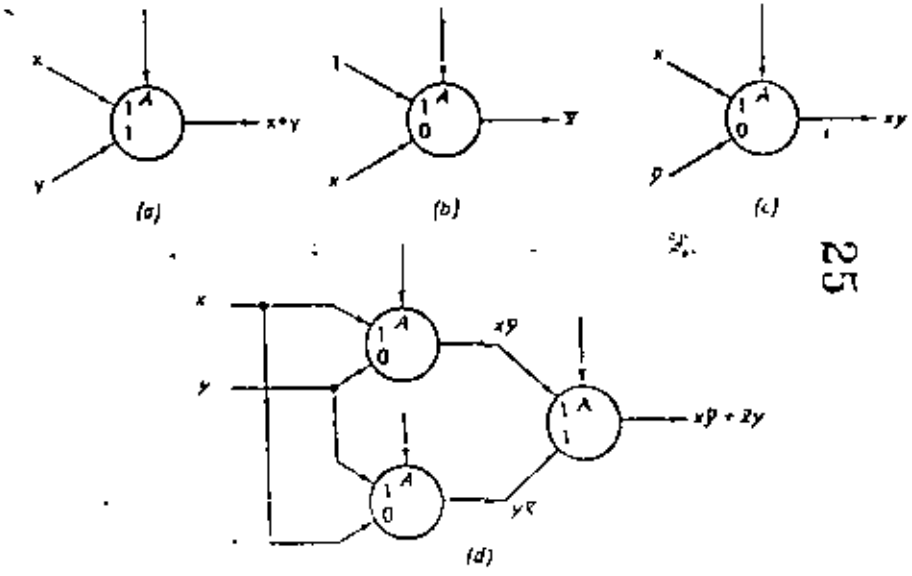


Fig. 12-7. Magnetic-core logic element. (*a*) Core windings. (*b*) Logic symbol.

$X = 1$, a current appears at the X input during the set time, and the effects of the X input current and the 1 input current cancel one another. The core then remains in the reset state (recall that the core is reset during the *advance* pulse). In this case no pulse appears at the output during the *advance* pulse since the core previously contained a 0. Thus the output represents $\bar{X} = 0$.

The AND function can be implemented using a core as shown in Fig. 12-8*c*. The two inputs to the core are X and $\bar{Y}$, and there are four possible combinations of these two inputs. Let's examine these input combinations in detail.

Fig. 12-8. Basic core logic functions. (*a*) OR. (*b*) Complement. (*c*) AND. (*d*) Exclusive-OR.



1. $X = 0$, $Y = 0$. Since $X = 0$, the core cannot be set. Since $Y = 0$, $\overline{Y} = 1$ and the core will then be reset. Thus this input combination resets the core and it stores a 0.
2. $X = 0$, $Y = 1$. Since $X = 0$, the core still cannot be set. $Y = 1$ and therefore $\overline{Y} = 0$. In this input combination, there is no input current in either winding and the core cannot change state. Thus the core remains in the 0 state because of the previous *advance* pulse.
3. $X = 1$, $Y = 0$. The current in the X *winding* will attempt to set a 1 in the core. However, $\overline{Y} = 1$ and this current will attempt to reset the core. These two currents offset one another, and the core does not change states. It remains in the 0 state because of the previous *advance* pulse.
4. $X = 1$, $Y = 1$. The current in the X *winding* will set a 1 in the core since $\overline{Y} = 0$ and there is no current in the Y *winding*. Thus this combination stores a 1 in the core.

In summary, the input X AND $\overline{Y}$ is the only combination which results in a 1 being stored in the core. Thus this is truly an AND function.

An exclusive-OR function can be implemented as shown in Fig. 12-8*d* by ORing the outputs of two AND-function cores.

Example 12-2

Make a truth table for the exclusive-OR function shown in Fig. 12-8*d*.

Solution

X	Y	$X\overline{Y}$	$\overline{X}Y$	$X\overline{Y} + \overline{X}Y$
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	0	0	0

One of the major problems of core logic becomes apparent in the operation of the exclusive-OR shown in Fig. 12-8*d*. This is the problem of the time required for the information to shift down the line from one core to the next. For the exclusive-OR, the inputs X and Y appear at time t_1, and the AND cores are *set* or *reset* at this time. At time t_2 an *advance* pulse is applied to the AND cores and their outputs are used to set the OR core. Then at time t_3 an *advance* pulse is applied to the OR core and the final output appears. It should be obvious from this discussion that the operation time for more complicated logic functions may become excessively long.

A second difficulty with this type of logic is the fact that the *input* pulses must be of exactly the same width. This is particularly true for functions such as the COMPLEMENT and the AND, since the input signals are at times required to cancel one another. It is apparent that if one of the input signals is wider than the other, the core may contain erroneous data after the *input* pulses have disappeared.

You will recall that in order to switch a core from one state to another a cer minimum current I_m is required. This is sometimes referred to as the *select cur* The core arrangement shown in Fig. 12-8a can be used to implement an AND f tion if the X and Y inputs are each limited to one-half the select current $\frac{1}{2}I_m$. In way, the only time the core can be set is when both X and Y are present, since is the only time the core receives a full select current I_m. Core logic functions be constructed using the half-select current idea. This idea is quite importar forms the basis of one type of large-scale memory system which we discuss lat this chapter.

12-3 MAGNETIC-CORE SHIFT REGISTER

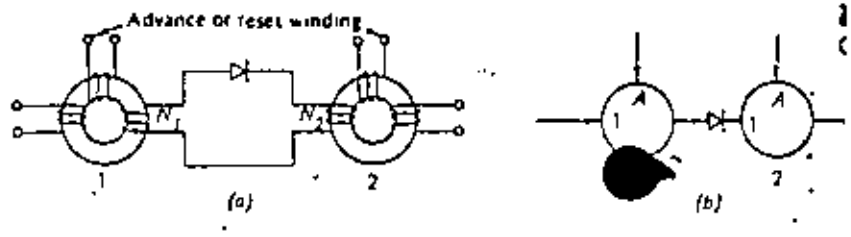
A review of the previous section will reveal that a magnetic core exhibits at l two of the major characteristics of a flip-flop: first, it is a binary device capabl storing binary information; second, it is capable of being set or reset. Thus it w seem reasonable to expect that the core could be used to construct a shift reg or a *ring* counter. Cores are indeed frequently used for these purposes, and in section we consider some of the necessary precautions and techniques.

The main idea involves connecting the output of each core to the input of next core. When a core is reset (or set), the signal appearing at the output of core is used to set (or reset) the next core. Such a connection between two c called a "single-diode transfer loop," is shown in Fig. 12-9.

There are three major problems to overcome when using the single-d transfer loop. The first problem is the gain through the core. This is similar to problem discussed previously, and the solution is the same. That is, the losse signal through the core can be overcome by constructing the output winding more turns than the input winding. This ensures that the output signal will have ficient amplitude to switch the next core.

The second problem concerns the polarity of the output signal. A signal app at the output when the core is set or when the core is reset. These two signals opposite polarities, and either is capable of switching the next core. In general, desirable that only one of the two output signals be effective, and this ca achieved by the use of the diode shown in Fig. 12-9. In this figure, the cu produced in the output winding will go through the diode in the forward dire (and thus set the next core) when the core is reset from the 1 state to the 0 state

Fig. 12-9. Single-diode transfer loop. (a) Circuit. (b) Symbolic representation.



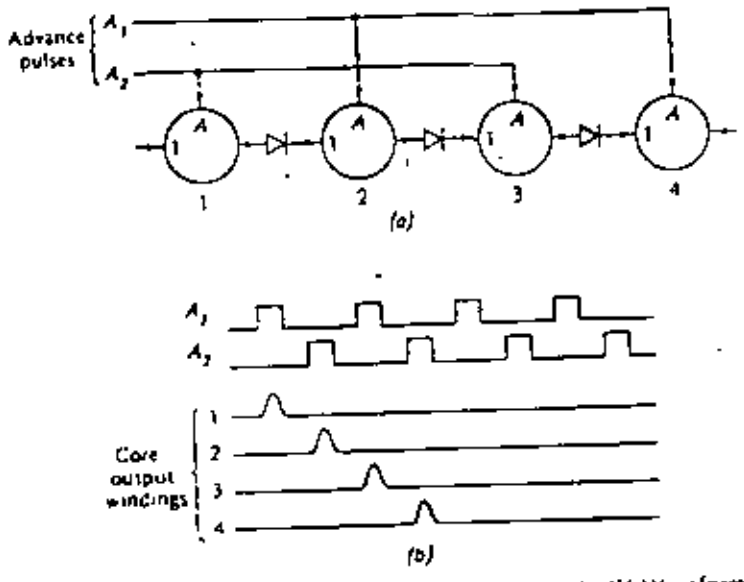


Fig. 12-10. Four-core shift register. (a) Symbolic circuit. (b) Waveforms.

the other hand, when the core is being set to the 1 state, the diode will prevent current flow in the output and thus the next core cannot be switched. Notice that the opposite situation could be realized by simply reversing the diode.

The third problem arises from the fact that resetting core 2 induces a current in winding N_3 which will pass through the diode in the forward direction and thus tend to set a 1 in core 1. This constitutes the transfer of information in the reverse direction and is highly undesirable. Fortunately, the solution to the first problem (that of gain) results in a solution for this problem as well. That is, since N_3 has fewer windings than N_1, this reverse signal will not have sufficient amplitude to switch core 1. With this understanding of the basic single-diode transfer loop, let us investigate the operation of a simple core shift register.

A basic magnetic-core shift register in symbolic form is shown in Fig. 12-10. Two sets of advance windings are necessary for shifting information down the line. The advance pulses occur alternately as shown in the figure. A_1 is connected to cores 1 and 3 and would be connected to all *odd*-numbered cores for a larger register. A_2 is connected to cores 2 and 4 and would be connected to all even-numbered cores. If we assume that all cores are reset with the exception of core 1, it is clear that the advance pulses will shift this 1 down the register from core to core until it is shifted "out the end" when core 4 is reset. The operation is as follows: the first A_1 pulse resets core 1 and thus sets core 2. This is followed by an A_2 pulse which resets core 2 and thus sets core 3. The next A_1 pulse resets core 3 and sets core 4, and the following A_2 pulse shifts the 1 "out the end" by resetting core 4. Notice that the two phases of advance pulses are required, since it is not possible to set a core while an advance (or reset) pulse is present.

The output of each core winding can be used as an input to an amplifier to

produce the waveforms shown in Fig. 12-10*b*. Notice that after four advance pulses the 1 has been shifted completely through the register, and the output lines all remain low after this time.

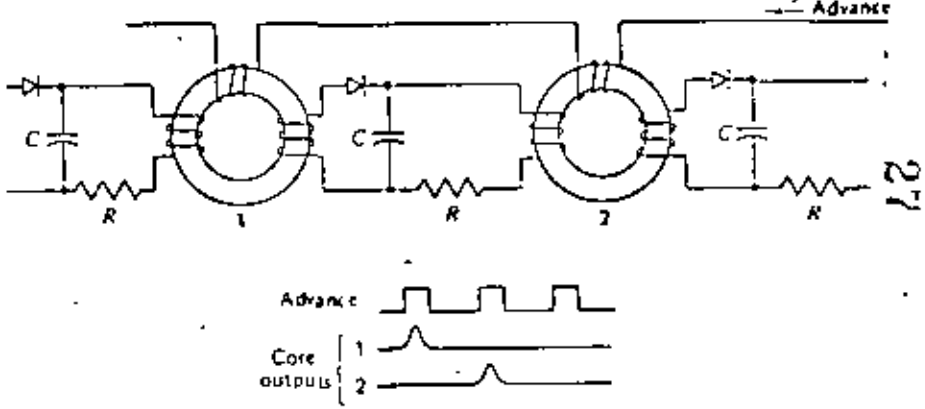
The need for a two-phase clock or advance pulse system could be eliminated if some delay were introduced between the output of each core and the input of the next core. Suppose that a delay greater than the width of the advance pulses were introduced between each pair of cores. In this case, it would be possible to drive every core with the same advance pulse since the output of any core could not arrive at the input to the next core until after the advance pulse had disappeared.

One method for introducing a delay between cores is shown in Fig. 12-11. The advance-pulse amplitude is several times the minimum required to switch the cores and will reset all cores to the 0 state. If a core previously contained a 0, no switching occurs and thus no signal appears at the output winding. On the other hand, if a core previously contained a 1, current flows in the output winding and charges the capacitor. Some current flows through the set winding of the next core, but it is small because of the presence of the resistor; furthermore, it is overridden by the magnitude of the advance pulse. However, at the cessation of the advance pulse, C remains charged. Thus C discharges through the input winding and *R*, and sets core 2 to the 1 state.

In this system, the amplitude of the advance pulses is not too critical, but the width must be matched to the *RC* time constant of the loop. If the advance pulses are too long, or alternatively if the RC time constant is too short, the capacitor will discharge too much during the advance pulse time and will be incapable of setting the core at the cessation of the advance pulse. The *RC* time constant may limit the upper frequency of operation; it should be noted, however, that resetting a core induces a current in its input winding in a direction which tends to discharge the capacitor.

The arrangements we have discussed here are called *one-core-per-bit* registers. There are numerous other methods (too many to discuss here) for implementing

Fig. 12-11. Core shift register using a capacitor for delay between cores.



registers and counters, and the reader is referred to the references for more advanced techniques. Some of the other methods include *two-core-per-bit* systems, *modified-advance-pulse* systems, *modified-winding-core* systems, *split-winding-core* systems, and *current-routing-transfer* systems.

Example 12-3

Using core symbols and the capacitor-delay technique, draw the diagram for a four-stage ring counter. Show the expected waveforms.

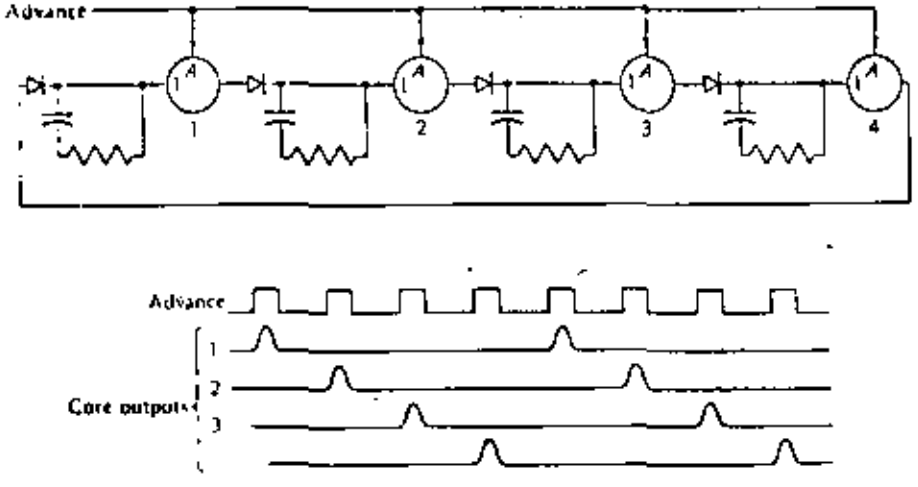
Solution

A ring counter can be formed from a simple shift register by using the output of the last core as the input for the first core. Such a system, along with the expected waveforms, is shown in Fig. 12-12.

12-4 COINCIDENT-CURRENT MEMORY

The core shift register discussed in the previous section suggests the possibility of using an array of magnetic cores for storing words of binary information. For example, a 10-bit core shift register could be used to store a 10-bit word. The operation would be serial in form, much like the 10-bit flip-flop shift register discussed earlier. It would, however, be subject to the same speed limitations observed in the serial flip-flop register. That is, since each bit must travel down the register from core to core, it requires n clock periods to shift an n-bit word into or out of the register. This shift time may become excessively long in some cases, and a faster method must then be developed. Much faster operation can be achieved if the information is written into and read out of the cores in a parallel manner. Since all the bits are processed simultaneously an entire word can be transferred in only one

Fig. 12-12. Four-stage ring counter for Example 12-3.



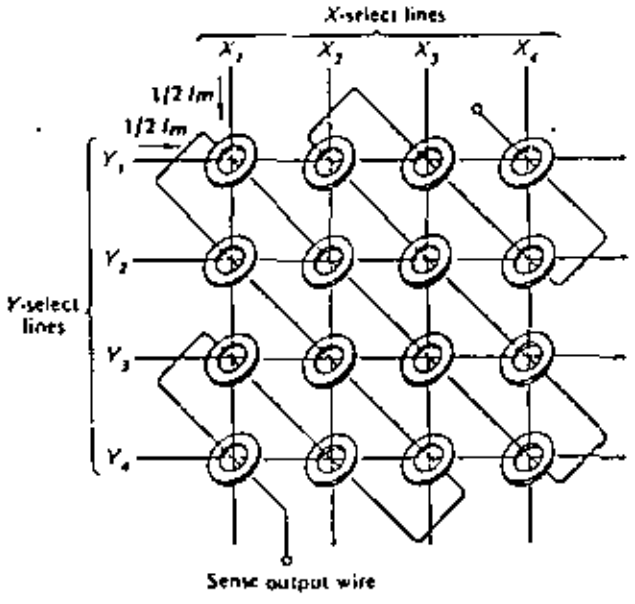


Fig. 12-13. Magnetic-core coincident-current memory.

clock period. A straight parallel system would, however, require one input wire and one output wire for each core. For a large number of cores the total number of wires makes this arrangement impractical, and some other form of core selection must be developed.

The most popular method for storing binary information in parallel form using magnetic cores is the *coincident-current drive* system. Such memory systems are widely used in all types of digital systems from small-scale special-purpose machines up to large-scale digital computers. The basic idea involves arranging cores in a matrix and using two *half-select currents;* the method is shown in Fig. 12-13.

The matrix consists of two sets of drive wires: the X drive wires (vertical) and the Y drive wires (horizontal). Notice that each core in the matrix is threaded by one X wire and one Y wire. Suppose one half-select current $\frac{1}{2}I_m$ is applied to line X_1 and one half-select current $\frac{1}{2}I_m$ is applied to line Y_1. Then the core which is threaded by both lines X_1 and Y_1 will have a total of $\frac{1}{2}I_m + \frac{1}{2}I_m = I_m$ passing through it, and it will switch states. The remaining cores which are threaded by X_1 or Y_1 will each receive only $\frac{1}{2}I_m$, and they will therefore not switch states. Thus we have succeeded in switching one of the 16 cores by selecting two of the input lines (one of the X lines and one of the Y lines). We designate the core that switched in this case as core X_1Y_1, since it was switched by selecting lines X_1 and Y_1. The designation X_1Y_1 is called the *address* of the core since it specifies its location. We can then switch any core X_aY_b located at address X_aY_b by applying $\frac{1}{2}I_m$ to lines X_a and Y_b. For example, the core located in the lower right-hand corner of the matrix is at the address X_4Y_4 and can be switched by applying $\frac{1}{2}I_m$ to lines X_4 and Y

In order that the selected core will switch, the directions of the half-select currents through the X line and the Y line must be additive in the core. In Fig. 12-13, the X select currents must flow through the X lines from the top toward the bottom, while the Y select currents flow through the Y lines from left to right. Application of the *right-hand rule* will demonstrate that currents in this direction switch the core such that the core flux is in a clockwise direction (looking from the top). We define this as switching the core to the 1 state. It is obvious, then, that reversing the directions of both the X and Y line currents will switch the core to the 0 state. Notice that if the X and Y line currents are in a subtractive direction the selected core receives $\frac{1}{2}I_m - \frac{1}{2}I_m = 0$ and the core does not change state.

With this system we now have the ability to switch any one of 16 cores by selecting any two of eight wires. This is a saving of 50 percent over a direct parallel selection system. This saving in input wires becomes even more impressive if we enlarge the existing matrix to 100 cores (a square matrix with 10 cores on each side). In this case, we are able to switch any one of 100 cores by selecting any two of only 20 wires. This represents a reduction of 5 to 1 over a straight parallel selection system.

At this point we need to develop a method of sensing the contents of a core. This can be very easily accomplished by threading one *sense wire* through every core in the matrix. Since only one core is selected (switched) at a time, any output on the sense wire will be due to the changing of state of the selected core, and we will know which core it is since the core address is prerequisite to selection. Notice that the sense wire passes through half the cores in one direction and through the other half in the opposite direction. Thus the output signal may be either a positive or a negative pulse. For this reason, the output from the sense wire is usually amplified and rectified to produce an output pulse which always appears with the same polarity.

Example 12-4

From the standpoint of construction, the core matrix in Fig. 12-14 is more convenient. Explain the necessary directions of half-select currents in the X and Y lines for proper operation of the matrix.

Solution

Core X_1Y_1 is exactly similar to the previously discussed matrix in Fig. 12-13. Thus a current passing down through X_1 and to the right through Y_1 will set core X_1Y_1 to the 1 state. To set core X_1Y_2 to the 1 state, current must pass down through line X_1, but current must pass from the right to the left through line Y_2 (check with the *right-hand rule*). Proceeding in this fashion, we see that core X_1Y_3 is similar to X_1Y_1. Therefore, current must pass through line Y_3 from left to right. Similarly, core X_1Y_4 is similar to core X_1Y_2 and current must therefore pass through line Y_4 from right to left. In general, current must pass from *left to right* through the *odd*-numbered Y lines, and from *right to left* through even-numbered Y lines.

Now, since current must pass from left to right through line Y_3, it is easily seen that current must pass upward through line X_2 in order to set core X_2Y_3. By an argument similar to that given for the Y lines, current must pass *downward* through the *odd*-numbered X lines and *upward* through the *even*-numbered X lines.

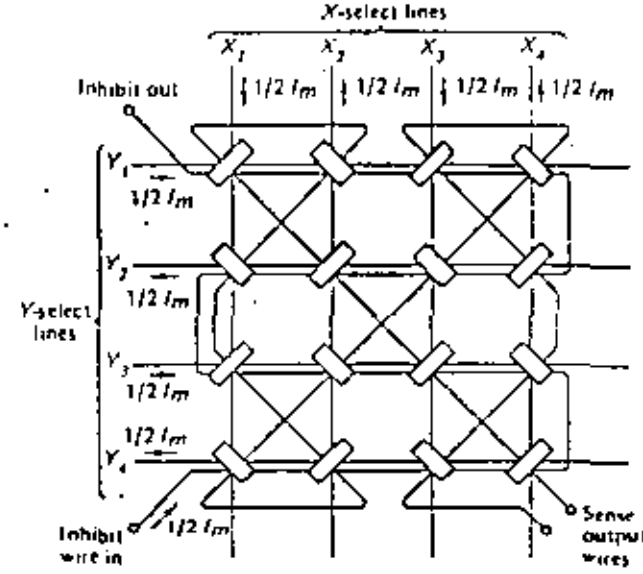


Fig. 12-14. Coincident-current memory matrix (one plane).

The matrix shown in Fig. 12-14 has one extra winding which we have not yet discussed. This is the *inhibit wire*. In order to understand its operation and function, let us examine the methods for writing information into the matrix and reading information from the matrix.

To write a 1 in any core (that is, to set the core to the 1 state), it is only necessary to apply $\frac{1}{2}I_m$ to the X and Y lines selecting that core address. If we desired to write a 0 in any core (that is, set the core to the 0 state), we could simply apply a current of $-\frac{1}{2}I_m$ to the X and Y lines selecting that core address. We can also write a 0 in any core by making use of the *inhibit* wire shown in Fig. 12-14. (We assume that all cores are initially in the 0 state.) Notice that the application of $\frac{1}{2}I_m$ to this wire in the direction shown on the figure results in a complete cancellation of the Y line select current (it also tends to cancel an X line current). Thus to write a 0 in any core, it is only necessary to select the core in the same manner as if writing a 1, and at the same time apply an *inhibit* current to the *inhibit* wire. The major reason for writing a 0 in this fashion will become clear when we use these matrix planes to form a complete memory.

To summarize, we write a 1 in any core X_aY_b by applying $\frac{1}{2}I_m$ to the select lines X_a and Y_b. A 0 can be written in the same fashion by simply applying $\frac{1}{2}I_m$ to the *inhibit* line at the same time (if all cores are initially reset).

To read the information stored in any core, we simply apply $-\frac{1}{2}I_m$ to the proper X and Y lines and detect the output on the sense wire. The select currents of $-\frac{1}{2}I_m$ reset the core, and if the core previously held a 1, an output pulse occurs. If the core previously held a 0, it does not switch, and no output pulse appears.

This, then, is the complete coincident-current selection system for one plane. Notice that reading the information out of the memory results in a complete loss of

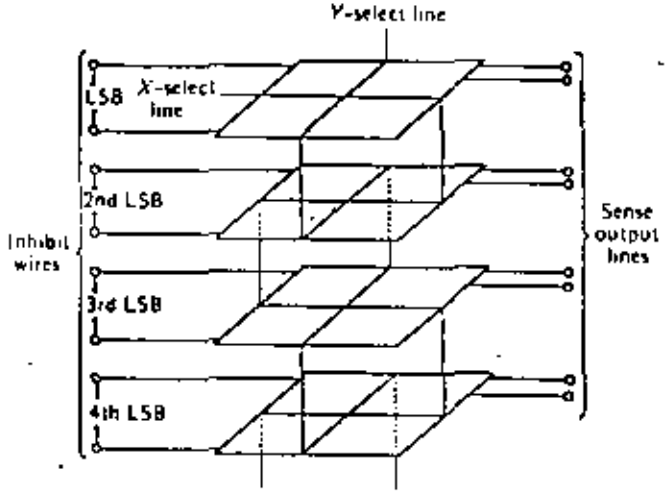


Fig. 12-15. Complete coincident-current memory system.

information from the memory, since all cores are reset during the *read* operation. This is referred to as a *destructive readout* or DRO system. This matrix plane is used to store one bit in a word, and it is necessary to use n of these planes to store an n-bit word.

A complete parallel coincident-current memory system can be constructed by stacking the basic memory planes in the manner shown in Fig. 12-15. All the X drive lines are connected in series from plane to plane as are all the Y drive lines. Thus the application of $V_s I_m$ to lines X_a and Y_b results in a selection of core $X_a Y_b$ in every plane. In this fashion we can simultaneously switch n cores, where n is the number of planes. These n cores represent one word of n bits. For example, the top plane might be the LSB, the next to the top plane would then be the second LSB, and so on; the bottom plane would then hold the MSB.

To read information from the memory, we simply apply $-V_s I_m$ to the proper address and sense the outputs on the n sense lines. Remember that readout results in resetting all cores to the 0 state, and thus that word position in the memory is cleared to all 0s.

To write information into the memory, we simply apply $V_s I_m$ to the proper X and Y select lines. This will, however, write a 1 in every core. So for the cores in which we desire a 0, we simultaneously apply $V_s I_m$ to the *inhibit* line. For example, to write 1001 in the upper four planes in Fig. 12-15, we apply $V_s I_m$ to the proper X and Y lines and at the same time apply $V_s I_m$ to the *inhibit* lines of the second and third planes.

This method of writing assumes that all cores were previously in the 0 state. For this reason it is common to define a *memory cycle*. One memory cycle is defined as a *read* operation followed by a *write* operation. This serves two purposes: first it ensures that all the cores are in the 0 state during the *write* operation; second, it provides the basis for designing a *nondestructive readout* (NDRO) system.

It is quite inconvenient to lose the data stored in the memory every time they are read out. For this reason, the NDRO has been developed. One method for accomplishing this function is to read the information out of the memory into a temporary storage register (flip-flops perhaps). The outputs of the flip-flops are then used to drive the *inhibit* lines during the *write* operation which follows (inhibit to write a 0 and do not inhibit to write a 1). Thus the basic memory cycle allows us to form an NDRO memory from a DRO memory.

Example 12-5

Describe how a coincident-current memory might be constructed if it must be capable of storing 1,024 twenty-bit words.

Solution

Since there are 20 bits in each word, there must be 20 planes in the memory (there is one plane for each bit). In order to store 1,024 words, we could make the planes square. In this case, each plane would contain 1,024 cores; it would be constructed with 32 rows and 32 columns since $(1024)^{1/2} = (2^{10})^{1/2} = 2^5 = 32$. This memory is then capable of storing $1,024 \times 20 = 20,480$ bits of information. Typically, a memory of this size might be constructed in a 3-in cube. Notice that in this memory we have the ability to switch any one of 20,480 cores by controlling the current levels on only 84 wires (32 X lines, 32 Y lines, and 20 *inhibit* lines). This is indeed a modest number of control lines.

Example 12-6

Devise a means for making the memory system in the previous example a NDRO system.

Solution

One method for accomplishing this is shown in Fig. 12-16. The basic core array consists of twenty 32-by-32 core planes. For convenience, only the three LSB planes and the MSB core plane are shown in the diagram. The wiring and operation for the other planes are the same. For clarity, the X and Y select lines have also been omitted. The output sense line of each plane is fed into a bipolar amplifier which rectifies and amplifies the output so that a positive pulse appears any time a set core is reset to the 0 state. A complete memory cycle consists of a *clear* pulse followed by a *read* pulse followed by a *write* pulse. The proper waveforms are shown in Fig. 12-17. The *clear* pulse first sets all flip-flops to the 0 state (this *clear* pulse can be generated from the trailing edge of the *write* pulse). When the *read* line goes high, all the AND gates driven by the bipolar amplifiers are enabled. Shortly after the rise of the *read* pulse, $-V_s I_m$ is applied to the X and Y lines designating the address of the word to be read out. This resets all cores in the selected word to the 0 state, and any core which contained a 1 will switch. Any core which switches generates a pulse on the sense line which is amplified and appears as a positive pulse at the output of one of the bipolar amplifiers. Since the *read* AND gates are enabled, a positive pulse at the output of any amplifier passes through the AND gate and sets the flip-flop. Shortly thereafter the half-select circuits disappear.

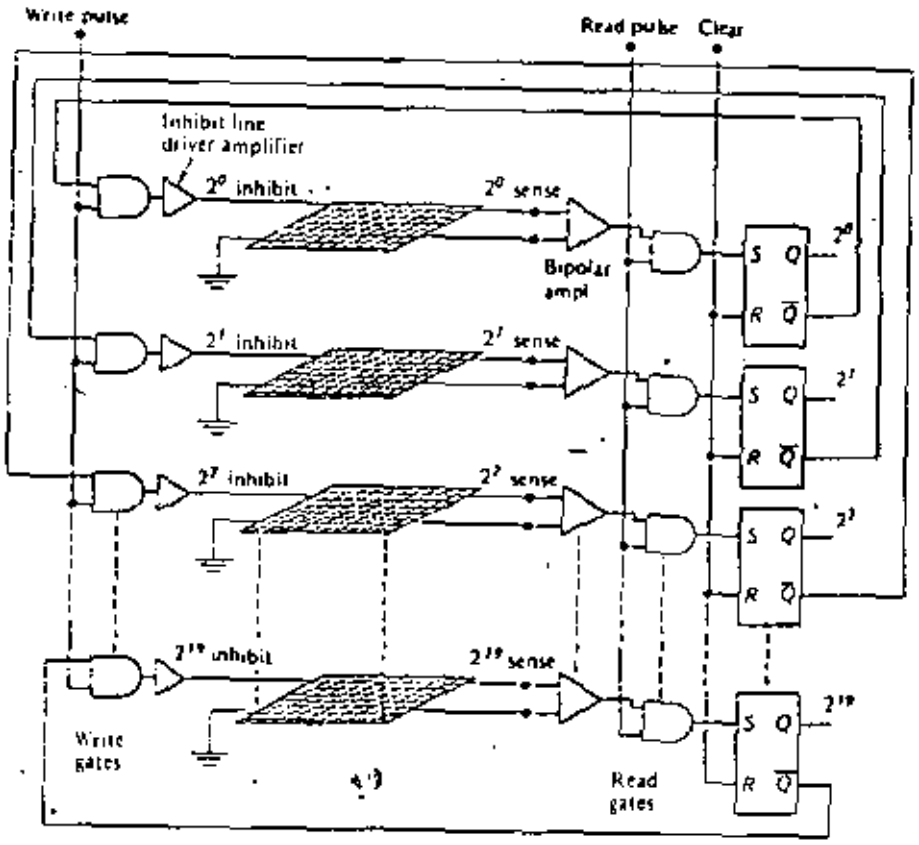


Fig. 12-16. NDRO system for Example 12-6.

the *read* line goes low, and the flip-flops now contain the data which were previously in the selected cores. Shortly after the *read* line goes low, the *write* line goes high, and this enables the *write* AND gates (connected to the *inhibit* line drivers). The 0 side of any flip-flop which has a 0 stored in it is high, and this enables the *write* AND gate to which it is connected. In this manner an *inhibit* current is applied to any core which previously held a 0. Shortly after the rise of the *write* pulse, positive half-select currents are applied to the same X and Y lines. These select currents set a 1 in any core which does not have an *inhibit* current. Thus the information stored in the flip-flops is written directly back into the cores from which it came. The half-select currents are then reduced to zero, and the *write* line goes low. The fall of the *write* line is used to reset the flip-flops, and the system is now ready for another *read/write* cycle.

The NDRO memory system discussed in the preceding example provides the means for reading information from the system without losing the individual bits stored in the cores. To have a complete memory system, we must have the

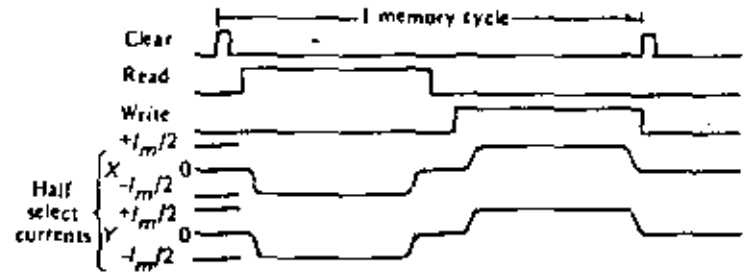


Fig. 12-17. NDRO waveforms for Fig. 12-16 (read from memory).

capability to write information into the cores from some external source (e.g., input data). The *write* operation can be realized by making use of the exact same NDRO waveforms shown in Fig. 12-17. We must, however, add some additional gates to the system such that during the *read* pulse the data set into the flip-flops will be the external data we wish stored in the cores. This could easily be accomplished by adding a second set of AND gates which can be used to set the flip-flops. The logic diagram for the complete memory system is shown in Fig. 12-18. For simplicity, only the LSB is shown since the logic for every bit is identical.

For the complete memory system we recognize that there are two distinct operations. They are *write into memory* (i.e., store external data in the cores) and *read from memory* (i.e., extract data from the cores to be used elsewhere). For these two operations we must necessarily generate two distinct sets of control waveforms. The waveforms for *read from memory* are exactly those shown in Fig. 12-17, and the events are summarized as follows:

1. The *clear* pulse resets all flip-flops.
2. During the *read* pulse, all cores at the selected address are reset to 0, and the data stored in them are transferred to the flip-flops by means of the read AND gates.
3. During the *write* pulse, the data held in the flip-flops are stored back in the cores by applying positive half-select currents (the *inhibit* currents are controlled by the 0 sides of the flip-flops and provide the means of storing 0s in the cores).

The *write into memory* waveforms are exactly the same as shown in Fig. 12-17 with one exception: that is, the *read* pulse is replaced with the *enter data* pulse. The events for *write into memory* are shown in Fig. 12-19, and are summarized as follows:

1. The *clear* pulse resets all flip-flops.
2. During the *enter data* pulse, the negative half-select currents reset all cores at the selected address. The core outputs are not used, however, since the *read* AND gates are not enabled. Instead, external data are set into the flip-flops through the enter AND gates.

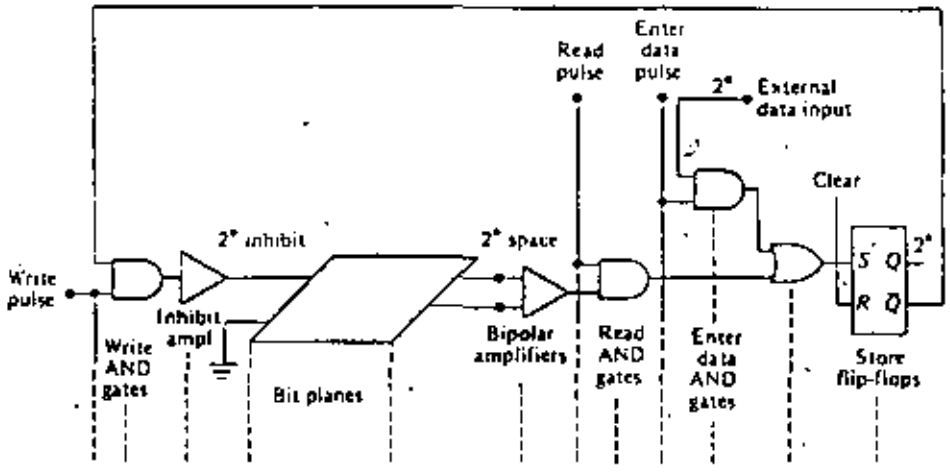


Fig. 12-18. Complete NDRO memory system (LSB plane only).

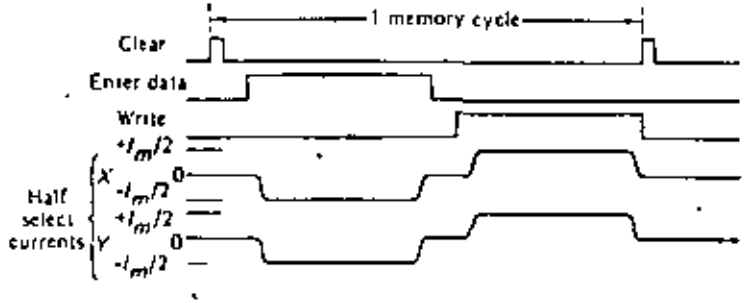
3. During the *write* pulse, data held in the flip-flops are stored in the cores exactly as before.

In conclusion, we see that *write into memory* and *read from memory* are exactly the same operations with the exception of the data stored in the flip-flops. The waveforms are exactly the same when the *read* and *enter data* pulses are used appropriately, and the same total cycle time is required for either operation.

It should be pointed out that a number of difficulties are encountered with this type of system. First of all, since the *sense* wire in each plane threads every core in that plane, a number of undesired signals will be on the *sense* wire. These undesired signals are a result of the fact that many of the cores in the plane receive a half-select current and thus exhibit a slight flux change.

The geometrical pattern of core arrangement and wiring shown in Fig. 12-13 represents an attempt to minimize the *sense*-line noise by cancellation. For example, the signals induced in the *sense* line by the *X* and *Y* drive currents would hopefully

Fig. 12-19. NDRO waveforms for Fig. 12-18 (write into memory).



be canceled out since the *sense* line crosses these lines in the opposite direction the same number of times. Furthermore, the sense line is always at a 45° angle to the *X* and *Y* select lines. Similarly, the noise signals induced in the *sense* line by the partial switching of cores receiving half-select currents should cancel one another. This, however, assumes that all cores are identical, which is hardly ever true!

Another method for eliminating noise due to cores receiving half-select currents would be to have a core which exhibits an absolutely rectangular BH-curve as shown in Fig. 12-20a. In this case, a half-select current would move the operating point of the core perhaps from point *a* to point *b* on the curve. However, since the top of the curve is horizontal, no flux change would occur, and therefore no undesired signal could be induced in the sense wire. This is an ideal curve, however, and cannot be realized in actual practice. A measure of core quality is given by the *squareness ratio*, which is defined as

$$\text{Squareness ratio} = \frac{B_r}{B_m}$$

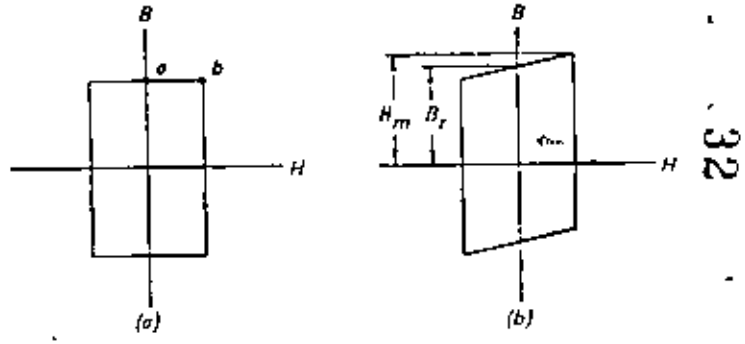
This is the ratio of the flux density at the remanent point B_r to the flux density at the switching point B_m and is shown graphically in Fig. 12-20*b*. The ideal value is, of course, 1.0, but values between 0.9 and 1.0 are the best obtainable.

12-5 MEMORY ADDRESSING

In this section we investigate the means for activating the *X* and *Y* selection lines which supply the half-select currents for switching the cores in the memory. First of all, since it typically requires 100 to 500 mA in each select line (that is, I_m is typically between 100 and 500 mA), each select line must be driven by a current amplifier. A special class of transistors has been developed for this purpose; they are referred to as *core drivers* in data sheets. What is then needed is the means for activating the proper core-driver amplifier.

Up to this point, we have designated the *X* lines as X_1, X_2, X_3, . . . , X_n, and the *Y*

Fig. 12-20. Hysteresis curves. (a) Ideal. (b) Practical (realizable).



lines as Y_1, Y_2, Y_3, . . . , Y_n. For a square matrix, n is the number of cores in each row or column, and there are then n^2 cores in a plane. When the planes are arranged in a stack of M planes, where M is the number of bits in a word, we have a memory capable of storing n^2, M-bit words. Any two select lines can then be used to read or write a word in memory, and the address of that word is X_aY_b, where a and b can be any number from 1 to n. For example, X_2Y_3 represents the column of cores at the intersection of the X_2 and Y_3 select lines, and we can then say that the address of this word is 23. Notice that the first digit in the address is the X line and the second digit is the Y line. This is arbitrary and could be reversed.

This method of address designation entails but one problem: in a digital system we can use only the numbers 1 and 0. The problem is easily resolved, however, since the address 23, for example, can be represented by 010 011 in binary form. If we use three bits for the X line position and three bits for the Y line position, we can then designate the address of any word in a memory having a capacity of 64 words or less. This is easy to see, since with three bits we can represent eight decimal numbers, which means we can define an $8 \times 8 = 64$ word memory. If we chose an eight-bit address, four bits for the X line and four bits for the Y line, we could define a memory having $2^4 \times 2^4 = 16 \times 16 = 256$ words. In general, an address of B bits can be used to define a square memory of 2^B words, where there are $B/2$ bits for the X lines and $B/2$ bits for the Y lines. From this discussion it is easy to see why large-scale coincident-current memory systems usually have a capacity which is an even power of 2.

Example 12-7

What would be the structure of the binary address for a memory system having a capacity of 1,024 words?

Solution

Since $2^{10} = 1{,}024$, there would have to be 10 bits in the address word. The first five bits could be used to designate one of the required 32 X lines, and the second five bits could be used to designate one of the 32 Y lines.

Example 12-8

For the memory system described in the previous example, what is the decimal address for the following binary addresses?

(a) 10110 00101
(b) 11001 01010
(c) 11110 00001

Solution

(a) The first five bits are the X line and correspond to the decimal number 22. The second five bits represent the Y line and correspond to the decimal number 5. Thus the address is $X_{22}Y_5$.
(b) $11001_2 = 25_{10}$ and $01010_2 = 10_{10}$. Therefore, the address is $X_{25}Y_{10}$.
(c) The address is $X_{30}Y_1$.

The B bits of the address in a typical digital system are stored in a series of flip-flops called the "address register." The address in binary form must then be decoded into decimal form in order to drive one of the X line drivers and one of the Y line driver amplifiers as shown in Fig. 12-21. The X and Y decoding matrices shown in the figure can be identical, and are essentially binary-to-decimal decoders. Binary-to-decimal decoding and appropriate matrices were discussed in Chap. 10.

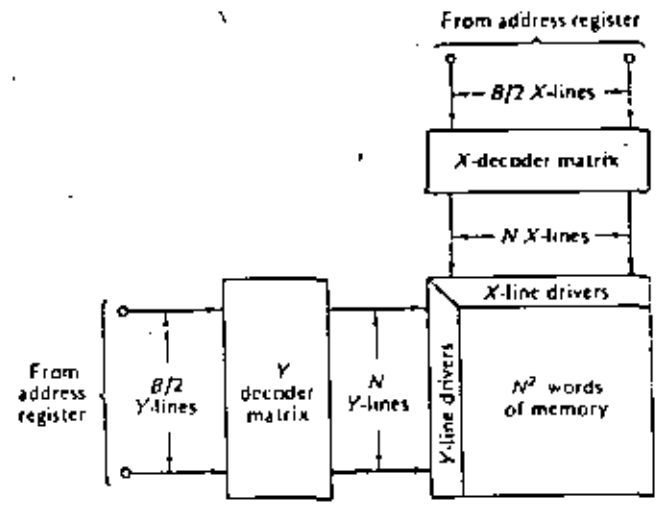


Fig. 12-21. Coincident-current memory addressing.

12-6 SEMICONDUCTOR MEMORIES—BIPOLAR

Reduced cost and size, improved reliability and speed of operation, and increased packing density are among the technological advances which have made semiconductor memories a reality in modern digital systems. A *bipolar* memory is constructed using the familiar bipolar transistor, while the MOS memory makes use of the MOSFET. In this section we consider the characteristics of bipolar semiconductor memories; MOS memories are considered in the next section.

A "memory cell" is a unit capable of storing binary information; the basic memory unit in a bipolar semiconductor memory is the flip-flop (latch) shown in Fig. 12-22. The cell is *selected* by raising the X *select* line and the Y *select* line; the sense lines are both returned through low-resistance sense amplifiers to ground. If the cell contains a 1, current is present in the 1 sense line. On the other hand, if the cell contains a 0, current is present in the 0 sense line.

To write information into the cell, the X and Y select lines are held high; holding the 0 sense line high ($+V_{cc}$) while the 1 sense line is grounded writes a 1 into the cell. Alternatively, holding the 1 sense line high ($+V_{cc}$) and the 0 sense line at ground during a select writes a 0 into the cell. The basic bipolar memory cell in Fig. 12-22 can be used to store one binary digit (bit), and thus many such cells are required to form a memory.

Sixteen of the RS flip-flop cells in Fig. 12-22 have been arranged in a 4-by-4 ma-

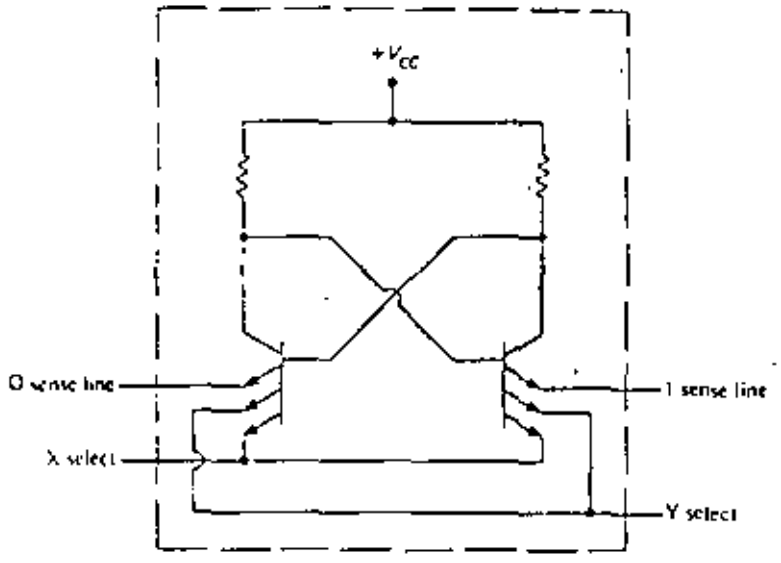


Fig. 12-22. Bipolar memory cell circuit.

tric to form a 16-word by one-bit memory in Fig. 12-23. It is referred to as a random access memory (RAM) since each bit is individually addressable by selecting one X line and one Y line. It is also a nondestructable readout since the *read* operation does not alter the state of the selected flip-flop. This memory comes on a single semiconductor chip (in a single package) as shown in Fig. 12-24a. To construct a 16-word memory with more than one bit per word requires stacking these basic units. For example, six of these chips can be used to construct a 16-word by six-bit memory as shown in Fig. 12-24b. The X and Y address lines are all connected in parallel. The units shown in Figs. 12-23 and 12-24 are essentially equivalent to the Texas Instruments 9033 and Fairchild 93407 (5033 or 9033).

Example 12-9

Using a 9033, explain how to construct a 16-word by 12-bit memory. What address would select the 12-bit word formed by the bits in column 1 and row 1 of each plane?

Solution

Connect twelve 16-word by one-bit memory planes in parallel. The address $X_0X_1X_2X_3Y_0Y_1Y_2Y_3 = 10001000$ selects the bit in the first column and the first row of each plane (a 12-bit word represented by the vertical column of 12 bits).

For larger memories, the appropriate address decoding, driver amplifiers, and *read/write* logic are all constructed in a single package. Such a unit, for example, is the Fairchild 93415—this is a 1,024-word by one-bit read/write RAM. The logic diagram is shown in Fig. 12-25. An address of 10 bits is required ($A_0A_1A_2A_3A_4$ $A_5A_6A_7$) to obtain 1,024 words. That is, n bits provide 2^n word

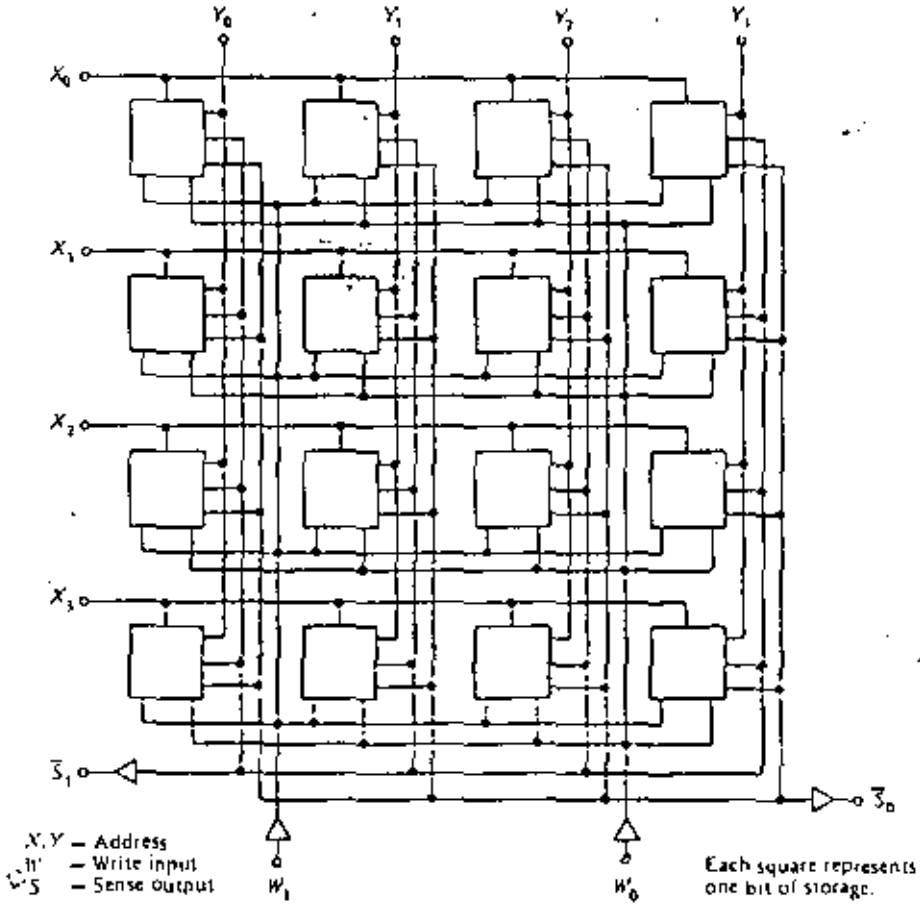


Fig. 12-23. 16-word 1-bit memory.

locations. In this case, the 10-bit address is divided into two groups of five bits each. The first five (A_0, A_1, A_2, A_3, A_4) select a unique group of 32 lines from the 32-by-32 array. The second five (A_5, A_6, A_7, A_8, A_9) select exactly one of the 32 preselected lines for reading or writing. These basic units are then stacked in parallel as shown previously; n units provide a memory having 1,024 words by n bits.

Another interesting and useful type of semiconductor memory is shown in Fig. 12-26. This is a bipolar TTL read-only memory (ROM). The information stored in a ROM can be read out, but new information cannot be written into it. Thus, the information stored is permanent in nature. ROMs can be used to store mathematical tables, code translations, and other fixed data. The logic required for a ROM is generally simpler than that required for a *read/write* memory, and the unit shown in Fig. 12-26 (equivalent to a TI 9034 or Fairchild 93434) provides an eight-bit output word for each five-bit input address. There are, of course, 32 words, since an address of five bits provides 32 words ($2^5 = 32$).

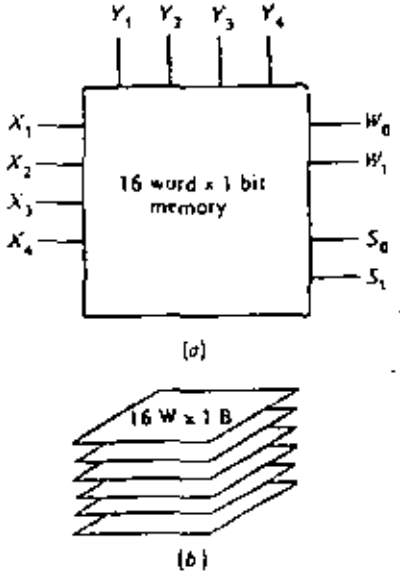
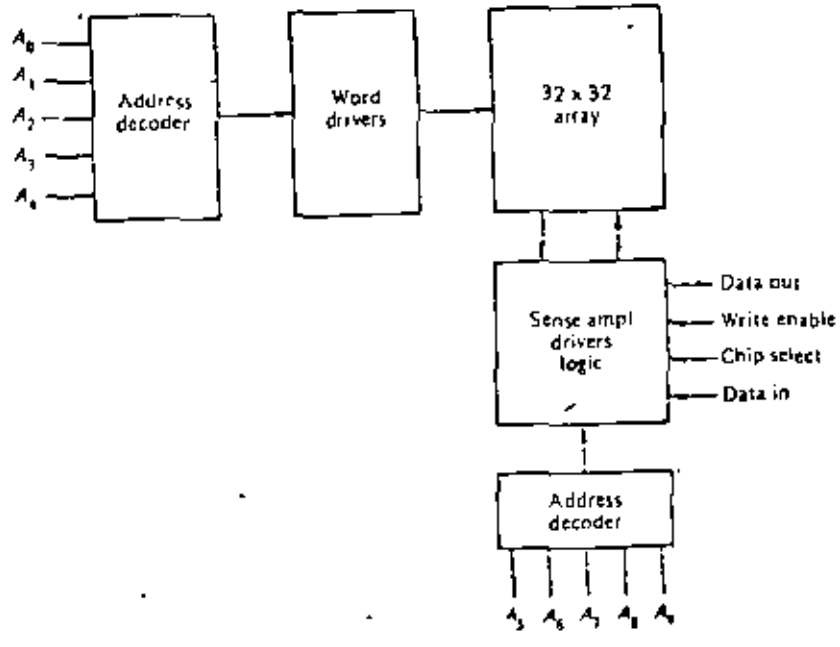


Fig. 12-24. (a) Logic diagram. (b) Six chips stacked to get a 16-word × 6-bit memory.

Fig. 12-25. 1024-word × 1-bit RAM.



Example 12-10

How many address bits are required for a 123-word by four-bit ROM constructed similarly to the unit in Fig. 12-26? How many memory cells are there in such a unit?

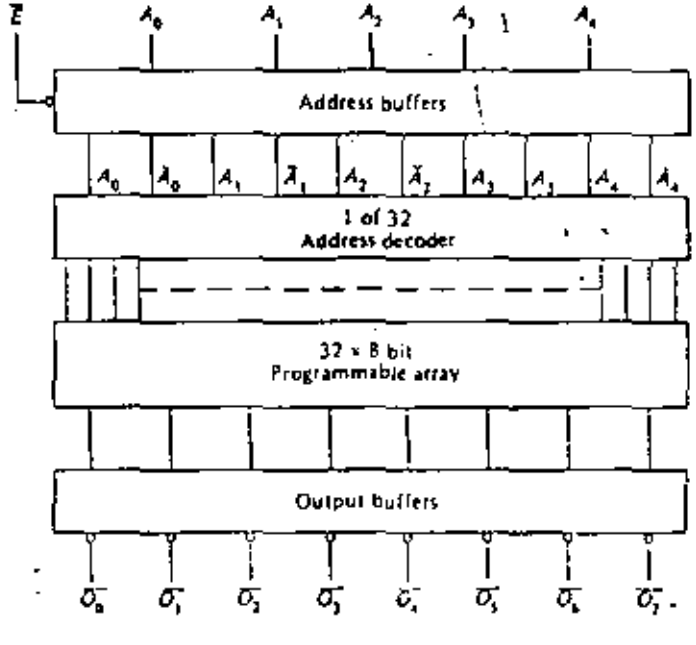
Solution

It requires seven address bits, since $2^7 = 128$. There would be $128 \times 4 = 512$ memory cells.

12-7 SEMICONDUCTOR MEMORIES—MOS

The basic device used in the construction of an MOS semiconductor memory is the MOSFET. Both *p*-channel and *n*-channel devices are available. The *n*-channel memories have simpler power requirements, usually only $+V_{cc}$, and are quite compatible with TTL since they are usually referenced to ground and have positive signal levels up to $+V_{cc}$. The *p*-channel devices generally require two power-supply voltages and may require signal inversion in order to be compatible with TTL. MOS devices are somewhat simpler than bipolar devices; as a result, MOS memories can be constructed with more bits on a chip, and they are generally less expensive than bipolar memories. The intrinsic capacitance associated with an MOS device generally means that MOS memories are slower than bipolar units, but this capacitance can be used to good advantage, as we shall see.

Fig. 12-26. 256-bit (32-word × 8-bit) ROM.



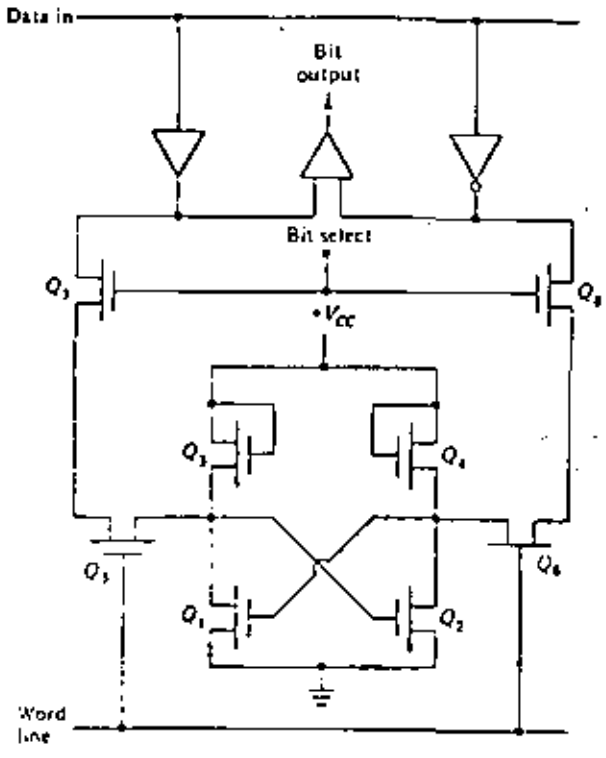


Fig. 12-27.

An *RS flip-flop* constructed using MOSFETs is shown in Fig. 12-27. It is a standard bistable circuit, with Q_1 and Q_2 as the two active devices, and Q_3 and Q_4 acting as active pull-ups (essentially resistances). Q_5 and Q_6 couple the flip-flop outputs to the two *bit lines*. This cell is constructed using *n*-channel devices, and selection is accomplished by holding both the *word* line and the *bit select* line high ($+V_{CC}$). The positive voltage on the *word* line turns on Q_5 and Q_6, and the positive voltage in the *bit select* line turns on Q_7 and Q_8. Under this condition, the flip-flop outputs are coupled directly to the *bit output* amplifier (one input side is high, and the other must be low). On the other hand, data can be stored in the cell when it is selected by applying 1 or 0 ($+V_{cc}$ or 0 V dc) at the *data input* terminal. The basic memory cell in Fig. 12-27 is used to construct a 1,024-bit RAM having a logic diagram similar to Fig. 12-25. This particular unit is a 2602 as manufactured by Signetics Corp.

A memory cell using *p*-channel MOSFETs is shown in Fig. 12-28. Q_1 and Q_2 are the two active devices forming the flip-flop, while Q_3 and Q_4 act as active load resistors. The cell is selected by a low logic level at the *bit select* input. This couples the contents of the flip-flop out to appropriate amplifiers (as in Fig. 12-27) through Q_5 and Q_6.

A *static memory* is composed of cells capable of storing binary information indefinitely. For example, the bipolar or MOSFET flip-flop remains set or reset as long

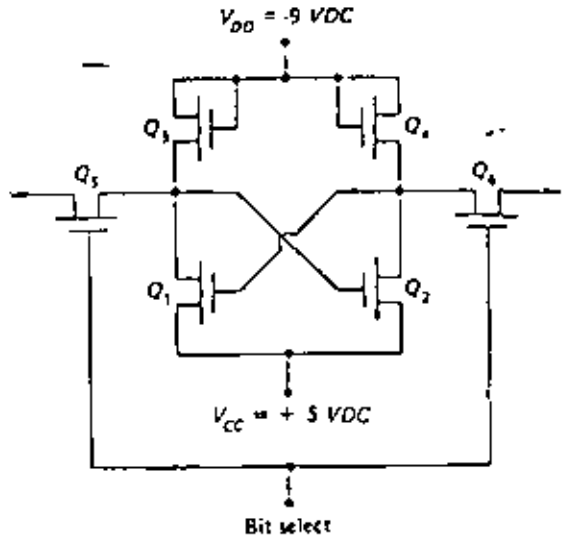
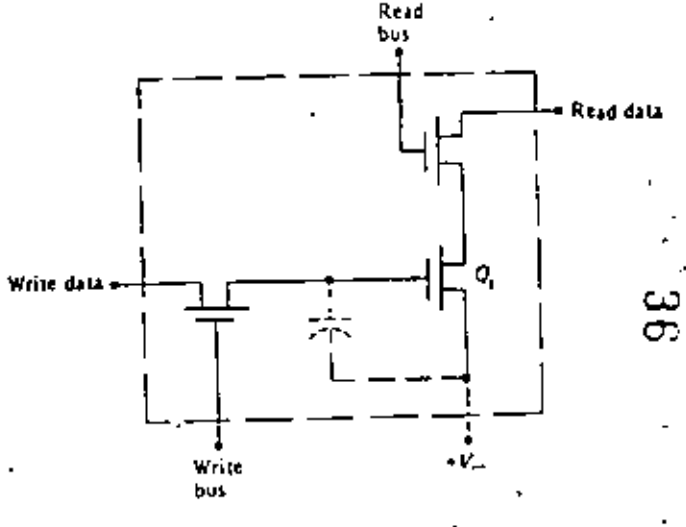


Fig. 12-28.

as power is applied to the circuit. Also, a magnetic core remains set or reset, even if power is removed. These basic memory cells are used to form a *static memory*. On the other hand, a *dynamic memory* is composed of memory cells whose contents tend to decay over a period of time (perhaps milliseconds or seconds); thus, their contents must be restored (refreshed) periodically. The leaky capacitance associated with a MOSFET can be used to store charge, and this is then the basic unit used to form a dynamic memory. (There are no dynamic bipolar memories because there is no suitable intrinsic capacitance for charge storage.) The need for extra

Fig. 12-29. Basic dynamic memory cell.



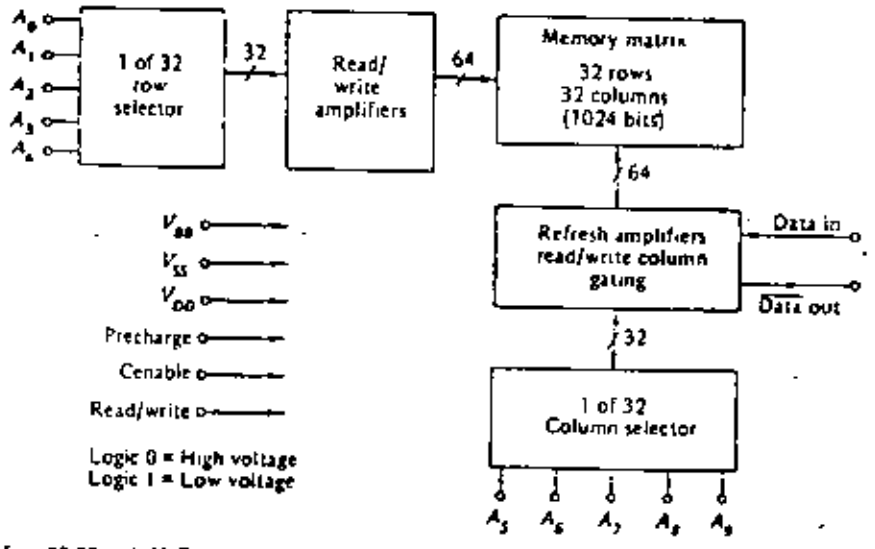


Fig. 12-30. 1103 Dynamic RAM logic diagram.

timing signals and logic to periodically refresh the dynamic memory is a disadvantage, but the higher speeds and lower power dissipation, and therefore the increased cell density, outweighs the disadvantages. Note that a dynamic memory dissipates energy only when reading, writing, or refreshing cells. A typical dynamic memory cell is shown in Fig. 12-29.

The dynamic memory cell in Fig. 12-29 is constructed from p-channel MOSFETs. The gate capacitance (shown as a dotted capacitor) is used as the basic storage element. To write into the cell requires holding the *write bus* at a low logic level; then a low level at the *write data* input charges the gate capacitance (stores a 1 in the cell). With the write bus held low, and a high logic level ($+V_{cc}$) at the *write data* input, the gate capacitance is discharged (a 0 is stored in the cell).

To read from the cell requires holding the *read bus* input at a low logic level. If the gate capacitance is charged (cell contains a 1), the *read data* line goes to $+V_{cc}$; if the cell contains a 0, the *read data* line remains low.

The memory cell in Fig. 12-29 is used by a number of manufacturers to construct the widely used 1103 1,024-bit dynamic RAM. The logic diagram is shown in Fig. 12-30. Refer to manufacturers' data sheets for more detailed operating information.

12-8 MAGNETIC-DRUM STORAGE

Magnetic cores and semiconductor devices arranged in three-dimensional form offer great advantages as memory systems. By far the most important advantage is the speed with which data can be written into or read from the memory system. This is called the *access time*, and for core memory systems it is simply the time of one *read/write* cycle. Thus the access time is directly related to the clock, and typical values are from less than 1 to a few microseconds. These types of memory

systems are said to be *random-access* since any word in the memory can be selected at random. The primary disadvantage of this type of memory system is the cost of construction for the amount of storage available. As an example, recall that a magnetic tape is capable of storing large quantities of data at a relatively low cost per bit of storage. A typical tape might be capable of storing up to 20 million characters, which corresponds to 120 million bits (Chap. 10). To construct such a memory with magnetic cores requires about 3 million cores per plane, assuming we use a stack of 36 planes corresponding to a 36-bit word. It is quite easy to understand the impracticality of constructing such a system. What is needed, then, is a system capable of storing information with less cost per bit but having a greater capacity.

Such a system is the *magnetic-drum* storage system. The basis of a magnetic drum is a cylindrical-shaped drum, the surface of which has been coated with a magnetic material. The drum is rotated on its axis as shown in Fig. 12-31, and the *read/write* heads are used to record information on the drum or read information from the drum. Since the surface of the drum is magnetic, it exhibits a rectangular-hysteresis-loop property and can thus be magnetized. The process of recording on the drum is much the same as for recording on magnetic tape, as discussed in Chap. 10, and the same methods for recording are commonly used (i.e., RZ, NRZ, and NRZI). The data are recorded in tracks around the circumference of the drum, and there is one *read/write* head for each track. There are three major methods for storing information on the drum surface; they are *bit-serial*, *bit-parallel*, and *bit-serial-parallel*.

In bit-serial recording, all the bits in one word are stored sequentially, side by side, in one track of the drum. Bit-serial storage is shown in Fig. 12-32*a*. Storage densities of 200 to 1,000 bits per in are typical for magnetic drums. A typical drum might be 8 in in diameter and thus have the capacity to store $\pi \times 8$ in $\times$ 200 bits per in = 5,024 bits in each track. Drums have been constructed with anywhere from 15 to 400 tracks, and a spacing of 20 tracks to the inch is typical. If we assume this particular drum is 8 in wide and has a total of 100 tracks, we see immediately that it has a storage capacity of 5,024 bits per track $\times$ 100 tracks =

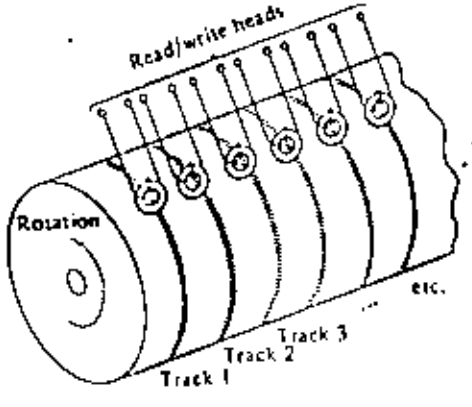


Fig. 12-31. Magnetic-drum storage.

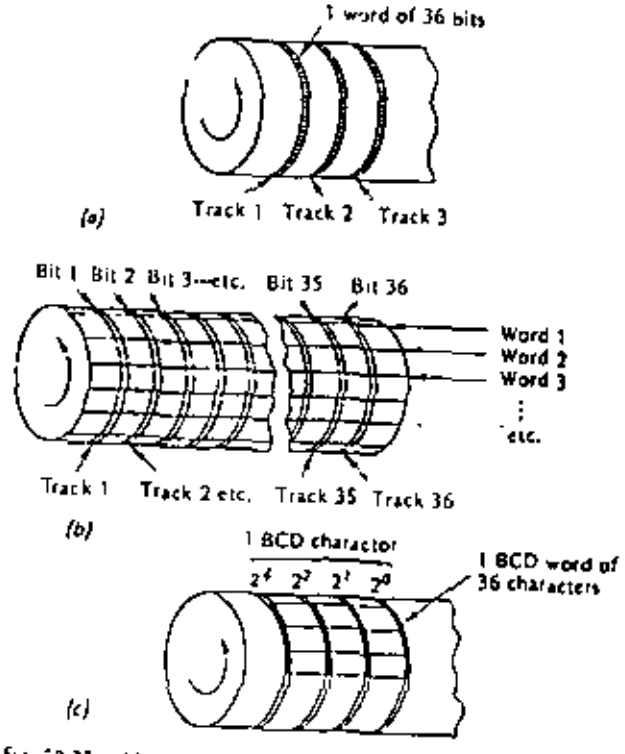


Fig. 12-32. Magnetic-drum organization. (a) Bit-serial storage. (b) Bit-parallel storage. (c) Bit-serial-parallel storage.

502,400 bits of information. Compare this capacity with that of a coincident core memory, which is 64 cores on a side (quite a large core system) with 64 core planes. This core memory has a capacity of $2^6 \times 2^6 \times 2^6 = 262{,}144$ bits. The drum described above is actually considered small, and much larger drums have been constructed and are now in use.

Example 12-11

A certain magnetic drum is 12 in in diameter and 12 in long. What is the storage capacity of the drum if there are 200 tracks and data are recorded at a density of 500 bits per in?

Solution

Each track has a capacity of $\pi \times 12$ in $\times$ 500 bits per in $\approx$ 18,840 bits. Since there are 200 tracks, the drum has a total capacity of $18{,}840 \times 200 = 3{,}768{,}000$ bits.

In the preceding example, each track has the ability to store about 18,840 bits. If we use a 36-bit word, we can store about 523 words in each track. Since the words are stored sequentially around the drum, and since there is only one *read/write*

head for the track, it is easy to see that we may have to wait to read any one word. That is, the drum is rotating, and the word we want to read may not be under the read head at the time we choose to read it. It may in fact have just passed under the head, and we will have to wait until the drum completes nearly a full revolution before it is under the head again. This points out one of the major disadvantages of the drum compared with the core storage. That is the problem of access time. On the average, we can assume that we will have to wait the time required for the drum to complete one-half a revolution. A drum is thus said to have *restricted* access.

Example 12-12

If the drum in Example 12-11 rotates at a speed of 3,000 rpm, what is the average access time for the drum?

Solution

3,000 rpm = 50 rps. Thus the time for one revolution is 1/(50 rps) = 20 ms. Thus, the average access time is one-half the time of one revolution, which is 10 ms. Contrast this with a coincident-current core memory which has a direct access time of a few microseconds.

Notice in the previous example that it requires a short period of time to read the 36 bits of the word, since they appear under the *read* head one bit at a time in a serial fashion. The actual time required is small compared with the access time and is found to be (20 ms/r)/(523 words per track) = 40 μs. This read time can be reduced by storing the data on the drum in a parallel manner, as shown in Fig. 12-32b.

The average access time for bit-parallel storage is the same as for bit-serial storage, but it is possible to read and record information at a much faster rate with the bit-parallel system. Let us use the drum in Example 12-11 once more. Since there are 523 words around each track, and since the drum rotates at 50 rps, we can read (or write) 523 words per revolution $\times$ 50 rps = 26,150 words per second. If the data were stored in parallel fashion, we could read (or write) at 36 times this rate, or at a rate of 18,840 words per revolution $\times$ 50 rps = 942,000 words per second. We would, of course, arrange to have the number of tracks on the drum an even multiple of the number of bits in a word. For example, with a 36-bit word we might use a drum having 36 or 72 or 108 tracks.

A third method for recording data on a drum is called "bit-serial-parallel." The method is shown in Fig. 12-32c and is commonly used for storing BCD information. The access and read (or write) times are a combination of the serial and parallel times. One BCD character occupies one bit in each of four adjacent tracks. Thus, every four tracks might be called a "band," and each BCD character occupies one space in the band. If there are 36 BCD characters in a word, we can store 523 words on the drum of Example 12-11.

Quite often the access time is speeded up by the addition of extra *read/write* heads around the drum. For example, we might use two sets of heads placed on opposite sides of the drum. This would obviously cut the access time in half. Alter-

natively, we might use three sets of heads arranged around the drum at 120° angles. This would reduce the access time by one-third.

Since writing on and reading from the drum must be very carefully timed, one track in the drum is usually reserved as a timing track. On this track, a series of timing pulses is permanently recorded and is used to synchronize the *write* and *read* operations. For the drum discussed in Example 12-11, there are 523 words in each track around the circumference of the drum. We might then record a series of 523 equally spaced timing marks around the circumference of the timing track. Each pulse would then designate the *read* or *write* position for a word on the drum.

STUDY AIDS

Summary

A wide variety of magnetic devices can be used as binary devices in digital systems. By far the most widely used is the magnetic core. Cores can be used to implement various logic functions such as AND, OR, and NOT, and more complicated functions can be formed from combinations of these basic circuits. Magnetic-core shift registers and ring counters can be constructed by using the single-diode transfer loop between cores. Magnetic-core logic is particularly useful in applications experiencing environmental extremes.

Direct-access memories with very fast access times can be conveniently constructed using either magnetic cores or transistors. The most popular method for constructing these memories is the coincident-current technique. Memories constructed using cores are inherently DRO-type memories but can be transformed into NDRO memories by the addition of external logic.

Semiconductor memories constructed from bipolar transistors or MOSFETs are available. Bipolar memories are static memories, but are available as random-access ROMs, or as complete *read/write* units. MOS memories can be either static or dynamic, and are available as RAMs.

Magnetic drums and disks provide larger storage capacities at a lower cost per bit than core-type memories. They do, however, offer the disadvantage of increased access time.

Glossary

access time For a coincident-current memory, it is the time required for one *read/write* cycle. In general, it is the time required to write one word into memory or to read one word from memory.

address A series of binary digits used to specify the location of a word stored in a memory.

coincident-current selection The technique of applying $\frac{1}{2}I_m$ on each of two lines passing through a magnetic device in such a way that the net current of I_m will switch the device.

DRO Destructive readout.

dynamic memory A memory whose contents must be restored periodically.

hysteresis Derived from the Greek word *hysterein*, which means to lag behind.

hysteresis curve Generally a plot of magnetic flux density B versus magnetic force H. Can also refer to the plot of magnetic flux ϕ versus magnetizing current I.

memory cycle In a coincident-current memory system, a *read* operation followed by a *write* operation.

NDRO Nondestructive readout.

RAM Random-access memory.

ROM Read-only memory.

select current I_m The minimum current required to switch a magnetic device.

single-diode transfer loop A method of coupling the output of one magnetic core to the input of the next magnetic core.

squareness ratio A measure of core quality. From the hysteresis curve, it is the ratio B_r/B_m.

static memory A memory capable of storing binary information indefinitely.

Review Questions

1. Name one advantage of a ferrite core over a metal-ribbon core.
2. Name one advantage of a metal-ribbon core over a ferrite core.
3. Describe the method for detecting a stored 1 in a core.
4. Why is a strobing technique often used to detect the output of a switched core?
5. How is core switching time t_s affected by the switching current?
6. Explain why more complicated logic functions using cores can lead to excessive operating times.
7. What is the purpose of the diode in the single-diode transfer loop?
8. Why is a delay in signal transfer between cores desired?
9. Explain how the R and C in Fig. 12-11 introduce a delay in signal transfer between cores.
10. Explain the operation of the *sense wire* in a magnetic-core matrix plane. Why is it possible to thread every core in the plane with the same wire?
11. Explain how it is possible to store a 0 in a coincident-current memory core using the *inhibit* line.
12. Why is a basic coincident-current core memory inherently a DRO-type system?
13. In the basic memory cycle for a coincident-current core memory system, why must the *read* operation come before the *write* operation?
14. What is the difference between the *write into memory* and the *read from memory* cycles for a coincident-current core memory system?
15. Explain the meaning of the title "64-word by eight-bit static RAM."
16. Why are there no dynamic bipolar memories?
17. What does it mean to "refresh" a dynamic memory?

18. Describe the difference between random-access and restricted-access memories.

19. Describe the advantages of using a magnetic-drum storage system.

Problems

12-1. Draw a typical hysteresis curve for a core, and show the two remanent points.

12-2. Show graphically on a ϕI curve the path of the operating point as the core is switched from a 1 to a 0. Repeat for switching from a 0 to a 1.

12-3. Draw the symbol for a magnetic-core logic element, and explain the function of each winding.

12-4. Draw a set of waveforms showing how the exclusive-OR circuit of Fig. 12-8*d* must operate (notice it requires only two clocks which are spaced 180° out of phase).

12-5. Draw a single-diode transfer loop between two cores, and explain its operation (use waveforms if needed).

12-6. Draw a schematic and the waveforms for a core ring counter which provides seven output pulses.

12-7. Draw a sketch and explain how a core can be switched by the coincident-current method.

12-8. Make a sketch similar to Fig. 12-15 showing a three-dimensional core memory capable of storing 100 ten-bit words. Show all input and output lines clearly.

12-9. Describe the geometry of a coincident-current core memory capable of storing 4,096 thirty-six-bit words (i.e., how many planes, how many cores per plane, etc.).

12-10. How many bits can be stored in the memory in Prob. 12-9?

12-11. How many control lines are required for the memory in Prob. 12-9?

12-12. Show graphically the meaning of squareness ratio for a magnetic core, and explain its importance for magnetic-core memories.

12-13. Describe a structure for the address which could be used for the memory of Prob. 12-9.

12-14. If a certain core memory is composed of square matrices, what is the word capacity if the address is 12 binary digits?

12-15. How many bits are required in the address of a 256-word by one-bit *read/write* bipolar RAM?

12-16. Draw the polarity of the stored charge on the gate capacitance shown in the basic dynamic memory cell in Fig. 12-29.

12-17. What is the bit-storage capacity of a magnetic drum 10 in in diameter if data are stored with a density of 200 bits per in in 20 tracks?

12-18. What would be the diameter of a magnetic drum capable of storing 3,140 thirty-six-bit words if there are 10 tracks and data are stored bit-serial at 300 bits per in?

12-19. What is the average access time for the drum in Prob. 12-18 if it rotates at 36,000 rpm? What could be done to reduce this access time by a factor of 2?

12-20. For the drum in Prob. 12-18, at what bit rate must data be moved (i.e., read or write) if the drum rotates at 36,000 rpm?

Introduction to Digital Computers

14

The digital principles discussed in the previous chapters have been utilized to devise a great many different digital systems. The applications are many and varied. They include simple systems such as counters and digital clocks, and more complex applications such as digital voltmeters, A/D converters, frequency counters, and time-period measuring systems. Among the most sophisticated digital systems devised are digital computers, including special-purpose machines, small general-purpose computers (such as the Digital Equipment Corp. PDP-8/E), and large general-purpose computers (such as the IBM 360 and 370 systems). In this chapter we consider some of the basic principles common to digital computer systems.

After studying this chapter you should be able to

1. State the difference between a special purpose and a general purpose digital computer.
2. Discuss the 4 main blocks in a general purpose computer.
3. Write a simple computer program using mnemonic code.

14-1 BASIC CLOCKS

The operation or control of a digital system can be classified in two general categories—synchronous and asynchronous. In a *synchronous* system the flip-flops are controlled by the system clock and can therefore change states only when the clock changes state. Therefore, all the flip-flops and logic gates change levels in time (or in synchronism) with the clock. An example of such a synchronous system is the parallel counter constructed using the master/slave clocked flip-flops. In this counter, the flip-flops can change state only when the clock goes low and at no other time (notice that a system could be constructed such that the flip-flops would change state when the clock goes high). On the other hand, in an *asynchronous* system the flip-flops are controlled by events which occur at random times. Thus

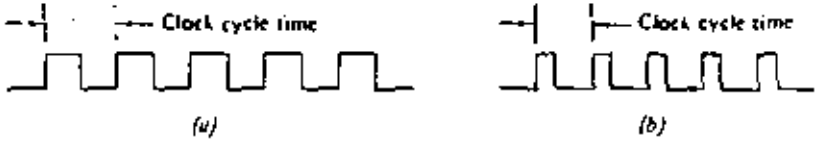


Fig. 14-1. Basic system clock.

the flip-flops may change states at random and are not in synchronism with any timing signal such as a clock. An example of such a system might be the operation of a push button by a human operator. Depression of the push button would cause a flip-flop to change state. Since the operator can depress the button at any time he or she desires, the flip-flop would change states at some random time, and this is therefore an asynchronous operation. Most large-scale digital systems operate in the synchronous mode; if you give a little thought to the checkout and maintenance of such a system, it is easy to see why.

Since all logic operations in a synchronous machine occur in synchronism with a clock, the system clock becomes the basic timing unit. The system clock must provide a periodic waveform which can be used as a synchronizing signal. The square wave shown in Fig. 14-1a is a typical clock waveform used in a digital system. It should be noted that the clock need not be a perfectly symmetrical square wave as shown. It could simply be a series of positive pulses (or negative pulses) as shown in Fig. 14-1b. This waveform could, of course, be considered as an asymmetrical square wave. The main requirement is simply that the clock be perfectly periodic. Notice that the clock defines a basic timing interval during which logic operations must be performed. This basic timing interval is defined as a *clock cycle time* and is equal to one period of the clock waveform. Thus all logic elements, flip-flops, counters, gates, etc., must complete their transitions in less than one clock cycle time.

Example 14-1

What is the clock cycle time for a system which uses a 500-kHz clock? A 2-MHz clock?

Solution

A clock cycle time is equal to one period of the clock. Therefore, the clock cycle time for a 500-kHz clock is $1/(500 \times 10^3) = 2\ \mu s$. For a 2-MHz clock, the clock cycle time is $1/(2 \times 10^6) = 0.5\ \mu s$.

Example 14-2

The total propagation delay through a master/slave clocked flip-flop is given as 100 ns. What is the maximum clock frequency that can be used with this flip-flop?

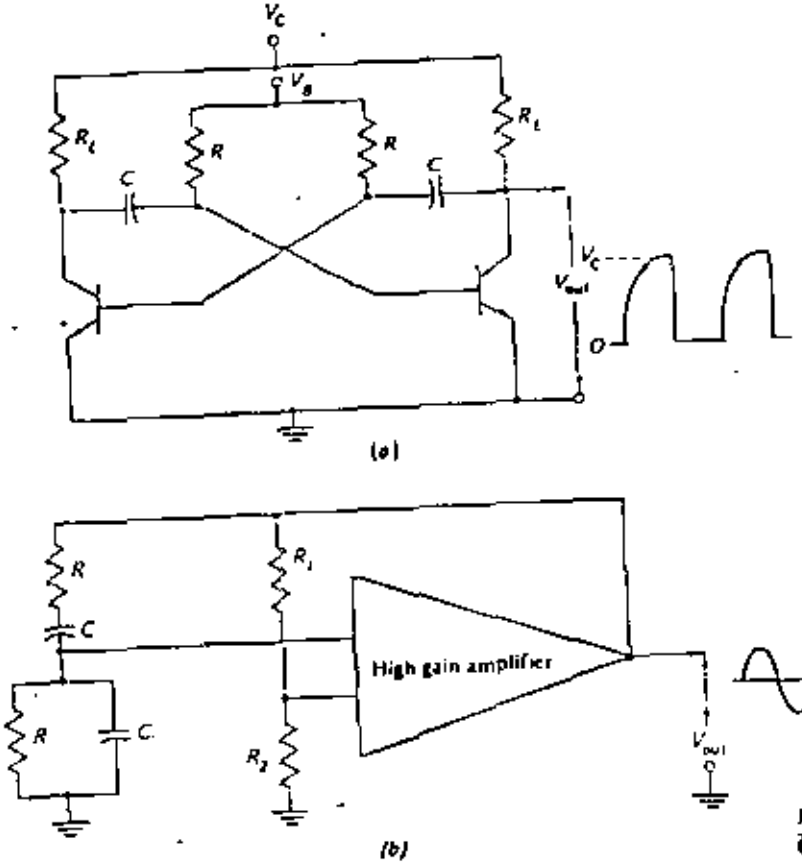
Solution

An alternative way of expressing the question is, how fast can the flip-flop operate? The flip-flop must complete its transition in less than one clock cycle time. Therefore, the minimum clock cycle time must be 100 ns. So, the maximum clock frequency must be $1/(100 \times 10^{-9}) = 10$ MHz.

In many digital systems the clock is used as the basic standard for measurement. For example, the accuracy of the digital clock discussed in Chap. 9 is related directly to the frequency of the clock used to drive the counter. If the clock changes frequency, the accuracy is reduced. For this reason, it is necessary to ensure that the clock maintains a stable and predictable frequency. In many digital systems only short-term stability is required of the clock. This would be the case in a system where the clock could be monitored and adjusted periodically. For such a system, the basic clock might be derived from a free-running multivibrator or a simple sine-wave oscillator as shown in Fig. 14-2a and b. For the free-running multivibrator the clock frequency f is given by

$$f = \frac{1}{2RC \ln (1 + V_C/V_B)} \tag{14.1}$$

Fig. 14-2. Basic clock circuits. (a) Free-running multivibrator. (b) Wien-bridge oscillator.



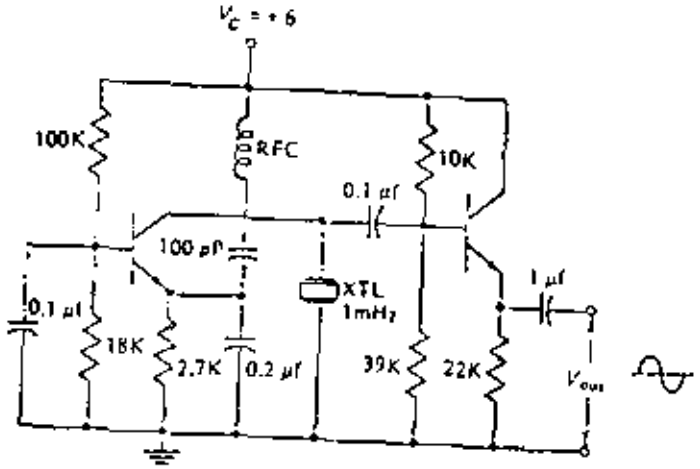


Fig. 14-3. Crystal oscillator.

From Eq. (14-1) it can be seen that the basic clock frequency is affected by the supply voltages as well as the values of the resistors R and capacitors C. Even so, it is possible to construct multivibrators such as this which have stabilities better than a few parts in 10^3 per day. The frequency of oscillation f for the Wien-bridge oscillator is given by

$$f \cong \frac{1}{2\pi RC}. \tag{14-2}$$

Again it is not difficult to construct these oscillators with stabilities better than a few parts in 10^3 per day. If greater clock accuracy is desired, a crystal-controlled oscillator such as that shown in Fig. 14-3 might be used. This type of oscillator is quite often housed in an enclosure containing a heating element which maintains the crystal at a constant temperature. Such oscillators can have accuracies better than a few parts in 10^6 per day.

Example 14-3

The multivibrator in Fig. 14-2a is being used as a system clock and operated at a frequency of 100 kHz. If its accuracy is better than ±2 parts in 10^3 per day, what are the maximum and minimum frequencies of the multivibrator?

Solution

One part in 10^3 can be thought of as 1 cycle in 1,000 cycles. Two parts in 10^3 can be thought of as 2 cycles in 1,000 cycles. Since the multivibrator runs at 100 kHz, two parts in 10^3 is equivalent to 200 cycles. Thus the maximum frequency would be 100 kHz + 200 cycles = 100.2 kHz, and the minimum frequency would be 100 kHz − 200 cycles = 99.8 kHz.

Fig. 14-4. Oscillator and output amplifier.

None of the oscillators shown in Figs. 14-2 and 14-3 has a square-wave output waveform, and it is therefore necessary to convert the basic frequency into a square wave before use in the system. The simplest way of accomplishing this is to use a Schmitt trigger on the output of the basic oscillator as shown in Fig. 14-4. This provides two advantages:

1. It provides a square wave of the basic clock frequency as desired.
2. It ensures that the clock-output amplifier (the Schmitt trigger in this case) has enough power to drive all the necessary circuits without loading the basic oscillator and thus changing the oscillating frequency.

14-2 CLOCK SYSTEMS

Quite often it is desirable to have clocks of more than one frequency in a system. Alternatively, it might be desirable to have the ability to operate a system at different clock frequencies. We might then begin with a basic clock which is the highest frequency desired and develop other basic clocks by simple frequency division using counters. As an example of this, suppose we desire a system which will provide basic clock frequencies of 3, 1.5, and 1 MHz. This could be accomplished by using the clock system shown in Fig. 14-5. We begin with a 3-MHz oscillator followed by a Schmitt trigger to provide the 3-MHz clock. The 3-MHz signal is then fed through one flip-flop which divides the signal by 2 to provide the 1.5-MHz clock. The 3 MHz signal is also fed through a divide-by-3 counter, which provides the 1-MHz clock. Systems having multiple clock frequencies can be provided by using this basic method.

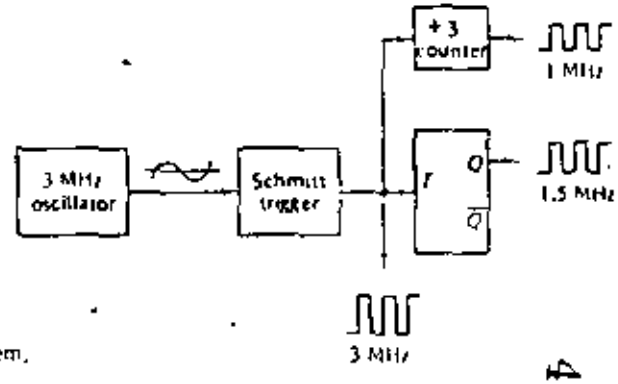


Fig. 14-5. Basic clock system.

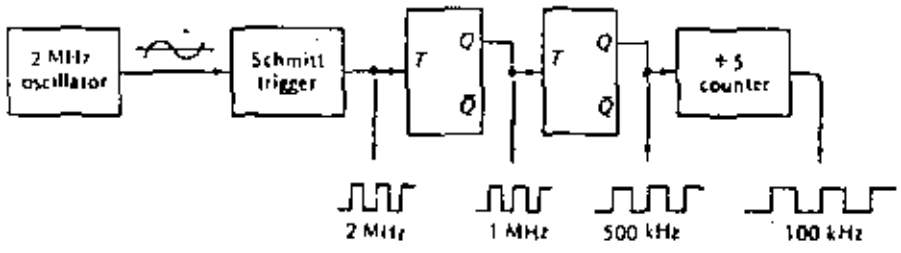


Fig. 14-6. Clock system.

Example 14-4

Show a clock system which will provide clock frequencies of 2 MHz, 1 MHz, 500 kHz, and 100 kHz.

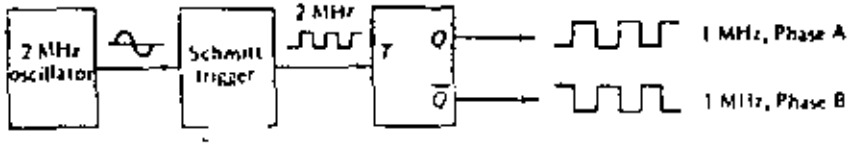
Solution

The desired system is shown in Fig. 14-6. Beginning with a 2-MHz oscillator and a Schmitt trigger, the 2-MHz clock appears at the output of the Schmitt trigger. The first flip-flop divides the 2 MHz signal by 2 to provide the 1 MHz clock. The second flip-flop divides the 1-MHz clock by 2 to provide the 500-kHz clock. Dividing the 500-kHz clock by 5 provides the 100-kHz clock.

It is sometimes desirable to have a two-phase clock in a digital system. A *two-phase clock* simply means we have two clock signals of the same frequency which are 180° out of phase with one another. This can be accomplished with the outputs of a flip-flop. The Q output is one phase of the clock and the $\bar{Q}$ output is the other phase. These two signals are clearly 180° out of phase with one another, since one is the complement of the other. A system for developing a two-phase clock of 1 MHz is shown in Fig. 14-7. For distinction, the two clocks are sometimes referred to as phase A and phase B. You will recall that one use for a two-phase clock system is to drive the magnetic-core shift register discussed in Chap. 12 (Fig. 12-10). It is interesting to note that the two-phase clock system can be used to overcome the race problem encountered with the basic parallel counter discussed in Chap. 8 (Fig. 8-5). The race problem is solved by driving the *odd* flip-flops (i.e., flip-flops A, C, E, etc.) with phase A of the clock, and the *even* flip-flops (i.e., flip-flops B, D, F, etc.) with phase B of the clock (see Prob. 14-12).

The race problem as initially discussed in Chap. 8 can occur any time two or more signals at the inputs of a gate are undergoing changes at the same time. The

Fig. 14-7. 1-MHz two-phase clock.



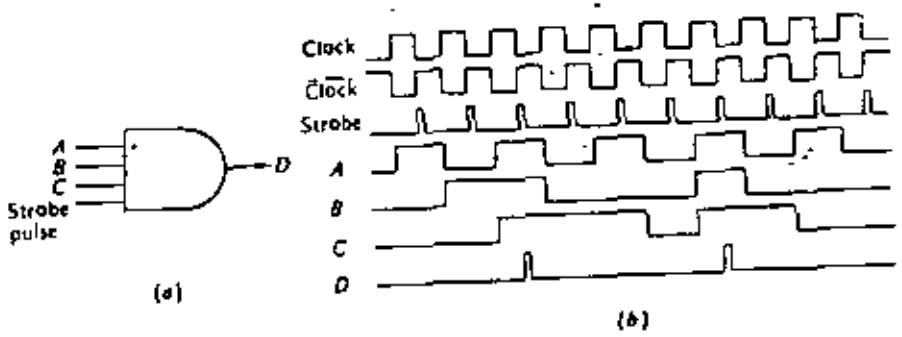


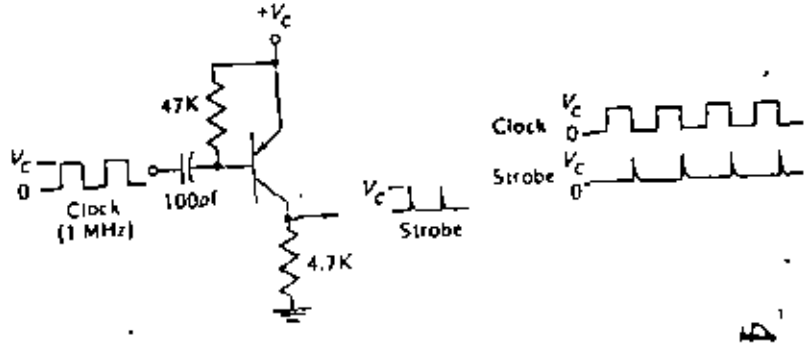
Fig. 14-8. The use of a strobe pulse. (a) Three-input AND interrogated by a strobe pulse. (b) Waveforms for the AND gate.

problem is therefore not unique in counters and can occur anywhere in a digital system. For this reason, a *strobe pulse* is quite often developed using the basic clock. This strobe pulse is used to interrogate the condition of a gate at a time when the input levels to the gate are not changing. If the gate levels render the gate in a true condition, a pulse appears at the output of the gate when the strobe pulse is applied. If the gate is false, no pulse appears. In Fig. 14-8, a strobe pulse is used to interrogate the simple three-input AND gate. The waveforms clearly show that outputs appear only when the three input levels to the gate are true. It is also quite clear that no racing can possibly occur since the strobe pulses are placed exactly midway between the input-level transitions. The strobe signal can be developed in a number of ways. One way is to differentiate the complement of the clock, $\overline{\text{clock}}$, and use only the positive pulses. A second method would be to differentiate the clock and feed it into an "off" transistor as shown in Fig. 14-9.

14-3 MPC COMPUTER

Up to this point we have covered quite a wide variety of the topics generally encountered in the study of digital systems. Some of the topics have been discussed in

Fig. 14-9. Developing a strobe pulse.



great detail, while others have been treated in a more general way. In any case you should now have the necessary background to study any digital system with good comprehension and a minimum of effort. Even so, you may be somewhat unsure about the overall organization of a digital system. In an effort to overcome this feeling and to attempt to tie together many of the topics discussed in the previous chapters, we shall at this time consider the implementation of a small special-purpose digital computer.

The special-purpose computer we shall consider will be used to calculate the *miles per gallon* of a motor vehicle, thus the name *MPG computer*. It is a *special-purpose computer* since this is the only use for which it is intended. A *general-purpose computer* would be a more complicated machine which might be used for a number of different applications.

The first step in the design of the MPG computer must necessarily be the determination of the system performance requirements. The first requirement might be that the system be capable of operating from a supply voltage of ±6 or ±12 V dc since the machine will be operated in a motor vehicle. The second requirement might be that the readout of the computer be in decimal form. Nixie tubes might be good for the readout, but they require an additional power supply of around +100 V to operate the tubes. Digital modules are commercially available which provide decimal readout, and they operate on +6 or +12 V dc. These modules do not require the +100 V, and might be a better choice in this case. The final decision will be one of economics. The third requirement is that the computer calculate the miles per gallon used by the vehicle to an accuracy of ±1 mile per gallon. The fourth requirement we shall impose is that the computer perform a calculation at least once every 15 s when the vehicle is traveling at a speed greater than 10 mph. In other words, we would like to sample the mileage performance of the vehicle at least once every 15 s (faster sampling rates are acceptable). The fifth requirement is that the computer be capable of operating in vehicles using fuel at rates between 10 and 40 miles per gallon. We can now summarize the five basic requirements of the MPG computer as follows:

1. Power-supply voltage is either ±6 or ±12 V dc.
2. The computer must provide a decimal readout in miles per gallon.
3. The computer must provide the readout to an accuracy of ±1 mile per gallon.
4. The computer must provide a readout of miles per gallon at least once every 15 s when the vehicle is traveling at a speed greater than 10 mph.
5. The computer must be capable of calculating miles per gallon between the limits of 10 and 40 miles per gallon.

It should be noted that the system requirements for the computer under study here are quite simple and somewhat less stringent than in the usual case. The requirements here are intentionally made simple in order to simplify the discussion. Nevertheless the principles are the same regardless of the severity of the system specifications, and the study is therefore instructive.

We assume that we have available two transducers which are to be used as an integral part of the MPG computer. The first transducer is used to measure the volume of fuel flowing into the engine. This flow tranducer provides an electrical pulse each time $\frac{1}{1000}$ of a gallon of fuel passes through it. The second transducer is used to measure the distance traveled and is driven by the speedometer cable. This distance transducer provides an electrical pulse each time the vehicle has traveled a distance of $\frac{1}{1000}$ of a mile.

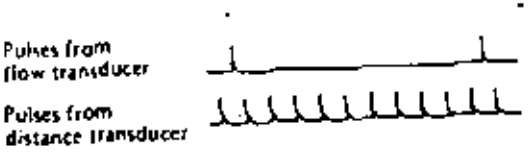


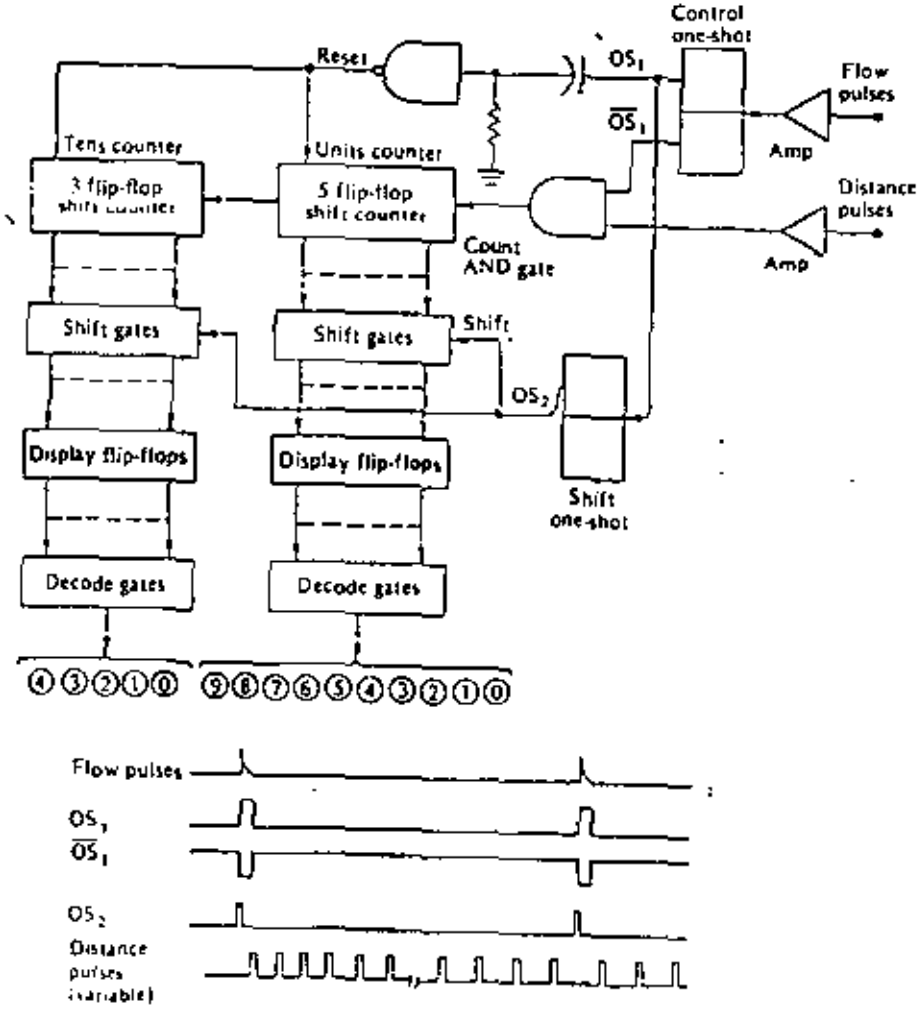
Fig. 14-10. Transducer pulses for the MPG computer when the rate is 10 miles per gallon.

Now in order to implement the necessary logic for the computer, let us examine the outputs of the flow and distance transducers. Let us begin by assuming that we have a flow transducer which gives an output pulse each time 1 gallon is used, and we have a distance transducer which gives an output pulse each time the vehicle has traveled 1 mile. If our vehicle is obtaining a mileage slightly better than 10 miles per gallon, the transducer waveforms appear as shown in Fig. 14-10. Notice that the number of distance pulses appearing between two flow pulses is exactly equal to the miles per gallon we desire. Thus we can *calculate* the miles per gallon by simply counting the number of distance pulses occurring between two flow pulses. We can check this by noting that, if the vehicle were operating at 20 miles per gallon, there would be 20 distance pulses between two flow pulses. Notice that if the flow transducer supplied 10 pulses per gallon, and at the same time the distance transducer provided 10 pulses per mile, the basic waveform in Fig. 14-10 would remain unchanged. That is, the number of distance pulses appearing between two flow pulses would still be equal to the number of miles per gallon. From this it should be clear that we can choose any number of pulses per gallon from the flow transducer so long as we choose the same number of pulses per mile from the distance transducer. The transducers we are going to use in the MPG computer provide 1,000 pulses per gallon of flow and 1,000 pulses per mile of distance. Therefore, the number of miles per gallon can be obtained by simply counting the number of distance pulses between consecutive flow pulses.

The reason for using these transducers can be seen by examining the time between flow pulses. Let us first consider the flow transducer having one pulse per gallon and the distance transducer having one pulse per mile. If the vehicle were obtaining a rate of 10 miles per gallon, one flow pulse would occur every 10 miles. If the vehicle were traveling at a speed of 10 mph, the flow pulses would occur at a rate of one per hour. This is clearly not a fast enough sampling rate. On the other hand, with the specified transducers, the flow pulses occur at a rate of 1,000 pulses per gallon and at the rate of 1,000 pulses per hour under the same conditions. Thus the flow pulses occur every 1 hr/1000 = 3.6 s. This sampling time is clearly within the specified rate. The worst case occurs when the vehicle obtains the maximum miles per gallon. At 40 miles per gallon and 10 mph the flow pulses occur every $3.6 \times 4 = 14.4$ s. We have therefore met the minimum-sampling-time requirements.

The logic diagram for the MPG computer can now be drawn; it is shown in Fig. 14-11 along with the complete waveforms. The flow pulses are fed into a conditioning amplifier and then into a one-shot to develop the waveform OS_1 and $\overline{OS}_1$. The distance pulses are also fed into a conditioning amplifier. Since we desire to count the number of distance pulses occuring between two pulses, we use the distance pulses as one input to the COUNT AND gate. If $\overline{OS}_1$ is used as the other input to this AND gate, it is enabled between flow pulses, and the distance pulses appear at its output. We use the pulses appearing at the output of the COUNT AND gate to drive a counter. Since we desire to display the miles per gallon between the limits

Fig. 14-11. Complete MPG computer.



of 10 and 40, we use a five-flip-flop shift counter for the *units* digits, and a thr flip-flop shift counter for the *tens* digits of miles per gallon.

One *conversion time* is the time between two flow pulses, and we want to s the accumulated count into the display flip-flops at the end of each convers cycle. Notice first of all that, when $\overline{OS}_1$ is low, the count AND gate is disabled a therefore the *units* and *tens* counters cannot change states. It is during this time t we must shift the contents of these counters into the display flip-flops. We use leading edge of OS_1 to trigger the *shift* one-shot and develop the shift wavef OS_2. The falling edge of OS_2 is applied to the shift gates, and at this time the co stored in the *units* and *tens* counters is shifted into the display flip-flops. The fall edge of OS_1 is then used to reset all flip-flops in the *units* and *tens* counters. contents of the display flip-flops are then decoded and used to illuminate the dicator lights. In this system, the distance pulses can be considered to be the b system clock. The flow pulses form a variable control gate by means of the con one-shot which determines the period of time that the count AND gate is enal and therefore the number of distance pulses counted. The output of the *shift* c shot OS_2 can be considered as a strobe pulse which shifts data from the coun into the display flip-flops in such a way that racing is avoided. The system cle has an accuracy of ± one count, which corresponds to ±1 mile per gallon

14-4 GENERAL-PURPOSE COMPUTER

The MPG computer discussed in the previous section is considered a special- pose computer since it is designed and constructed to perform a single functio alter it so that it could perform another function would require a major chan design. On the other hand, a general-purpose computer is designed so that it perform a number of fundamental operations—addition, subtraction, multip tion, division, comparison, etc. The computer can then be used in any numb different applications by simply instructing it to perform the appropriate opera in an orderly fashion. The functions to be performed, listed in the order in w they are to be accomplished, is known as a *program* (instruction set). This list structions, or program, is normally stored in the computer memory; when the puter is started, it simply performs these instructions in the order stored. H lies the difference between an electronic calculator and a general-purpose d computer—the calculator performs a function (add, subtract, etc.) each tim operator depresses a button, but the stored-program computer performs the plete list of stored instructions without human intervention. Furthermore, the puter is capable of completing the instruction set in a very short period of (addition in perhaps a few microseconds), and the operation is virtually error

The simplified block diagram in Fig. 14-12 shows the basic units to be fou any general-purpose computer system. The *input/output* block represents the face between man and machine. It could simply be a teletype unit, where inp formation is typed in on the keyboard and output information is printed on pa could also represent any of the other input/output medi eviously discussed, as punched paper tape, punched unit-record cards, ar netic tape. In any

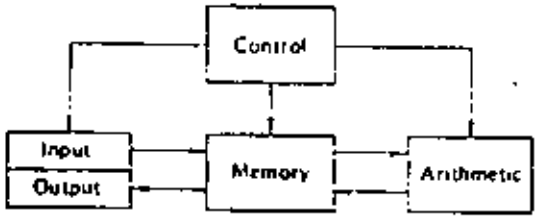


Fig. 14-12. Basic computer unit block diagram.

input data are taken into the system and stored in the memory according to the appropriate signals as generated by the control block. Similarly, the control unit generates the appropriate signals to read data from the memory and move it to the output block.

The arithmetic unit consists of the registers, counters, and logic required for the basic operations, including addition, subtraction, complementation, shifting right or left, comparison, etc. Since the manipulation of data is accomplished in this unit, it is sometimes referred to as the *central processing unit* (CPU). The topics previously covered (number systems, digital arithmetic, etc.) provide an insight into the logic circuits and configurations required in a CPU. Again, the control unit provides the necessary signals to move data from the memory unit to the arithmetic unit, perform the desired data manipulation, and move the resulting data back into memory.

The memory block represents the area used to store the two types of information present in the computer; namely, the list of instructions (program) and the data to be operated on as well as the resulting output data. The memory itself could be constructed using any of the devices previously discussed—magnetic cores, magnetic drums or disks, semiconductor memory units, magnetic tapes, and so on. Reading data from or writing data into the memory is again under the guidance of the control unit.

The control unit generally contains the counters, registers, and logic necessary to develop the control signals required for moving data into and out of the memory, and for performing the necessary data manipulations in the arithmetic unit. The system clock is a part of the control unit, and it is usually the starting point for generating the proper control signals as discussed in the first part of this chapter.

It is interesting to consider an actual general-purpose digital computer in light of the above discussion. For this purpose, a block diagram of the Digital Equipment Corp. PDP-8/E is shown in Fig. 14-13.[1] Note how the system diagram can be broken into the four basic blocks previously discussed—input/output, arithmetic, memory, and control. A table-model PDP-8/E is shown in Fig. 14-14, and the following excerpt gives a general description of the system.[2]

> The PDP-8/E is specially designed as a general perpose computer. It is fast, compact, inexpensive, and easy to interface. The PDP-8/E is designed to meet

[1] "Small Computer Handbook," chap. 1, Digital Equipment Corporation, Maynard, Mass., 1971.
[2] Ibid.

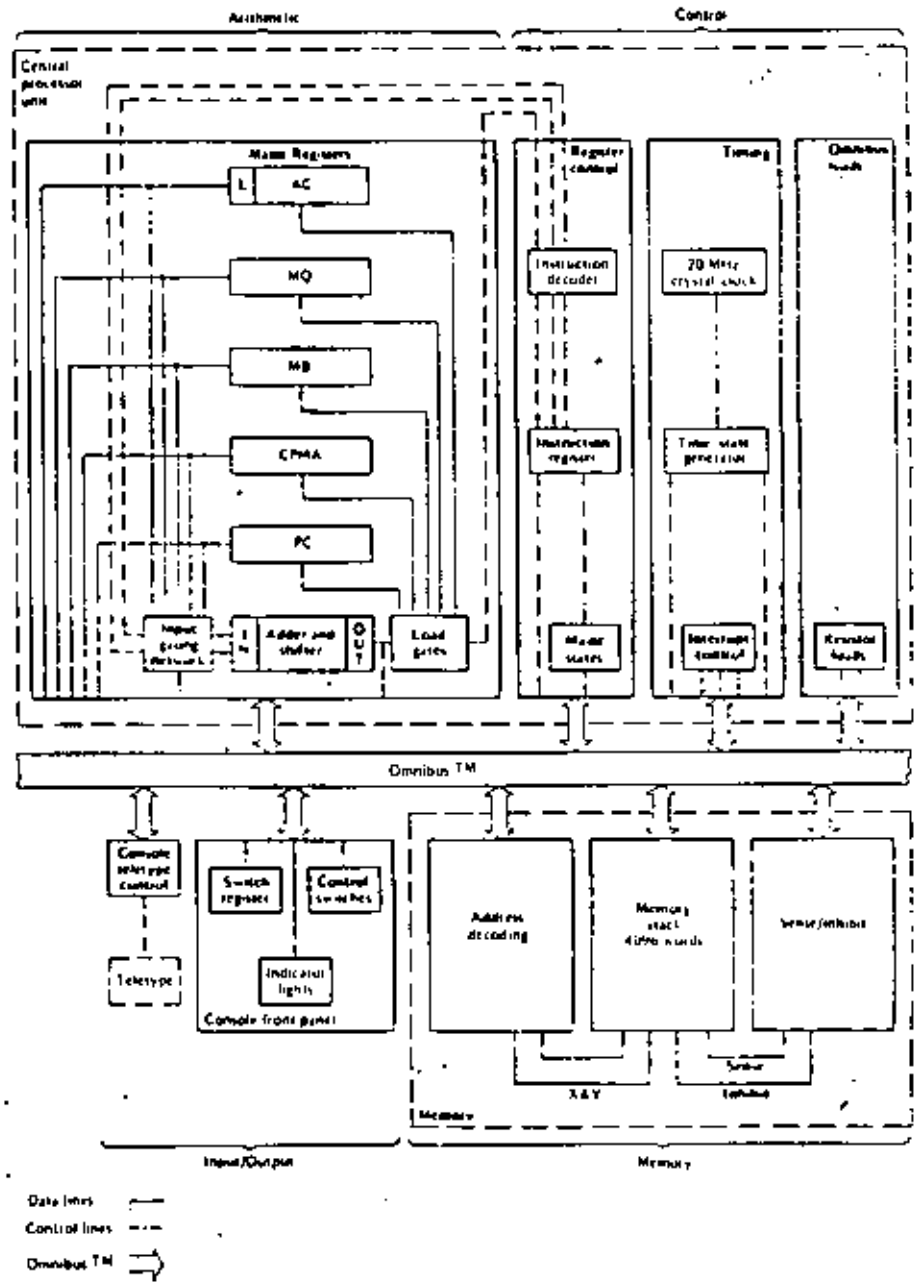


Fig. 14-13. PDP-8/E basic system block diagram.

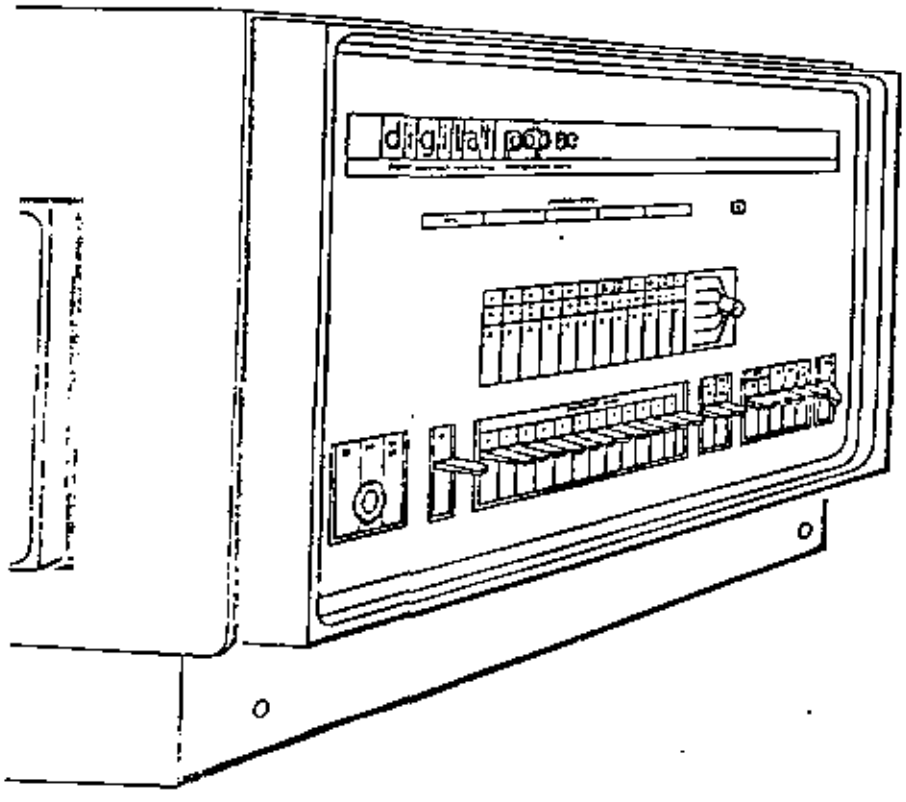


Fig. 14-14. PDP-8/E programmed data processor.

the needs of the average user and is capable of modular expansion to accomodate most individual requirements for a user's specific applications.

The PDP-8/E basic processor is a single-address, fixed word length, parallel-transfer computer using 12-bit, 2's complement arithmetic. The cycle time of the 4096-word random address magnetic core memory is 1.2 microseconds for fetch and defer cycles without autoindex; and 1.4 microseconds for all other cycles. Standard features include indirect addressing and facilities for instruction skip and program interrupt as a function of the input/output device condition.

Five 12-bit registers are used to control computer operations, address memory, operate on data and store data. A Programmer's console provides switches to allow addressing and loading memory and indicators to observe the results. The PDP-8/E may also be programmed using the console Teletype with a reader/punch facility. Thus, programs can be loaded into memory using the switches on the Programmer's console, the Teletype keyboard, or the paper tape reader. Processor operation includes addressing memory, storing data, retrieving data, receiving and transmitting data and mathematical computations.

The 1.2/1.4 microsecond cycle time of the machine provides a computation rate of 385,000 additions per second. Each addition requires 2.6 microseconds (with one number in the accumulator) and subtraction requires 5.0 microseconds (with the subtrahend in the accumulator). Multiplication is performed in 256.5 microseconds or less by a subroutine that operates on two signed 12-bit numbers to produce a 24-bit product, leaving the 12 most significant bits in the accumulator. Division of two signed 12-bit numbers is performed in 342.4 microseconds or less by a subroutine that produces a 12-bit quotient in the accumulator and a 12-bit remainder in core memory. Similar signed multiplication and division operations are performed in approximately 40 microseconds, utilizing the optional Extended Arithmetic Element.

The flexible, high-capacity input/output capabilities of the computer allow it to operate a large variety of peripheral machines. Besides the standard keyboard and paper-tape punch and reader equipment, these computers are capable of operating in conjunction with a number of optional devices (such as high-speed perforated-tape punch and reader equipment, card reader equipment, line printers, analog-to-digital converters, cathode ray tube (CRT) displays, magnetic tape equipment, a 32,764-word random-access disk file, a 262,112-word random-access disk file, etc.).

14-5 COMPUTER ORGANIZATION AND CONTROL

In this short chapter devoted to digital computers, we cannot possibly give an exhaustive treatment of all machines; however, we can discuss in general terms those aspects of computer organization and operation which are common to many different types of digital computers.

The information stored in the computer memory is of two types—either *data words* (numeric information) or *instruction words*. In Sec. 13-1, we considered in some detail the various formats available for storing numbers, including both fixed-point and floating-point numbers. We must now consider an appropriate format for a computer instruction word.

In general, a computer instruction word will have two distinct sections, as shown in Fig. 14-15. In this case the word length is 12 bits; however, the number of bits in a word varies from machine to machine (e.g., 36 in the IBM 7090/7094, 32 in the IBM 360, 36 in the GE 635, and 12 in the PDP-8/E). The first section (the three bits on the left in this case) are used for the *operation code* (op-code) of the instruction to be performed. The op-codes are defined by the computer designer when the machine is initially designed. For example, the op-code for addition might be defined as 001_2. In this case, there are only three bits reserved for op-codes, and a computer using this format would therefore be limited to $2^3 = 8$ op-codes.

The remaining bits in the instruction word shown in Fig. 14-15 are used to specify the address in memory to which the instruction applies. In this case, the nine bits can be used to specify any one of $2^9 = 512$ locations in memory. As an example, the instruction word 001 000001100 means add (001) the contents of the

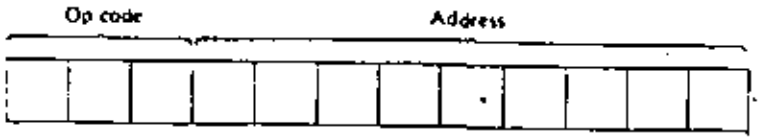


Fig. 14-15. Instruction word format.

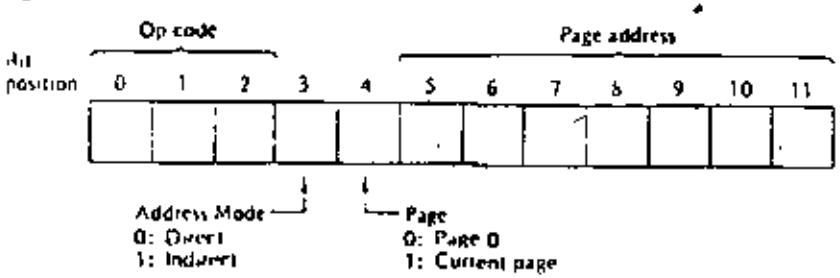
memory located at address 12_{10} (000001100) to the contents of the accumulator register in the arithmetic unit.

Frequently the memory is broken up into sections called "pages" in order to provide for more efficient addressing. For example, the PDP-8/E has a basic memory of 4,096 twelve-bit words. The memory is broken up into 32 pages of 128 words on each page. Thus any word on a page can be addressed by means of only seven bits ($2^7 = 128$). The instruction word for the PDP-8/E is then arranged as shown in Fig. 14-16. If the address mode bit (bit 3) is 0, the op-code simply refers to one of the 128 page addresses given by the last seven bits in the word. However, if the address mode bit is 1, *indirect addressing* is indicated. This means the control unit will go either to page 0 or remain on the current page (depending on whether bit 4 is 1 or 0), take the contents of the given address, and treat it as another address. The first five bits of this new address specify which of the 32 pages ($2^5 = 32$), and the remaining seven bits give the address on that page ($2^7 = 128$) containing the data to which the op-code applies.

In this way, the instruction word format need only have seven bits devoted to an address, and only an occasional 12-bit address word is needed to reference data on any one of the other 31 available pages. Clearly this word format is more efficient than simply carrying 12 ($2^{12} = 4,096$) bits for address locations in memory.

As an example of indirect addressing, suppose the data being operated on are stored on page 15 of the memory—in order to get to another page, one must use indirect addressing. The instruction word 001 10 0001110 means add (001) the contents of the data located in address 14_{10} (0001110) on page 0 to the contents of the accumulator register in the arithmetic unit. Note that the 1 in the fourth bit position specifies indirect addressing, and the 0 in the fifth bit position refers to page 0. Now, if the contents of memory location 14_{10} on page 0 is 00101 0001111, the data to be added to the accumulator will be found on page 5_{10} (00101) in location 15_{10} (0001111).

Fig. 14-16. PDP-8/E instruction word format.



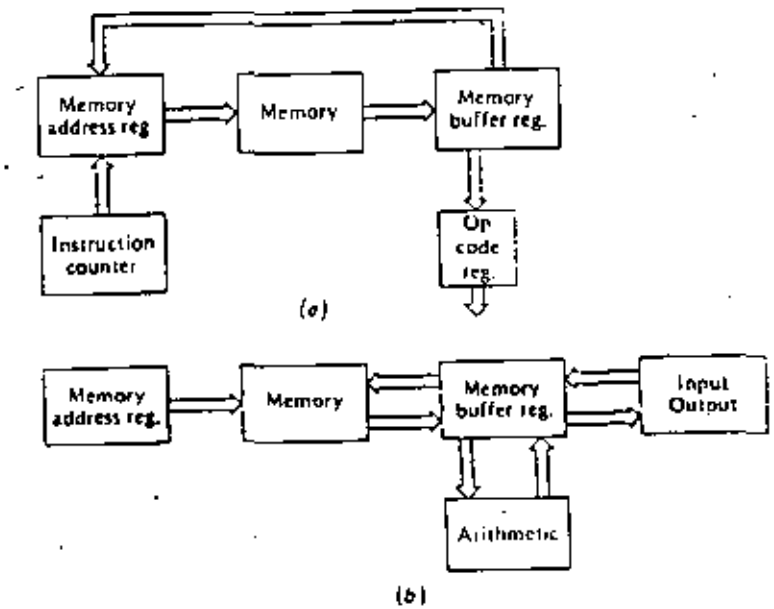


Fig. 14-17. Basic computer operating cycles. (a) *Fetch*. (b) *Execute*.

The instructions to be executed by the computer are normally stored in the memory in the order in which they are to be performed. To begin an operation, the address in the memory of the first instruction to be executed is entered into the machine by an operator. The control unit then *fetches* this instruction from memory, executes the proper operation, and proceeds to the next instruction stored in the memory. This basic two-cycle process continues until all the instructions have been completed and the machine stops. Thus the operation of a computer can be explained in terms of two fundamental cycles—*fetch* and *execute*. Let's examine these two cycles and determine the tasks to be accomplished by the control unit during each cycle.

The computer units involved during a fetch cycle are shown in Fig. 14-17a. During a *fetch* cycle, the following operations are performed:

1. The address in memory of the first instruction to be executed is placed in the instruction counter. This address is read into the memory address register (MAR) and a *read/write* cycle is initiated in the memory.
2. The instruction stored at the given address in memory is read into the memory buffer register (MBR).
3. The op-code portion of the instruction in the MBR is then stored in the op-code register, and the address portion is placed in the MAR (in place of the previous address) in preparation for the following execute cycle.
4. The instruction counter is increased by one in order to be ready for the next fetch cycle.

The computer units active during an execute cycle are shown in Fig. 14-17*b*, and the following operations are performed:

1. The address in memory containing data to be read out, or where data is to be stored, is contained in the MAR as a result of the previous *fetch* cycle. Similarly, the op-code is contained in the op-code register.
2. The contents of the op-code register are decoded and the control unit provides the necessary control signals to perform the operation called for—e.g. read data from an input TTY, into the MBR and store it at the address in memory according to the contents of the MAR; or, read data from the address in memory as given by the MAR, and move it to the arithmetic unit via the MBR; or, read data from the memory via the MBR and print the data on a TTY; or, read data from the arithmetic unit via the MBR and store it in the memory at the address specified by the MAR.
3. At the completion of the execute cycle, return to the next *fetch* cycle.

The *fetch/execute* method of operation is quite common to most general-purpose digital computers, even though the two states might be referred to by different names. When an operation is begun, the control unit first places the computer in the *fetch* mode, and thereafter alternates *execute* and *fetch* modes until the desired operation is complete. A series of clock pulses (perhaps four or five, or even ten) during each *fetch* cycle is used to time the various operations. A similar sequence of clock pulses is utilized during the *execute* cycle.

14-6 COMPUTER INSTRUCTIONS

Every general-purpose computer must have an instruction set. There may be only a few (10 or so) for a small computer, while a large computer may have hundreds of instructions. The set of instructions used with any particular computer is of course devised during the initial design phases, and anyone who uses that computer must become intimately familiar with its instruction set. Incidently, an individual who specializes in efficiently arranging computer instructions for the purpose of solving problems is known as a *computer programmer*.

Inside the computer, every instruction must be represented as a group of binary numbers (e.g., 001 for addition), but to ease the burden of the programmer, the op-codes are frequently assigned *mnemonic* titles. For example, the op-code for addition might be 001, but we could code it as ADD. The programmer could then use ADD in arranging his list of instructions, and when the alphanumeric input ADD appeared at the computer input, it would simply be encoded as the instruction 001.

In general, there are four different types of instructions—arithmetic, data manipulation, transfer, and input/output. Let's list a ficticious set of instructions and then see how they might be arranged as a program to solve a problem. Even though this instruction set is ficticious, it is quite similar to those found in actual computer systems. Each instruction is given in mnemonic form, with its binary code in parenthesis, and a description of the operation it requires.

HLT (0000) Halts computer operation. Operator may restart by depressing the start button.

ADDX (0001) The content of memory location X is added to the content of the accumulator register in the arithmetic unit.

SUBX (0010) The content of memory location X is subtracted from the content of the accumulator register in the arithmetic unit.

MPYX (0011) The content of memory location X is multiplied by the content of the MQ register in the arithmetic unit, and the product is stored in the MQ register.

DIVX (0100) The content of memory location X is divided into the content of the MQ register, and the quotient is stored in the MQ register.

DCAX (0101) The content of the accumulator is stored in memory location X, and the accumulator is cleared to all zeros.

DCQX (0110) The content of the MQ register is stored in memory location X, and the MQ register is cleared to all zeros.

JMPX (0111) The next instruction is taken from memory location X.

LDQX (1000) The content of memory location X is entered into the MQ register.

REDX (1001) One word of data is read at the input device and stored in memory at address X.

PRTX (1010) One word of data is read from memory at address X and printed on the output device.

This list of instructions is of course not complete enough to allow every possible operation, but it allows us to illustrate basic *machine-language programming*. Notice that there are four bits in each op-code; this is necessary since we want to include more than eight but fewer than 16 instructions. Further, suppose these instructions are used in a small general-purpose computer having only 128 memory

Table 14-1

Operation	Instruction	Memory location	Instruction as stored in memory
Read *R* and store at memory address 50.	RED 50	0	1001 0110010
Read *A* and store at memory address 51.	RED 51	1	1001 0110011
Read *Y* and store at memory address 52.	RED 52	2	1001 0110100
Clear MQ register	DCQ 127	3	0110 1111111
Clear accumulator	DCA 127	4	0101 1111111
Put *A* in MQ	LDQ 51	5	1000 0110011
Multiply *A* by *Y*	MPY 52	6	0011 0110100
Store *AY* in 53	DCQ 53	7	0110 0110101
Put *R* in accumulator	ADD 50	8	0001 0110010
Add *AY* to *R* in accumulator	ADD 53	9	0001 0110101
Store *Z* in 54	DCA 54	10	0101 0110110
Print out *Z*	PRT 54	11	1010 0110110
Halt	HLT	12	0000 0000000

locations so that an instruction word is composed of 11_{10} bits—four bits of op-code and seven bits for memory address.

Now, let's utilize the instructions for our fictitious computer to solve the problem $Z = R + AY$. The program will read the values of R, A, and Y, perform the necessary calculations, and print out the value of Z. The complete program, as written in machine language (mnemonic code) and as stored in memory, would appear as in Table 14-1.

To initiate the program, the operator sets the instruction counter at 0 and depresses the start button. The computer initiates a *fetch* cycle and obtains the first instruction (RED 50) from memory address 0. This is followed by an *execute* cycle. The next *fetch* cycle obtains the instruction in memory address 1, and so on. The program ends after the computed value for Z is printed out and the HLT instruction is obtained in memory address 12_{10}.

STUDY AIDS

Summary

There are basically two types of digital computers—special purpose and general purpose. Special-purpose computers are designed for a single purpose only, while general-purpose machines can be used in any number of different applications. A general-purpose machine is designed with a basic set of instructions, and a programmer can use such a computer to solve specific problems. The computer solves problems by executing a set of instructions which have been ordered and placed in the computer memory by a programmer. Most computers operate in a basic two-cycle *fetch/execute* mode, and the appropriate control signals are generated in the control unit in synchronism with the system clock.

Glossary

asynchronous system A system in which logic operations and level changes occur at random times.

clock cycle time One clock period; the reciprocal of clock frequency.

computer program A list of specific instructions which a computer executes to solve a given problem.

fetch/execute The two alternating modes of operation in a general-purpose computer.

general-purpose computer A computer designed to accomplish a number of tasks. For example, all the arithmetic operations as well as decision making (i.e., equal to, greater than, less than, go, no go).

instruction word A computer word having two sections, the op-code section and the address section.

mnemonic Intended to assist the memory.

op-code Operation code. The code which defines a specific computer operation.

oscillator stability The stability of the frequency of oscillation; usually expressed in parts per thousand or parts per million for a period of time.

secondary clock A clock of frequency lower than the basic system clock which is derived from the basic system clock.

special-purpose computer A computer designed to accomplish only one task, for example, the MPG computer in this chapter.

strobe pulse A pulse developed to interrogate gates or to shift data at a time such that racing is avoided.

synchronous system A system in which logic operations and level changes occur in synchronism with a system clock.

two-phase clock The use of two clock waveforms of the same frequency which are 180° out of phase with one another, for example, the 1 and 0 outputs of a flip-flop.

Review Questions

1. Explain why a clock must be perfectly periodic.
2. How can the clock cycle time be found from the clock frequency?
3. Why must flip-flops have a delay time less than one clock cycle time?
4. What factors affect the oscillating frequency of the multivibrator in Fig. 14-2?
5. What is the purpose of the Schmitt trigger in Fig. 14-4?
6. Explain one method for obtaining a two-phase clock.
7. What is the main purpose for developing a strobe pulse?
8. Why is it advantageous to develop the strobe pulse in Fig. 14-9 by turning the transistor on rather than off?
9. Explain the difference between special- and general-purpose computers.
10. What is a computer program?
11. Explain what is meant by fetch and execute in terms of computer operation.

Problems

14-1. Beginning with a symmetrical square wave, show a method for developing a clock consisting of a series of positive pulses. A series of negative pulses.

14-2. What is the clock cycle time for a system using a 1-MHz clock? A 250-kHz clock?

14-3. What is the maximum delay time for a flip-flop if it is to be used in a system having an 8-MHz clock?

14-4. At what frequency will the multivibrator in Fig. 14-2*a* oscillate if $R = 100$ kΩ, $C = 100$ pF, $V_c = 20$ V dc, and $V_a = 10$ V dc?

14-5. What will be the frequency of the multivibrator in Prob. 14-4 if V_a is changed to 20 V dc?

14-6. What value of C is required for the multivibrator in Fig. 14-2a if $V_c = V_{cc}$, $R = 47$ kΩ, and the desired frequency is 100 kHz?

14-7. What is the oscillating frequency of the Wien-bridge oscillator in Fig. 14-2b if $R = 47$ kΩ, and $C = 100$ pF?

14-8. If the crystal oscillator in Fig. 14-3 has a stability of ±3 parts in 10^7 per day, what are the maximum and minimum frequencies of the oscillator?

14-9. Show the logic necessary to develop clock frequencies of 5 MHz, 2.5 MHz, 1 MHz, and 200 kHz.

14-10. The 5-MHz oscillator in Prob. 14-9 has a stability of ±1 part in 10^6 per day. What will be the maximum and minimum frequency of the 1-MHz clock?

14-11. What would be the maximum and minimum frequency of the 200-kHz clock in Prob. 14-10?

14-12. Draw the waveforms for a parallel binary counter being driven by a two-phase clock. Show that this will result in a solution to the race problem. Remember that each flip-flop has a finite delay time.

14-13. How could the MPG computer be modified to give a solution to the nearest 1/10 mile per gallon?

14-14. Draw a block diagram showing the four major blocks in a general-purpose computer system.

14-15. How many op-code bits would be required in a machine having 35 instructions?

14-16. How many address bits would be required to handle 1,000 words of memory?

14-17. How many page address bits would be required to form a 16-page memory having 64 words per page?

14-18. Write a machine-language program to solve the problem $Z = 3R/(A + B)$.

Appendix A

States and Resolution for Binary Numbers

Word length in bits n	Max number of combinations 2^n	Resolution of a binary ladder ppm
1	2	500 000.
2	4	250 000.
3	8	125 000.
4	16	62 500.
5	32	31 250.
6	64	15 625.
7	128	7 812.5
8	256	3 906.25
9	512	1 953.13
10	1 024	976.56
11	2 048	488.28
12	4 096	244.14
13	8 192	122.07
14	16 384	61.04
15	32 768	30.52
16	65 536	15.26
17	131 072	7.63
18	262 144	3.81
19	524 288	1.91
20	1 048 576	0.95
21	2 097 152	0.48
22	4 194 304	0.24
23	8 388 608	0.12
24	16 777 216	0.06

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

ARQUITECTURA DE PDP-11

DR. ADOLFO GUZMAN ARENAS

JUNIO, 1980

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 UNIBUS

Most computer system components and peripherals connect to and communicate with each other on a single high-speed bus known as the UNIBUS— a key to the PDP-11's many strengths. Addresses, data, and control information are sent along the 56 lines of the bus.

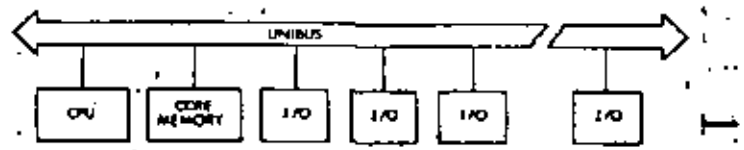


Figure 2-1 PDP-11 System Simplified Block Diagram

The form of communication is the same for every device on the UNIBUS. The processor uses the same set of signals to communicate with memory as with peripheral devices. Peripheral devices also use this set of signals when communicating with the processor, memory or other peripheral devices. Each device, including memory locations, processor registers, and peripheral device registers, is assigned an address on the UNIBUS. Thus, peripheral device registers may be manipulated as flexibly as core memory by the central processor. All the instructions that can be applied to data in core memory can be applied equally well to data in peripheral device registers. This is an especially powerful feature, considering the special capability of PDP-11 instructions to process data in any memory location as though it were an accumulator.

2.1.1 Bidirectional Lines

With bidirectional and asynchronous communications on the UNIBUS, devices can send, receive, and exchange data independently without processor intervention. For example, a cathode ray tube (CRT) display can refresh itself from a disk file while the central processor unit (CPU) attends to other tasks. Because it is asynchronous, the UNIBUS is compatible with devices operating over a wide range of speeds.

2.1.2 Master-Slave Relation

Communication between two devices on the bus is in the form of a master-slave relationship. At any point in time, there is one device that has control of the bus. This controlling device is termed the "bus master." The master device controls the bus when communicating with another device on the bus, termed the "slave." A typical example of this relationship is the processor, as master, fetching an instruction from memory (which is always a slave). Another example is the disk, as

master, transferring data to memory, as slave. Master-slave relationships are dynamic. The processor, for example, may pass bus control to a disk. The disk, as master, could then communicate with a slave memory bank.

Since the UNIBUS is used by the processor and all I/O devices, there is a priority structure to determine which device gets control of the bus. Every device on the UNIBUS which is capable of becoming bus master is assigned a priority. When two devices, which are capable of becoming a bus master, request use of the bus simultaneously, the device with the higher priority will receive control.

2.1.3 Interlocked Communication

Communication on the UNIBUS is interlocked so that for each control signal issued by the master device, there must be a response from the slave in order to complete the transfer. Therefore, communication is independent of the physical bus length (as far as timing is concerned) and the timing of each transfer is dependent only upon the response time of the master and slave devices. The asynchronous operation precludes the need for synchronizing with, and waiting for, clock impulses. Thus, each system is allowed to operate at its maximum possible speed.

Input/output devices transferring directly to or from memory are given highest priority and may request bus mastership and steal bus and memory cycles during instruction operations. The processor resumes operation immediately after the memory transfer. Multiple devices can operate simultaneously at maximum direct memory access (DMA) rates by "stealing" bus cycles.

Full 16-bit words or 8-bit bytes of information can be transferred on the bus between a master and a slave. The information can be instructions, addresses, or data. This type of operation occurs when the processor, as master, is fetching instructions, operands, and data from memory, and storing the results into memory after execution of instructions. Direct data transfers occur between a peripheral device control and memory.

2.2 CENTRAL PROCESSOR

The central processor, connected to the UNIBUS as a subsystem, controls the time allocation of the UNIBUS for peripherals and performs arithmetic and logic operations and instruction decoding. It contains multiple high-speed general-purpose registers which can be used as accumulators, address pointers, index registers, and other specialized functions. The processor can perform data transfers directly between I/O devices and memory without disturbing the processor registers; does both single- and double-operand addressing and handles both 16-bit word and 8-bit byte data.

2.2.1 General Registers

The central processor contains 8 general registers which can be used for a variety of purposes.(The PDP-11/55, 11/45 contains 16 general

registers.) The registers can be used as accumulators, index registers, autoincrement registers, autodecrement registers, or as stack pointers for temporary storage of data. Chapter 3 on Addressing describes these uses of the general registers in more detail. Arithmetic operations can be from one general register to another, from one memory or device register to another, or between memory or a device register and a general register. Refer to Figure 2-2.

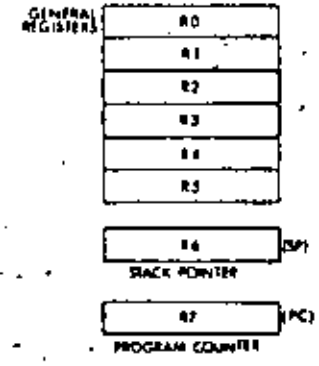


Figure 2-2 The General Registers

R7 is used as the machine's program counter (PC) and contains the address of the next instruction to be executed. It is a general register normally used only for addressing purposes and not as an accumulator for arithmetic operations.

The R6 register is normally used as the Stack Pointer indicating the last entry in the appropriate stack (a common temporary storage area with "Last-In First-Out" characteristics).

2.2.2 Instruction Set

The instruction complement uses the flexibility of the general-purpose registers to provide over 400 powerful hard-wired instructions—the most comprehensive and powerful instruction repertoire of any computer in the 16-bit class. Unlike conventional 16-bit computers, which usually have three classes of instructions (memory reference instructions, operate or AC control instructions and I/O instructions) all operations in the PDP-11 are accomplished with one set of instructions. Since peripheral device registers can be manipulated as flexibly as core memory by the central processor, instructions that are used to manipulate data in core memory may be used equally well for data in peripheral device registers. For example, data in an external device register can be tested or modified directly by the CPU, without bringing it into memory or disturbing the general registers. One can add data directly to a peripheral device register, or compare logically or arithmetically. Thus all PDP-11 instructions can be used to create a new dimension in the treatment of computer I/O and the need for a special class of I/O instructions is eliminated.

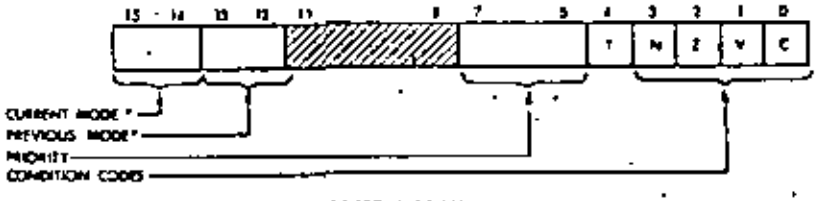
The basic order code of the PDP-11 uses both single and double operand address instructions for words or bytes. The PDP-11 therefore performs

very efficiently in one step, such operations as adding or subtracting two operands, or moving an operand from one location to another.

```
                        PDP-11 Approach
ADD A,B         ;add contents of location A to location B, store results at location B

                        Conventional Approach
LDA A           ;load contents of memory location A into AC
ADD B           ;add contents of memory location B to AC
STA B           ;store result at location B
```

Addressing
Much of the power of the PDP-11 is derived from its wide range of addressing capabilities. PDP-11 addressing modes include sequential addressing forwards or backwards, addressing indexing, indirect addressing, 16-bit word addressing, 8-bit byte addressing, and stack addressing. Variable length instruction formating allows a minimum number of bits to be used for each addressing mode. This results in efficient use of program storage space.

2.2.3 Processor Status Word



* MODE } USED ONLY ON PDP-11/55, & 11/45 WITH MEMORY MANAGEMENT

Figure 2-3 Processor Status Word

The Processor Status word (PS), at location 777776, contains information on the current status of the PDP-11. This information includes the current processor priority; current and previous operational modes; the condition codes describing the results of the last instruction; and an indicator for detecting the execution of an instruction to be trapped during program debugging.

Processor Priority
The Central Processor operates at any one of eight levels of priority, 0-7. When the CPU is operating at level 7 an external device cannot interrupt it with a request for service. The Central Processor must be operating at a lower priority than the external device's request in order for the interruption to take effect. The current priority is maintained in the

processor status word (bits 5-7). The 8 processor levels provide an effective interrupt mask.

Condition Codes
The condition codes contain information on the result of the last CPU operation.

The bits are set as follows:

- Z = 1, if the result was zero
- N = 1, if the result was negative
- C = 1, if the operation resulted in a carry from the MSB
- V = 1, if the operation resulted in an arithmetic overflow

Trap
The trap bit (T) can be set or cleared under program control. When set, a processor trap will occur through location 14 on completion of instruction execution and a new Processor Status Word will be loaded. This bit is especially useful for debugging programs as it provides an efficient method of installing breakpoints.

2.2.4 Stacks

In the PDP-11, a stack is a temporary data storage area which allows a program to make efficient use of frequently accessed data. A program can add or delete words or bytes within the stack. The stack uses the "last-in, first-out" concept; that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order. On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added. The stack is used automatically by program interrupts, subroutine calls, and trap instructions. When the processor is interrupted, the central processor status word and the program counter are saved (pushed) onto the stack area, while the processor services the interrupting device. A new status word is then automatically acquired from an area in core memory which is reserved for interrupt instructions (vector area). A return from the interrupt instruction restores the original processor status and returns to the interrupted program without software intervention.

2.3 MEMORY

Memory Organization

A memory can be viewed as a series of locations, with a number (address) assigned to each location. Thus an 8,192-word PDP-11 memory could be shown as in Figure 2-4.

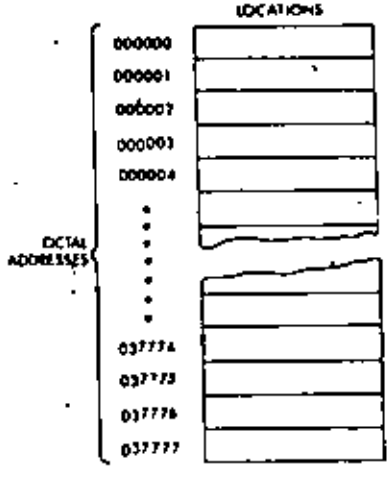


Figure 2-4 Memory Addresses

Because PDP-11 memories are designed to accommodate both 16-bit words and 8-bit bytes, the total number of addresses does not correspond to the number of words. An 8K-word memory can contain 16K bytes and consist of 037777 octal locations. Words always start at even-numbered locations.

A PDP-11 word is divided into a high byte and a low byte as shown in Figure 2-5.

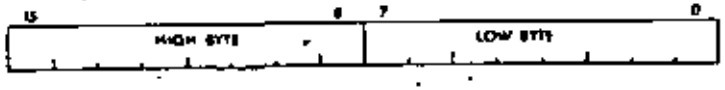


Figure 2-5 High & Low Byte

Low bytes are stored at even-numbered memory locations and high bytes at odd-numbered memory locations. Thus it is convenient to view the PDP-11 memory as shown in Figure 2-6.

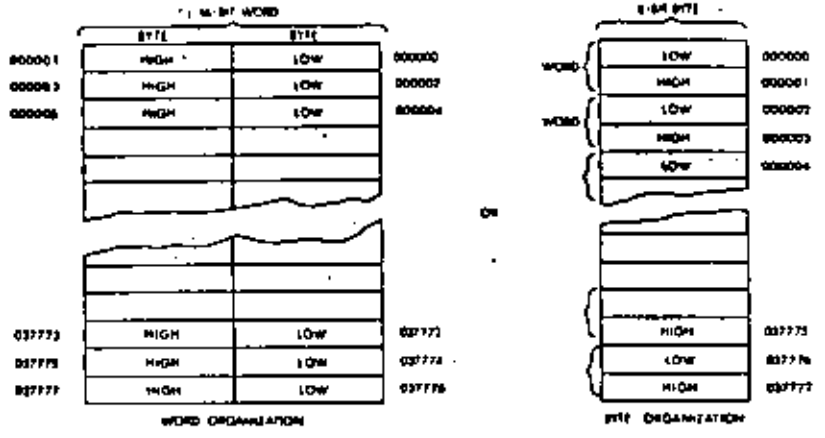


Figure 2-6 Word and Byte Addresses

Certain memory locations have been reserved by the system for interrupt and trap handling, processor stacks, general registers, and peripheral device registers. Addresses from 0 to 370_8 are always reserved and those to 777_8 are reserved on large system configurations for traps and interrupt handling.

A 16-bit word used for byte addressing can address a maximum of 32K words. However, the top 4,096 word locations are reserved for peripheral and register addresses and the user therefore has 28K of core to program. With the PDP-11/55 and 11/45, the user can expand above 28K with the Memory Management. This device provides an 18-bit effective memory address which permits addressing up to 124K words of actual memory.

If the Memory Management option is not used, an octal address between 160 000 and 177 777 is interpreted as 760 000 to 777 777. That is, if bit 15, 14 and 13 are 1's, then bits 17 and 16 (the extended address bits) are considered to be 1's, which relocates the last 4K words (8K bytes) to become the highest locations accessed by the UNIBUS.

2.4 AUTOMATIC PRIORITY INTERRUPTS

The multi-level automatic priority interrupt system permits the processor to respond automatically to conditions outside the system. Any number of separate devices can be attached to each level.

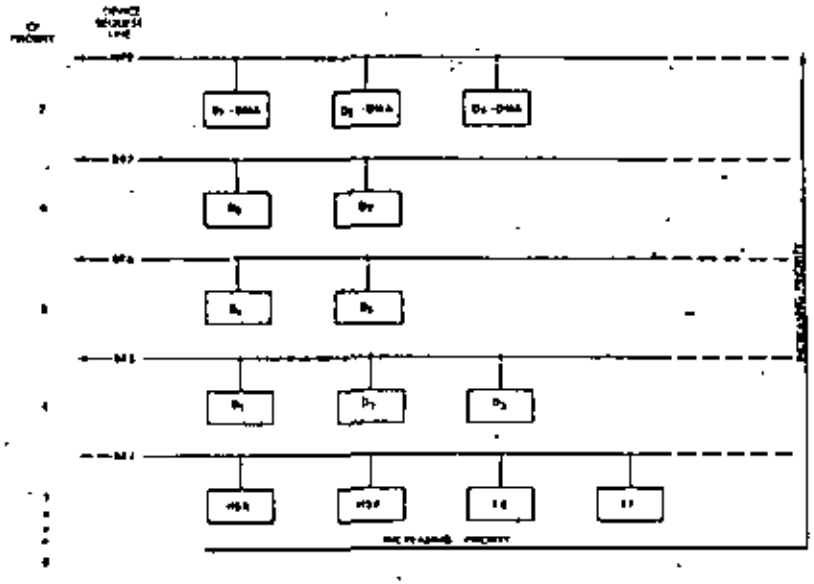


Figure 2-7 UNIBUS Priority

Each peripheral device in the PDP-11 system has a pointer to its own pair of memory words (one points to the devices's service routine, and the other contains the new processor status information). This unique identification eliminates the need for polling of devices to identify an interrupt, since the interrupt service hardware selects and begins executing the appropriate service routine after having automatically saved the status of the interrupted program segment.

The devices' interrupt priority and service routine priority are independent. This allows adjustment of system behavior in response to real-time conditions, by dynamically changing the priority level of the service routine.

The interrupt system allows the processor to continually compare its own programmable priority with the priority of any interrupting devices and to acknowledge the device with the highest level above the processor's priority level. The servicing of an interrupt for a device can be interrupted in order to service an interrupt of a higher priority. Service to the lower priority device is resumed automatically upon completion of the higher level servicing. Such a process, called nested interrupt servicing, can be carried out to any level without requiring the software to save and restore processor status at each level.

When a device (other than the central processor) is capable of becoming bus master and requests use of the bus, it is generally for one of two purposes:

1. To make a non-processor transfer of data directly to or from memory

2. To interrupt a program execution and force the processor to go to a specific address where an interrupt service routine is located.

Direct Memory Access

All PDP-11's provide for direct access to memory. Any number of DMA devices may be attached to the UNIBUS. Maximum priority is given to DMA devices, thus allowing memory data storage or retrieval at memory cycle speeds. Response time is minimized by the organization and logic of the UNIBUS, which samples requests and priorities in parallel with data transfers.

Direct memory or direct data transfers can be accomplished between any two peripherals without processor supervision. These non-processor request transfers, called NPR level data transfers, are usually made for Direct Memory Access (memory to/from mass storage) or direct device transfers (disk refreshing a CRT display).

Bus Requests

Bus requests from external devices can be made on one of five request lines. Highest priority is assigned to non-processor request (NPR). These are direct memory access type transfers, and are honored by the processor between bus cycles of an instruction execution.

The processor's priority can be set under program control to one of eight levels using bits 7, 6, and 5 in the processor status register. These bits set a priority level that inhibits granting of bus requests on lower levels or on the same level. When the processor's priority is set to a level, for example PS6, all bus requests on BR6 and below are ignored.

When more than one device is connected to the same bus request (BR) line, a device nearer the central processor has a higher priority than a device farther away. Any number of devices can be connected to a given BR or NPR line.

Thus the priority system is two-dimensional and provides each device with a unique priority. Each device may be dynamically, selectively enabled or disabled under program control.

Once a device other than the processor has control of the bus, it may do one of two types of operations: data transfers or interrupt operations.

NPR Data Transfers

NPR data transfers can be made between any two peripheral devices without the supervision of the processor. Normally, NPR transfers are between a mass storage device, such as a disk, and core memory. The structure of the bus also permits device-to-device transfers, allowing customer-designed peripheral controllers to access other devices, such as disks, directly.

An NPR device has very fast access to the bus and can transfer at high data rates once it has control. The processor state is not affected by the transfer; therefore the processor can relinquish control while an instruction is in progress. This can occur at the end of any bus cycles

except in between a read-modify-write sequence. An NPR device in control of the bus may transfer 16-bit words from memory at memory speed.

BR Transfers

Devices that gain bus control with one of the Bus Request lines (BR 7-BR4) can take full advantage of the Central Processor by requesting an interrupt. In this way, the entire instruction set is available for manipulating data and status registers.

When a service routine is to be run, the current task being performed by the central processor is interrupted, and the device service routine is initiated. Once the request has been satisfied, the Processor returns to its former task.

Interrupt Procedure

Interrupt handling is automatic in the PDP-11. No device polling is required to determine which service routine to execute. The operations required to service an interrupt are as follows:

1. Processor relinquishes control of the bus, priorities permitting.
2. When a master gains control, it sends the processor an interrupt command and an unique memory address which contains the address of the device's service routine, called the interrupt vector address. Immediately following this pointer address is a word (located at vector address +2) which is to be used as a new Processor Status Word.
3. The processor stores the current Processor Status (PS) and the current Program Counter (PC) into CPU temporary registers.
4. The new PC and PS (interrupt vector) are taken from the specified address. The old PS and PC are then pushed onto the current stack. The service routine is then initiated.
5. The device service routine can cause the processor to resume the interrupted process by executing the Return from Interrupt Instruction, described in Chapter 4, which pops the two top words from the current processor stack and uses them to load the PC and PS registers.

A device routine can be interrupted by a higher priority bus request any time after the new PC and PS have been loaded. If such an interrupt occurs, the PC and PS of the service routine are automatically stored in the temporary registers and then pushed onto the new current stack, and the new device routine is initiated.

Interrupt Servicing

Every hardware device capable of interrupting the processor has a unique set of locations (2 words) reserved for its interrupt vector. The first word contains the location of the device's service routine, and the second, the Processor Status Word that is to be used by the service routine. Through

proper use of the PS, the programmer can switch the operational mode of the processor, and modify the Processor's Priority level to mask out lower level interrupts.

Reentrant Code

Both the interrupt handling hardware and the subroutine call hardware facilitate writing reentrant code for the PDP-11. This type of code allows a single copy of a given subroutine or program to be shared by more than one process or task. This reduces the amount of core needed for multi-task applications such as the concurrent servicing of many peripheral devices.

Power Fail and Restart

Whenever AC power drops below 95 volts for 110v power (190 volts for 220v) or outside a limit of 47 to 63 Hz, as measured by DC power, the power fail sequence is initiated. The Central Processor automatically traps to location 24 and the power fail program has 2 msec. to save all volatile information (data in registers), and to condition peripherals for power fail.

When power is restored the processor traps to location 24 and executes the power up routine to restore the machine to its state prior to power failure.

CHAPTER 8

PDP-11/34 MEMORY MANAGEMENT

8.1 GENERAL

8.1.1 Memory Management

This chapter describes the Memory Management unit of the 11/34 Central Processor. The PDP-11/34 provides the hardware facilities necessary for complete memory management and protection. It is designed to be a memory management facility for systems where the memory size is greater than 28K words and for multi-user, multi-programming systems where protection and relocation facilities are necessary.

8.1.2 Programming

The Memory Management hardware has been optimized towards a multi-programming environment and the processor can operate in two modes, Kernel and User. When in Kernel mode, the program has complete control and can execute all instructions. Monitors and supervisory programs would be executed in this mode.

When in User Mode, the program is prevented from executing certain instructions that could:

a) cause the modification of the Kernel program.
b) halt the computer.
c) use memory space assigned to the Kernel or other users.

In a multi-programming environment several user programs would be resident in memory at any given time. The task of the supervisory program would be: control the execution of the various user programs, manage the allocation of memory and peripheral device resources, and safeguard the integrity of the system as a whole by careful control of each user program.

In a multi-programming system, the Management Unit provides the means for assigning pages (relocatable memory segments) to a user program and preventing that user from making any unauthorized access to those pages outside his assigned area. Thus, a user can effectively be prevented from accidental or willful destruction of any other user program or the system executive program.

Hardware implemented features enable the operating system to dynamically allocate memory upon demand while a program is being run. These features are particularly useful when running higher-level language programs, where, for example, arrays are constructed at execution time. No fixed space is reserved for them by the compiler. Lacking dynamic memory allocation capability, the program would have to calculate and allow sufficient memory space to accommodate the worst case. Memory Management eliminates this time-consuming and wasteful procedure.

8.1.3 Basic Addressing

The addresses generated by all PDP-11 Family Central Processor Units (CPUs) are 18-bit direct byte addresses. Although the PDP-11 Family word length is 16 bits, the UNIBUS and CPU addressing logic actually is 18 bits. Thus, while the PDP-11 word can only contain address references up to 32K words (64K bytes) the CPU and UNIBUS can reference addresses up to 128K words (256K bytes). These extra two bits of addressing logic provide the basic framework for expanding memory references.

In addition to the word length constraint on basic memory addressing space, the uppermost 4K words of address space is always reserved for UNIBUS I/O device registers. In a basic PDP-11 memory configuration (without Management) all address references to the uppermost 4K words of 16-bit address space (160000-177777) are converted to full 18-bit references with bits 17 and 16 always set to 1. Thus, a 16-bit reference to the I/O device register at address 173224 is automatically internally converted to a full 18-bit reference to the register at address 773224. Accordingly, the basic PDP-11 configuration can directly address up to 28K words of true memory, and 4K words of UNIBUS I/O device registers.

8.1.4 Active Page Registers

The Memory Management Unit uses two sets of eight 32-bit Active Page Registers. An APR is actually a pair of 16-bit registers: a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information needed to describe and relocate the currently active memory pages.

One set of APR's is used in Kernel mode, and the other in User mode. The choice of which set to be used is determined by the current CPU mode contained in the Processor Status word.

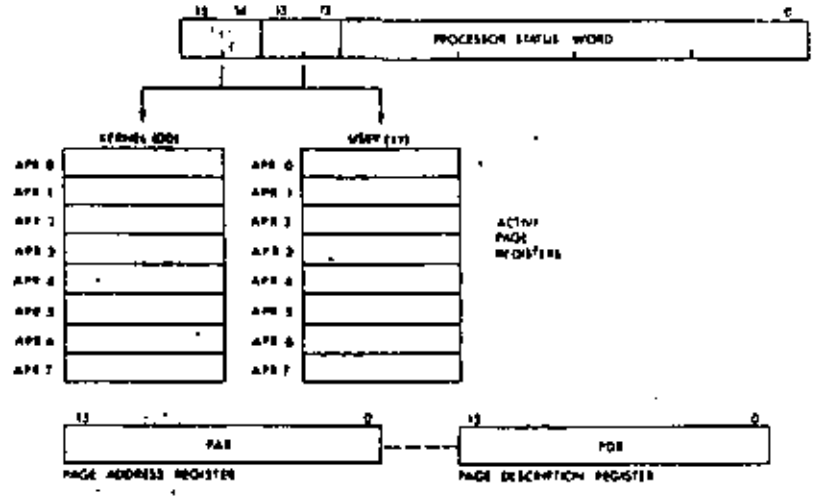


Figure 8-1 Active Page Registers

8.1.5 Capabilities Provided by Memory Management

Memory Size (words):	124K, max (plus 4K for I/O & registers)
Address Space:	Virtual (16 bits) Physical (18 bits)
Modes of Operation:	Kernel & User
Stack Pointers:	2 (one for each mode)
Memory Relocation:	
Number of Pages:	16 (8 for each mode)
Page Length:	32 to 4,096 words
Memory Protection:	no access read only read/write

8.2 RELOCATION

8.2.1 Virtual Addressing

When the Memory Management Unit is operating, the normal 16-bit direct byte address is no longer interpreted as a direct Physical Address (PA) but as a Virtual Address (VA) containing information to be used in constructing a new 18-bit physical address. The information contained in the Virtual Address (VA) is combined with relocation and description information contained in the Active Page Register (APR) to yield an 18-bit Physical Address (PA).

Because addresses are automatically relocated, the computer may be considered to be operating in virtual address space. This means that no matter where a program is loaded into physical memory, it will not have

to be "re-linked"; it always appears to be at the same virtual location in memory.

The virtual address space is divided into eight 4K-word pages. Each page is relocated separately. This is a useful feature in multi-programmed timesharing systems. It permits a new large program to be loaded into discontinuous blocks of physical memory.

A page may be as small as 32 words, so that short procedures or data areas need occupy only as much memory as required. This is a useful feature in real-time control systems that contain many separate small tasks. It is also a useful feature for stack and buffer control.

A basic function is to perform memory relocation and provide extended memory addressing capability for systems with more than 28K of physical memory. Two sets of page address registers are used to relocate virtual addresses to physical addresses in memory. These sets are used as hardware relocation registers that permit several user's programs, each starting at virtual address 0, to reside simultaneously in physical memory.

8.2.2 Program Relocation

The page address registers are used to determine the starting address of each relocated program in physical memory. Figure 8-2 shows a simplified example of the relocation concept.

Program A starting address 0 is relocated by a constant to provide physical address 6400_8.

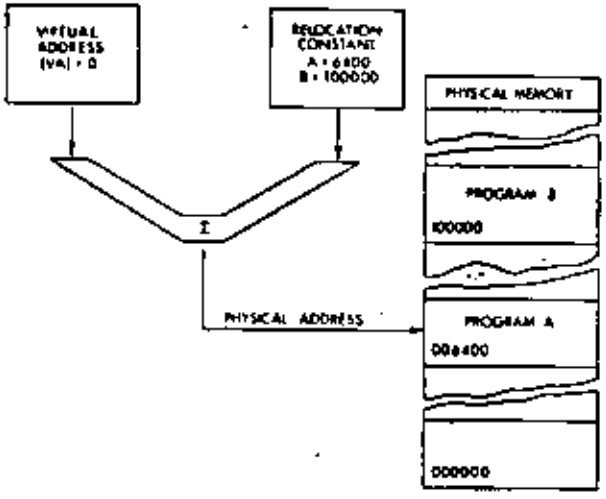


Figure 8-2 Simplified Memory Relocation Concept

If the next processor virtual address is 2, the relocation constant will then cause physical address 6402_8, which is the second item of Program A, to be accessed. When Program B is running, the relocation constant is changed to 100000_8. Then, Program B virtual addresses starting at 0, are relocated to access physical addresses starting at 100000_8. Using the active page address registers to provide relocation eliminates the need to "re-link" a program each time it is loaded into a different physical memory location. The program always appears to start at the same address.

A program is relocated in pages consisting of from 1 to 128 blocks. Each block is 32 words in length. Thus, the maximum length of a page is 4096 (128 x 32) words. Using all of the eight available active page registers in a set, a maximum program length of 32,768 words can be accommodated. Each of the eight pages can be relocated anywhere in the physical memory, as long as each relocated page begins on a boundary that is a multiple of 32 words. However, for pages that are smaller than 4K words, only the memory actually allocated to the page may be accessed.

The relocation example shown in Figure 8-3 illustrates several points about memory relocation.

a) Although the program appears to be in contiguous address space to the processor, the 32K-word physical address space is actually scattered through several separate areas of physical memory. As long as the total available physical memory space is adequate, a program can be loaded. The physical memory space need not be contiguous.

b) Pages may be relocated to higher or lower physical addresses, with respect to their virtual address ranges. In the example Figure 8-3, page 1 is relocated to a higher range of physical addresses, page 4 is relocated to a lower range, and page 3 is not relocated at all (even though its relocation constant is non-zero).

c) All of the pages shown in the example start on 32-word boundaries.

d) Each page is relocated independently. There is no reason why two or more pages could not be relocated to the same physical memory space. Using more than one page address register in the set to access the same space would be one way of providing different memory access rights to the same data, depending upon which part of a program was referencing that data.

Memory Units

Block:	32 words
Page:	1 to 128 blocks (32 to 4,096 words)
No. of pages:	8 per mode
Size of relocatable memory:	27,768 words, max (8 x 4,096)

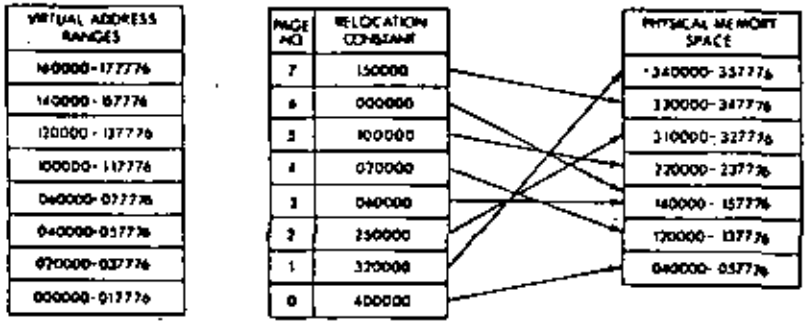


Figure 8-3 Relocation of a 32K Word Program into 124K Word Physical Memory

8.3 PROTECTION

A timesharing system performs multiprogramming: it allows several programs to reside in memory simultaneously, and to operate sequentially. Access to these programs, and the memory space they occupy, must be strictly defined and controlled. Several types of memory protection must be afforded a timesharing system. For example:

a) User programs must not be allowed to expand beyond allocated space, unless authorized by the system.

b) Users must be prevented from modifying common subroutines and algorithms that are resident for all users.

c) Users must be prevented from gaining control of or modifying the operating system software.

The Memory Management option provides the hardware facilities to implement all of the above types of memory protection.

8.3.1 Inaccessible Memory

Each page has a 2-bit access control key associated with it. The key is assigned under program control. When the key is set to 0, the page is defined as non-resident. Any attempt by a user program to access a non-resident page is prevented by an immediate abort. Using this feature to provide memory protection, only those pages asociated with the current program are set to legal access keys. The access control keys of all other program pages are set to 0, which prevents illegal memory references.

8.3.2 Read-Only Memory

The access control key for a page can be set to 2, which allows read (fetch) memory references to the page, but immediately aborts any attempt to write into that page. This read-only type of memory protection

can be afforded to pages that contain common data, subroutines, or shared algorithms. This type of memory protection allows the access rights to a given information module to be user-dependent. That is, the access right to a given information module may be varied for different users by altering the access control key.

A page address register in each of the sets (Kernel and User modes) may be set up to reference the same physical page in memory and each may be keyed for different access rights. For example, the User access control key might be 2 (read-only access), and the Kernel access control key might be 6 (allowing complete read/write access).

8.3.3 Multiple Address Space

There are two complete separate PAR/PDR sets provided: one set for Kernel mode and one set for User mode. This affords the timesharing system with another type of memory protection capability. The mode of operation is specified by the Processor Status Word current mode field, or previous mode field, as determined by the current instruction.

Assuming the current mode PS bits are valid, the active page register sets are enabled as follows:

PS(bits15, 14)	PAR/PDR Set Enabled
00	Kernel mode
01	Illegal (all references aborted on access)
10	
11	User mode

Thus, a User mode program is relocated by its own PAR/PDR set, as are Kernel programs. This makes it impossible for a program running in one mode to accidentally reference space allocated to another mode when the active page registers are set correctly. For example, a user cannot transfer to Kernel space. The Kernel mode address space may be reserved for resident system monitor functions, such as the basic Input/Output Control routines, memory management trap handlers, and timesharing scheduling modules. By dividing the types of timesharing system programs functionally between the Kernel and User modes, a minimum amount of space control housekeeping is required as the timeshared operating system sequences from one user program to the next. For example, only the User PAR/PDR set needs to be updated as each new user program is serviced. The two PAR/PDR sets implemented in the Memory Management Unit are shown in Figure 8-1.

8.4 ACTIVE PAGE REGISTERS

The Memory Management Unit provides two sets of eight Active Page Registers (APR). Each APR consists of a Page Address Register (PAR) and a Page Descriptor Register (PDR). These registers are always used as a pair and contain all the information required to locate and describe the current active pages for each mode of operation. One PAR/PDR set is used in Kernel mode and the other is used in User mode. The current mode bits (or in some cases, the previous mode bits) of the Processor Status Word determine which set will be referenced for each memory access. A program operating in one mode cannot use the PAR/PDR sets of the other mode to access memory. Thus, the two sets are

a key feature in providing a fully protected environment for a time-shared multi-programming system.

A specific processor I/O address is assigned to each PAR and PDR of each set. Table 7-1 is a complete list of address assignment.

NOTE
UNIBUS devices cannot access PARs or PDRs

In a fully-protected multi-programming environment, the implication is that only a program operating in the Kernel mode would be allowed to write into the PAR and PDR locations for the purpose of mapping user's programs. However, there are no restraints imposed by the logic that will prevent User mode programs from writing into these registers. The option of implementing such a feature in the operating system, and thus explicitly protecting these locations from user's programs, is available to the system software designer.

Table 8-1 PAR/PDR Address Assignments

Kernel Active Page Registers			User Active Page Registers		
No.	PAR	PDR	No.	PAR	PDR
0	772340	772300	0	777640	777600
1	772342	772302	1	777642	777602
2	772344	772304	2	777644	777604
3	772346	772306	3	777646	777606
4	772350	772310	4	777650	777610
5	772352	772312	5	777652	777612
6	772354	772314	6	777654	777614
7	772356	772316	7	777656	777616

8.4.1 Page Address Registers (PAR)

The Page Address Register (PAR), shown in Figure 8-4, contains the 12-bit Page Address Field (PAF) that specifies the base address of the page.

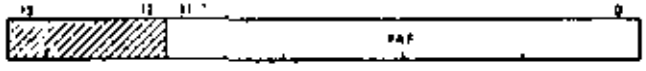


Figure 8-4 Page Address Register

Bits 15-12 are unused and reserved for possible future use.

The Page Address Register may be alternatively thought of as a relocation constant, or as a base register containing a base address. Either interpretation indicates the basic function of the Page Address Register (PAR) in the relocation scheme.

8.4.2 Page Descriptor Registers (PDR)

The Page Descriptor Register (PDR), shown in Figure 8-5, contains information relative to page expansion, page length, and access control.

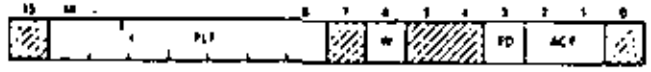


Figure 8-5 Page Descriptor Register

Access Control Field (ACF)

This 2-bit field, bits 2 and 1, of the PDR describes the access rights to this particular page. The access codes or "keys" specify the manner in which a page may be accessed and whether or not a given access should result in an abort of the current operation. A memory reference that causes an abort is not completed and is terminated immediately.

Aborts are caused by attempts to access non-resident pages, page length errors, or access violations, such as attempting to write into a read-only page. Traps are used as an aid in gathering memory management information.

In the context of access control, the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. A "write" is synonymous with what is usually called a "store" or "modify" in many computer systems. Table 8-2 lists the ACF keys and their functions. The ACF is written into the PDR under program control.

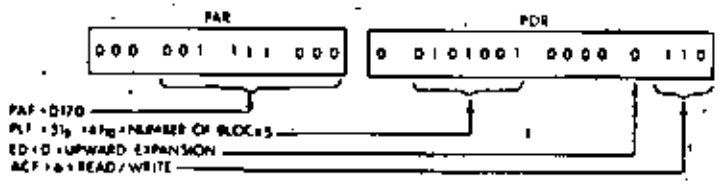
Table 8-2 Access Control Field Keys

AFC	Key	Description	Function
00	0	Non-resident	Abort any attempt to access this non-resident page
01	2	Resident read-only	Abort any attempt to write into this page.
10	4	(unused)	Abort all Accesses.
11	6	Resident read/ write	Read or Write allowed, No trap or abort occurs.

Expansion Direction (ED)

The ED bit located in PDR bit position 3 indicates the authorized direction in which the page can expand. A logic 0 in this bit (ED = 0) indicates the page can expand upward from relative zero. A logic 1 in this bit (ED = 1) indicates the page can expand downward toward relative zero. The ED bit is written into the PDR under program control. When the expansion direction is upward (ED = 0), the page length is increased by adding blocks with higher relative addresses. Upward expansion is usually specified for program or data pages to add more program or table space. An example of page expansion upward is shown in Figure 8-6.

When the expansion direction is downward (ED = 1), the page length is increased by adding blocks with lower relative addresses. Downward expansion is specified for stack pages so that more stack space can be added. An example of page expansion downward is shown in Figure 8-7.



NOTE:
To specify a block length of 42 for an upward expandable page, write highest authorized block no. directly into high byte of PDR. Bit 15 is not used because the highest allowable block number is 177_8.

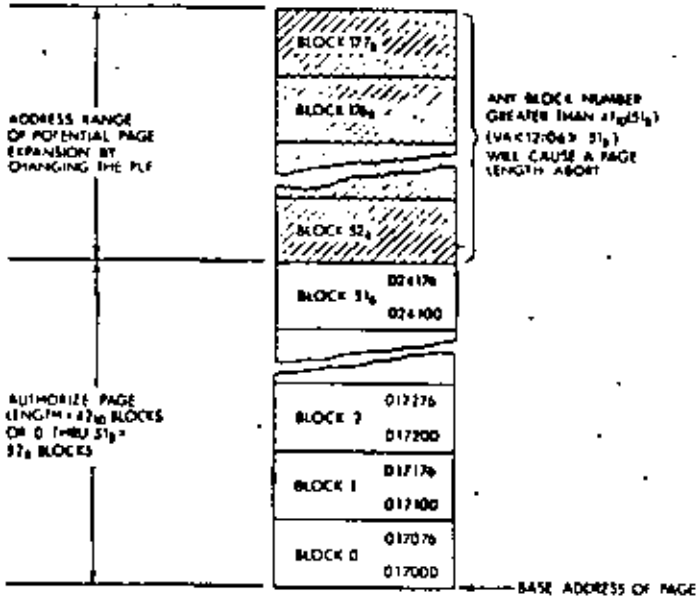


Figure 8-6 Example of an Upward Expandable Page

Written Into (W).
The W bit located in PDR bit position 6 indicates whether the page has been written into since it was loaded into memory. W = 1 is affirmative. The W bit is automatically cleared when the PAR or PDR of that page is written into. It can only be set by the control logic.

In disk swapping and memory overlay applications, the W bit (bit 6) can be used to determine which pages in memory have been modified by a user. Those that have been written into must be saved in their current form. Those that have not been written into (W = 0), need not be saved and can be overlayed with new pages, if necessary.

Page Length Field (PLF)
The 7-bit PLF located in PDR (bits 14-8) specifies the authorized length of the page, in 32-word blocks. The PLF holds block numbers from 0 to 177_8; thus allowing any page length from 1 to 128_{10} blocks. The PLF is written in the PDR under program control.

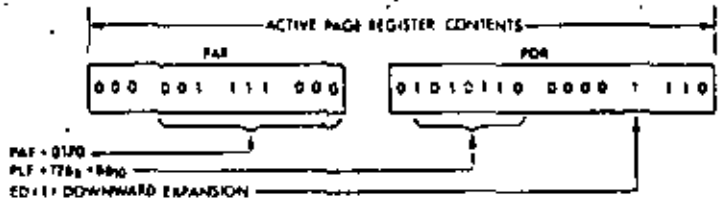
PLF for an Upward Expandable Page
When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}) (Figure 8-6). The hardware compares the virtual address block number, VA (bits 12-6) with the PLF to determine if the virtual address is within the authorized page length.

When the virtual address block number is less than or equal to the PLF, the virtual address is within the authorized page length. If the virtual address is greater than the PLF, a page length fault (address too high) is detected by the hardware and an abort occurs. In this case, the virtual address space legal to the program is non-contiguous because the three most significant bits of the virtual address are used to select the PAR/PDR set.

PLF for a Downward Expandable Page
The capability of providing downward expansion for a page is intended specifically for those pages that are to be used as stacks. In the PDP-11, a stack starts at the highest location reserved for it and expands downward toward the lowest address as items are added to the stack.

When the page is to be downward expandable, the PLF must be set to authorize a page length, in blocks, that starts at the highest address of the page. That is always Block 177_8. Refer to Figure 8-7, which shows an example of a downward expandable page. A page length of 42_{10} blocks is arbitrarily chosen so that the example can be compared with the upward expandable example shown in Figure 8-6.

NOTE
The same PAF is used in both examples. This is done to emphasize that the PAF, as the base address, always determines the lowest address of the page, whether it is upward or downward expandable.



To specify page length for a downward expandable page, write complement of blocks required into high byte of PDR.

In this example, a 42-block page is required.
PLF is derived as follows:

$42_{10} = 52_8$; two's complement = 126_8.

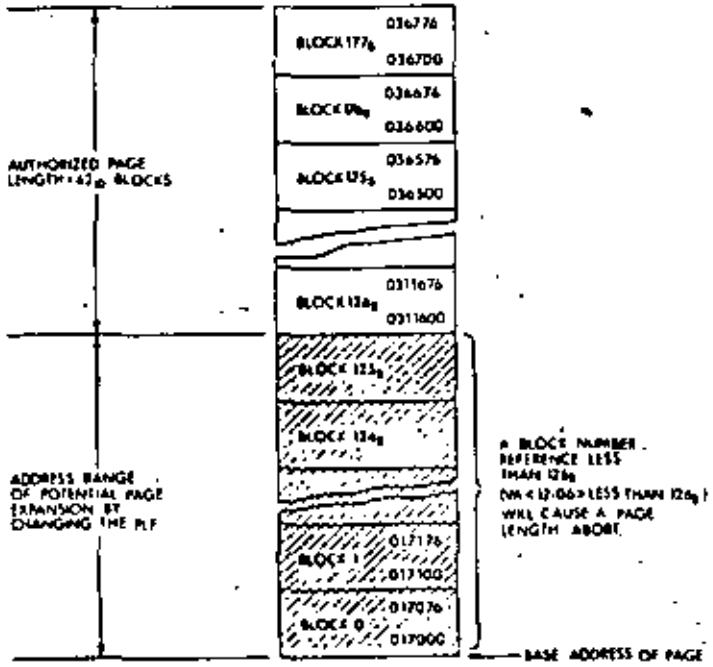


Figure 8-7 Example of a Downward Expandable Page

The calculations for complementing the number of blocks required to obtain the PLF is as follows:

MAXIMUM BLOCK NO.	MINUS	REQUIRED LENGTH	EQUALS	PLF
177_8	—	52_8	=	125_8
127_{10}	—	42_{10}	=	85_{10}

8.5 VIRTUAL & PHYSICAL ADDRESSES

The Memory Management Unit is located between the Central Processor Unit and the UNIBUS address lines. When Memory Management is enabled, the Processor ceases to supply address information to the Unibus. Instead, addresses are sent to the Memory Management Unit where they are relocated by various constants computed within the Memory Management Unit.

8.5.1 Construction of a Physical Address

The basic information needed for the construction of a Physical Address (PA) comes from the Virtual Address (VA), which is illustrated in Figure 8-8, and the appropriate APR set.

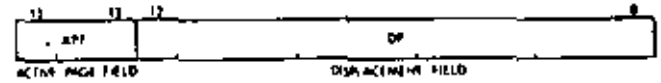


Figure 8-8 Interpretation of a Virtual Address

The Virtual Address (VA) consists of:

1. The Active Page Field (APF). This 3-bit field determines which of eight Active Page Registers (APR0-APR7) will be used to form the Physical Address (PA).

2. The Displacement Field (DF). This 13-bit field contains an address relative to the beginning of a page. This permits page lengths up to 4K words (2^{13} = 8K bytes). The DF is further subdivided into two fields as shown in Figure 8-9.

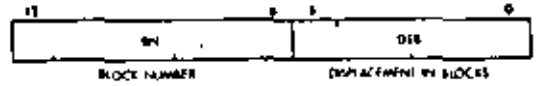


Figure 8-9 Displacement Field of Virtual Address

The Displacement Field (DF) consists of:

1. The Block Number (BN). This 7-bit field is interpreted as the block number within the current page.

2. The Displacement in Block (DIB). This 6-bit field contains the displacement within the block referred to by the Block Number.

The remainder of the information needed to construct the Physical Address comes from the 12-bit Page Address Field (PAF) (part of the Active Page Register) and specifies the starting address of the memory which that APR describes. The PAF is actually a block number in the physical memory, e.g. PAF = 3 indicates a starting address of 96, (3 X 32 = 96) words in physical memory.

The formation of the Physical Address is illustrated in Figure 8-10.

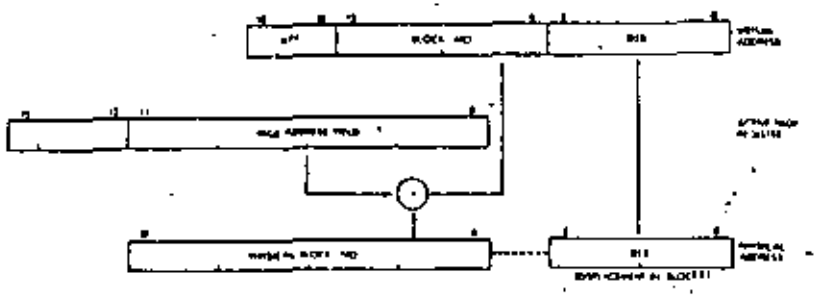


Figure 8-10 Construction of a Physical Address

The logical sequence involved in constructing a Physical Address is as follows:

1. Select a set of Active Page Registers depending on current mode.
2. The Active Page Field of the Virtual Address is used to select an Active Page Register (APR0-APR7).
3. The Page Address Field of the selected Active Page Register contains the starting address of the currently active page as a block number in physical memory.
4. The Block Number from the Virtual Address is added to the block number from the Page Address Field to yield the number of the block in physical memory which will contain the Physical Address being constructed.
5. The Displacement in Block from the Displacement Field of the Virtual Address is joined to the Physical Block Number to yield a true 18-bit Physical Address.

8.5.2 Determining the Program Physical Address

A 16-bit virtual address can specify up to 32K words, in the range from 0 to 177776_8 (word boundaries are even octal numbers). The three most significant virtual address bits designate the PAR/PDR set to be referenced during page address relocation. Table 8-3 lists the virtual address ranges that specify each of the PAR/PDR sets.

Table 8-3 Relating Virtual Address to PAR/PDR Set

Virtual Address Range	PAR/PDR Set
000000-17776	0
020000-37776	1
040000-57776	2
060000-77776	3
100000-117776	4
120000-137776	5
140000-157776	6
160000-177776	7

NOTE

Any use of page lengths less than 4K words causes holes to be left in the virtual address space.

8.6 STATUS REGISTERS

Aborts generated by the protection hardware are vectored through Kernel virtual location 250. Status Registers #0 and #2 are used to determine why the abort occurred. Note that an abort to a location which is itself an invalid address will cause another abort. Thus the Kernel program must insure that Kernel Virtual Address 250 is mapped into a valid address, otherwise a loop will occur which will require console intervention.

8.6.1 Status Register 0 (SR0)

SR0 contains abort error flags, memory management enable, plus other essential information required by an operating system to recover from an abort or service a memory management trap. The SR0 format is shown in Figure 8-11. Its address is 777 572.

Figure 8-11 Format of Status Register #0 (SR0)

Bits 15-13 are the abort flags. They may be considered to be in a "priority queue" in that "flags to the right" are less significant and should be ignored. For example, a "non-resident" abort service routine would ignore page length and access control flags. A "page length" abort service routine would ignore an access control fault.

NOTE

Bit 15, 14, or 13, when set (abort conditions) cause the logic to freeze the contents of SR0 bits 1 to 6 and status register SR2. This is done to facilitate recovery from the abort.

Protection is enabled when an address is being relocated. This implies that either SR0, bit 0 is equal to 1 (Memory Management enabled) or that SR0, bit 8, is equal to 1 and the memory reference is the final one of a destination calculation (maintenance/destination mode).

Note that SR0 bits 0 and 8 can be set under program control to provide meaningful memory management control information. However, information written into all other bits is not meaningful. Only that information which is automatically written into these remaining bits as a result of hardware actions is useful as a monitor of the status of the memory management unit. Setting bits 15-13 under program control will not cause traps to occur. These bits, however, must be reset to 0 after an abort or trap has occurred in order to resume monitoring memory management.

Abort-Nonresident
Bit 15 is the "Abort-Nonresident" bit. It is set by attempting to access a page with an access control field (ACF) key equal to 0 or 4 or by enabling relocation with an illegal mode in the PS.

Abort—Page Length
Bit 14 is the "Abort-Page Length" bit. It is set by attempting to access a location in a page with a block number (virtual address bits 12-6) that is outside the area authorized by the Page Length Field (PFL) of the PDR for that page.

Abort-Read Only
Bit 13 is the "Abort-Read Only" bit. It is set by attempting to write in a "Read-Only" page having an access key of 2.

NOTE

There are no restrictions that any abort bits could not be set simultaneously by the same access attempt.

Maintenance/Destination Mode
Bit 8 specifies maintenance use of the Memory Management Unit. It is used for diagnostic purposes. For the instructions used in the initial diagnostic program, bit 8 is set so that only the final destination reference is relocated. It is useful to prove the capability of relocating addresses.

Mode of Operation
Bits 5 and 6 indicate the CPU mode (User or Kernel) associated with the page causing the abort. (Kernel = 00, User = 11).

Page Number
Bits 3-1 contain the page number of reference. Pages, like blocks, are numbered from 0 upwards. The page number bit is used by the error recovery routine to identify the page being accessed if an abort occurs.

Enable Relocation and Protection
Bit 0 is the "Enable" bit. When it is set to 1, all addresses are relocated

and protected by the memory management unit. When bit 0 is set to 0, the memory management unit is disabled and addresses are neither relocated nor protected.

8.6.2 Status Register 2 (SR2)

SR2 is loaded with the 16-bit Virtual Address (VA) at the beginning of each instruction fetch but is not updated if the instruction fetch fails. SR2 is read only; a write attempt will not modify its contents. SR2 is the Virtual Address Program Counter. Upon an abort, the result of SR0 bits 15, 14, or 13 being set, will freeze SR2 until the SR0 abort flags are cleared. The address of SR2 is 777 576.

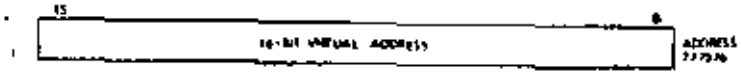


Figure 8-12 Format of Status Register 2 (SR2))

8.7 INSTRUCTIONS

Memory Management provides the ability to communicate between two spaces, as determined by the current and previous modes of the Processor Status word (PS).

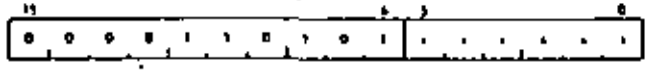
Mnemonic	Instruction	Op Code
MFPI	move from previous instruction space	0065SS
MTPI	move to previous instruction space	0066DD
MFPD	move from previous data space	1065SS
MTPD	move to previous data space	1066DD

These instructions are directly compatible with the larger 11 computers.

The PDP-11/45 Memory Management unit, the KT11-C, implements a separate instruction and data address space. In the PDP-11/34, there is no differentiation between instruction or data space. The 2 instructions MFPD and MTPD (Move to and from previous data space) execute identically to MFPI and MTPI.

MFPD
MFPI

move from previous data space	1065SS
move from previous instruction space	0065SS



Operation: (temp)←(src)
↓(SP)←(temp)

Condition Codes:
N: set if the source <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction pushes a word onto the current stack from an address in previous space, Processor Status (bits 13, 12). The source address is computed using the current registers and memory map.

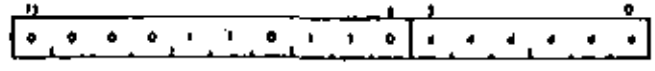
Example:

```
MFPI @ (R2)      R2 = 1000
                 1000 = 37526
```

The execution of this instruction causes the contents of (relative) 37526 of the previous address space to be pushed onto the current stack as determined by the PS (bits 15, 14).

MTPD
MTPI

move to previous data space	1066DD
move to previous instruction space	0066DD



Operation: (temp)←(SP)↑
(dst)←(temp)

Condition Codes:
N: set if the sourse <0; otherwise cleared
Z: set if the source =0; otherwise cleared
V: cleared
C: unaffected

Description: This instruction pops a word off the current stack determined by PS (bits 15, 14) and stores that word into an address in previous space PS (bits 13, 12). The destination address is computed using the current registers and memory map. An example is as follows:

Example:

```
MTPI @ (R2)      R2 = 1000
                 1000 = 37526
```

The execution of this instruction causes the top word of the current stack to get stored into the (relative) 37526 of the previous address space.

MTPI AND MFPI, MODE 0, REGISTER 6 ARE UNIQUE IN THAT THESE INSTRUCTIONS ENABLE COMMUNICATIONS TO AND FROM THE PREVIOUS USER STACK.

; MFPI, MODE 0, NOT REGISTER 6

```
MOV   #KM+PUM, PSW      ; KMODE, PREV USER
MOV   #-1, -2(6)        ; MOVE -1 on kernel stack -2
CLR   %0
INC   @#SR0             ; ENABLE MEM MGT
MFPI  %0                ; -(KSP)←R0 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of R0 which is 0.

; MFPI, MODE 0, REGISTER 6

```
MOV   #UM+PUM, PSW
CLR   %6                ; SET R16=0
MOV   #KM+PUM, PSW      ; K MODE, PREV USER
MOV   #-1, -2 (6)
INC   @#SR0             ; ENABLE MEM MGT
MFPI  %6                ; -(KSP)←R16 CONTENTS
```

The -1 in the kernel stack is now replaced by the contents of R16 (user stack pointer which is 0).

To obtain info from the user stack if the status is set to kernel mode, prev user, two steps are needed.

```
MFPI  %6          ; get contents of R16=user pointer
MFPI  @(5)+       ; get user pointer from kernel stack
                  ; use address obtained to get data
                  ; from user mode using the prev
                  ; mode
```

The desired data from the user stack is now in the kernel stack and has replaced the user stack address.

; MTPI, MODE 0 , NOT REGISTER 6

```
      MOV   #KM+PUM, PSW    ; KERNEL MODE, PREV USES
      MOV   #TAGX, (6)      ; PUT NEW PC ON STACK
      INC   @#SR0           ; ENABLE KT
      MTPI  %7              ; %7← (6)+
      HLT                   ; ERROR
TAGX: CLR   @#SR0           ; DISABLE MEM MGT
```

The new PC is popped off the current stack and since this is mode 0 and not register 6 the destination is register 7.

; MTPI, MODE 0, REGISTER 6

```
MOV   #UM+PUM, PSW    ; user mode, Prev User
CLR   %6              ; set user SP=0 (R16)
MOV   #KM+PUM, PSW    ; Kernel mode, prev user
MOV   #-1, -(6)       ; MOVE -1 into K stack (R6)
INC   @#SR0           ; Enable MEM MGT
MTPI  %6              ; %16 ←(6)+
```

The 0 in R16 is now replaced with -1 from the contents of the kernel stack.

To place info on the user stack if the status is set to kernel mode, prev user mode, 3 separate steps are needed.

```
MFPI  %6              ; Get content of R16=user pointer
MOV   #DATA, -(6)     ; put data on current stack
MTPI  @(6)+           ; @(6)+ [final address relocated]←
                      (R6)+
```

The data desired is obtained from the kernel stack then the destination address is obtained from the kernel stack and relocated through the previous mode.

Mode Description

In Kernel-mode the operating program has unrestricted use of the machine. The program can map users' programs anywhere in core and thus explicitly protect key areas (including the device registers and the Processor Status word) from the User operating environment.

In User mode a program is inhibited from executing a HALT instruction and the processor will trap through location 10 if an attempt is made to execute this instruction. A RESET instruction results in execution of a NOP (no-operation) instruction.

There are two stacks called the Kernel Stack and the User Stack, used by the central processor when operating in either the Kernel or User mode, respectively.

Stack Limit violations are disabled in User mode. Stack protection is provided by memory protect features.

Interrupt Conditions

The Memory Management Unit relocates all addresses. Thus, when Management is enabled, all trap, abort, and interrupt vectors are considered to be in Kernel mode Virtual Address Space. When a vectored transfer occurs, control is transferred according to a new Program Counter (PC) and Processor Status Word (PS) contained in a two-word vector relocated through the Kernel Active Page Register Set.

When a trap, abort, or interrupt occurs the "push" of the old PC, old PS is to the User/Kernel R6 stack specified by CPU mode bits 15, 14 of the new PS in the vector (00 = Kernel, 11 = User). The CPU mode bits also determine the new APR set. In this manner it is possible for a Kernel mode program to have complete control over service assignments for all interrupt conditions, since the interrupt vector is located in Kernel space. The Kernel program may assign the service of some of these conditions to a User mode program by simply setting the CPU mode bits of the new PS in the vector to return control to the appropriate mode.

User Processor Status (PS) operates as follows:

PS Bits	User RTI, RTT	User Traps, Interrupts	Explicit PS Access
Cond. Codes (3-0)	loaded from stack	loaded from vector	*
Trap (4)	loaded from stack	loaded from vector	cannot be changed
Priority (7-5)	cannot be changed	loaded from vector	*
Previous (13-12)	cannot be changed	copied from PS (15, 14)	*
Current (15-14)	cannot be changed	loaded from vector	*

* Explicit operation can be made if the Processor Status is mapped in User space.

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 hardware
0001		JUMP TO SRVICE	/ Trap location, contains jump to relocatable
3600	SRVICE	STORE ACCUMULATOR IN SAVAC	/ service routine

```
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.

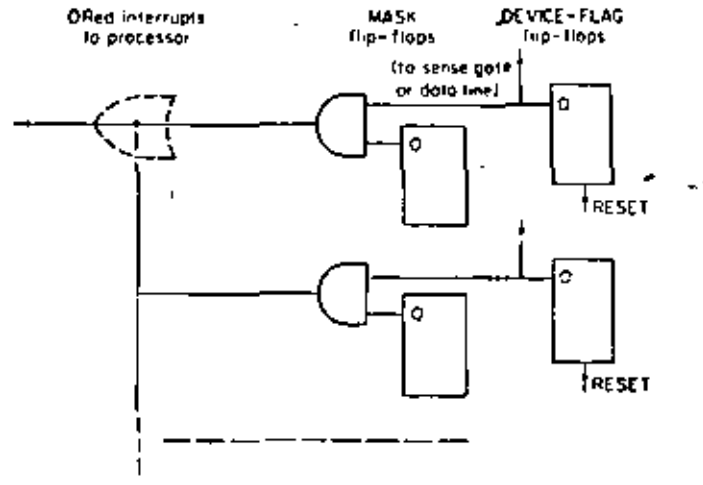


Fig. 5-9. Interrupt masking. The mask flip-flops are treated as a control register (*mask register*), which can be cleared and loaded by I/O instructions.

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

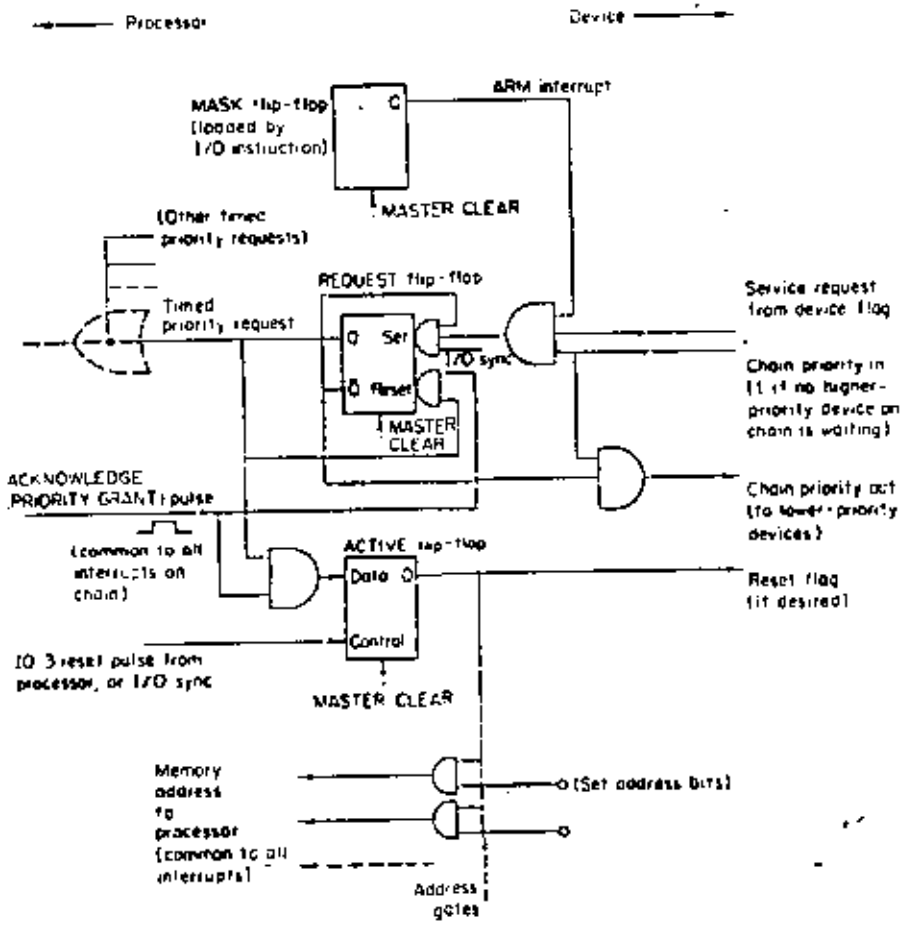


Fig. 5-10*a*. Priority-chain timing/queuing logic for one device (see also the timing diagram of Fig. 5-12). The ACKNOWLEDGE line is common to all interrupts on the chain. Note how the flip-flops are timed by the processor-supplied I/O SYNC pulses. MASTER CLEAR is issued by the processor whenever power 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

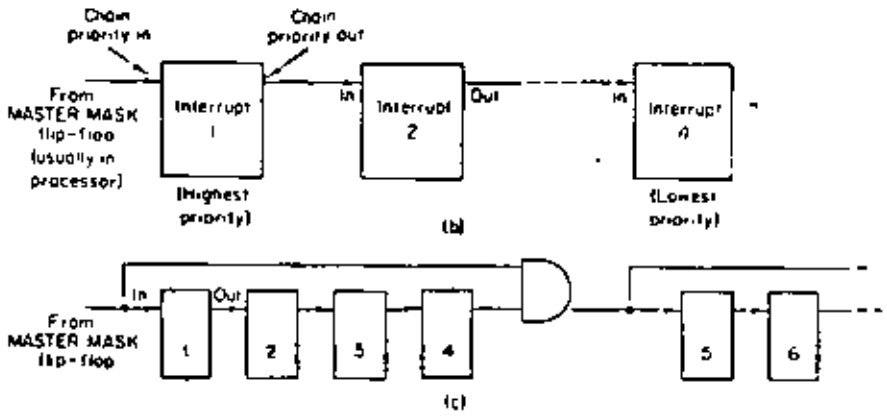


Fig. 5-10*b* and *c*. Wired-chain priority-propagation circuits. Since each subsystem (and its associated wiring) delays the propagated REQUEST flip-flop steps (Fig. 5-10*a*) by 10 to 30 nsec, the simple chain of Fig. 5-10*b* should not have more than four to six links; the circuit of Fig. 5-10*c* bypasses priority-inhibiting steps for faster propagation (*based on Ref.* 10).

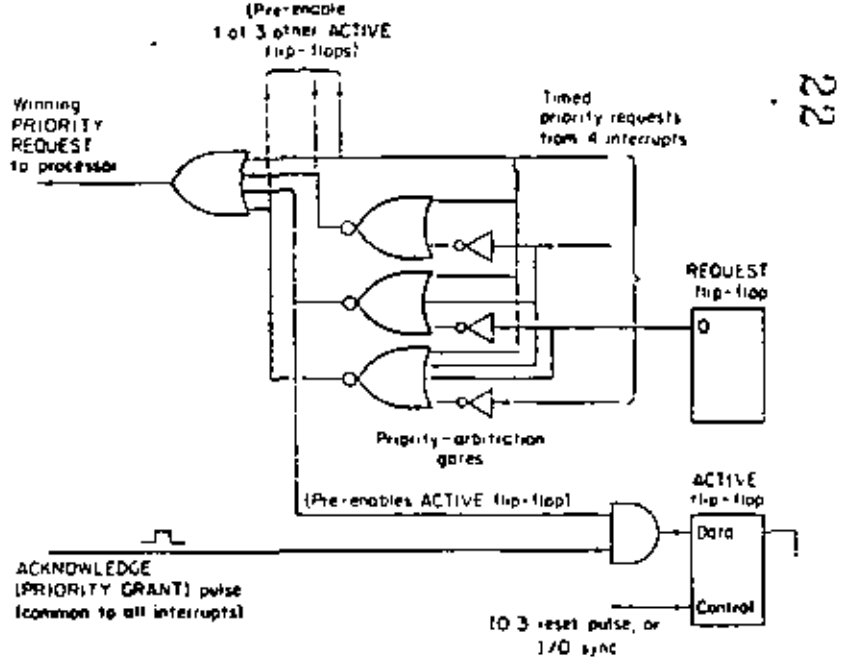


Fig. 5-11. This modified version of the priority-interrupt logic in Fig. 5-10*a* has priority-propagation gates at the output rather than at the input of the REQUEST flip-flop. Again, many similar circuits exist.

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
          S    E ACCUMULATOR IN     PTR
          JU   . 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 re[illegible] twice as in Secs. 5-11, 5-12, and 5-15*a*.

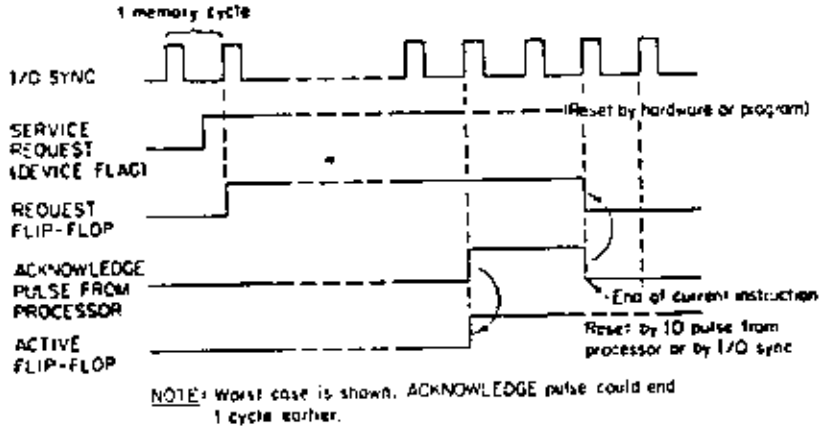


Fig. 5-12. Timing diagram for the priority-interrupt logic of Figs. 5-10 and 5-11. The ACKNOWLEDGE pulse remains ON until the trap address is transferred (either immediately over special address lines or by a programmed instruction).

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).

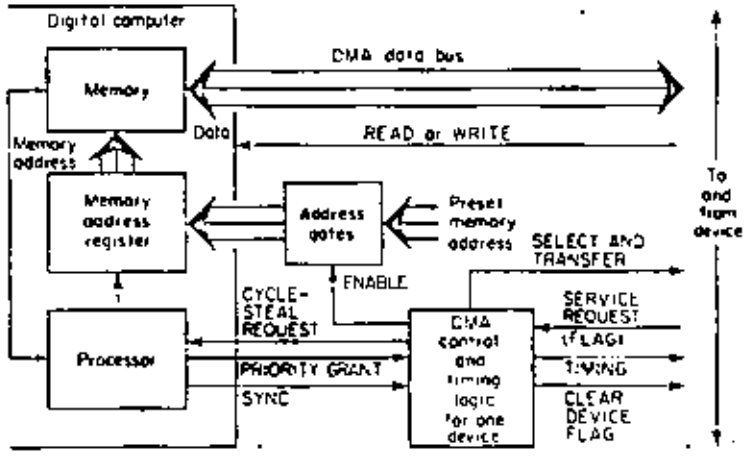
If very fast interrupt service is not a paramount consideration, *we can get around reentrant coding by programming interrupt masks which simply prevent interruption of critical service routines*.

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 memor ks 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.



25

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.

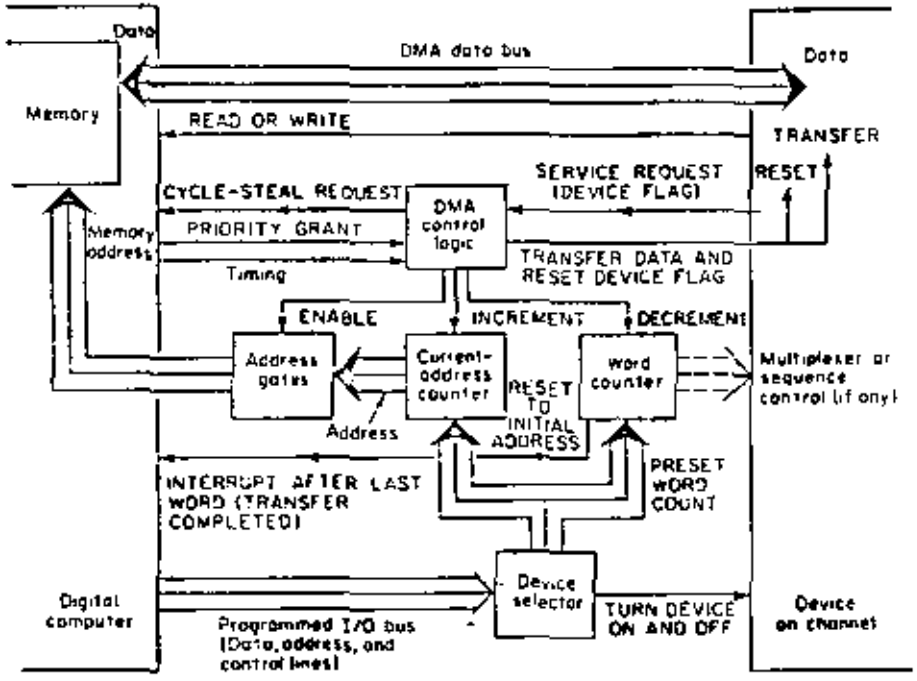


Fig. 5-14. A simple data channel for automatic block transfers.

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

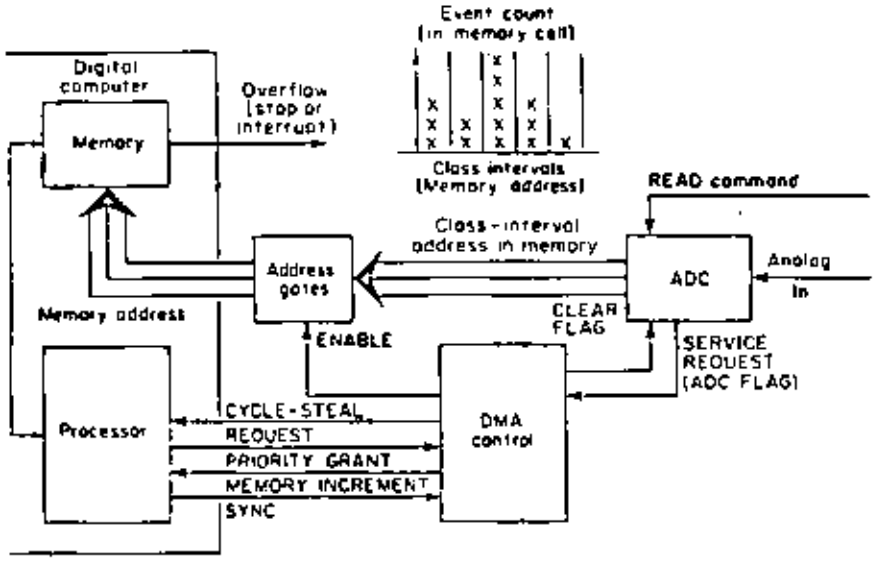


Fig. 5-15a. Memory-increment technique of measuring amplitude distributions (*based on Ref.* 6).

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

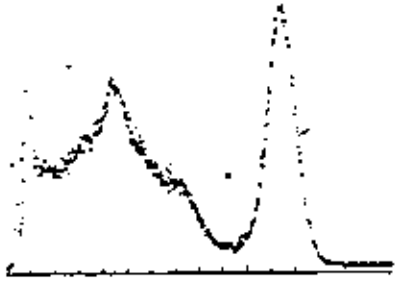


Fig. 5-15b. An amplitude-distribution display obtained by the method of Fig. 5-15a. (*Digital Equipment Corporation*)

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

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \qquad \overline{X^2} = \frac{1}{n}\sum_{i=1}^{n} X_i^2$$

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.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

ELEMENTOS DE UNA MINICOMPUTADORA

M. en C. MARCIAL PORTILLA ROBERTSON

JUNIO, 1980

7-6 RANDOM-ACCESS MEMORIES

A random-access memory (RAM) sometimes called a read/write memory, is equivalent to a group of addressable buffer registers. After applying an address, you can either read out (load) a word into the addressed register.

General Ideas

RAMs have been built with magnetic cores, bipolar transistors, MOSFETs, etc. The core RAM, the oldest of online computer memories, has the advantage of being *nonvolatile*; this means data is retained in the core registers even though the power is turned off. The bipolar and MOS RAMs, on the other hand, are volatile; shut off the power and the stored data is lost.

Semiconductor RAMs may be either static or dynamic. The static RAM uses bipolar or MOS flip-flops; data is retained indefinitely as long as power is applied to the flip-flops. On the other hand, dynamic RAM contains capacitors and MOSFETs that store the data bits as capacitor charges. Because all the stored data must be *refreshed* every few milliseconds by a separate clock.

The memory elements in static and dynamic RAMs may contain one, two, three, or four transistors. We do not have the time to go into the circuit design of static and dynamic RAMs. For the remainder of this book, our discussion is limited to basic ideas behind the static RAM.[1]

Three-State RAM

Figure 7-12 is the symbol for a three-state static RAM. The binary word ADDRESS selects the particular RAM register to be read out or written into.

The control signals *ME* and *WE* (memory enable and write enable) select a hold, read, or write operation as summarized in Table 7-3. Briefly, here's what happens. A low *ME* disables the RAM, equivalent to a hold or do-nothing operation; at this time, *WE* is a don't care. A high *ME* and low *WE* produce a read operation; in this case, the addressed word appears at the RAM output. Finally, a high *ME* and high *WE* give a write operation; this means D_{in} is stored in the addressed register.

[1] If interested in core RAMs, dynamic RAMs, and other memory topics, see A. P. Malvino and D. P. Leach, *Digital Principles and Applications*, McGraw-Hill Book Company, New York, 1975, Chap. 12.

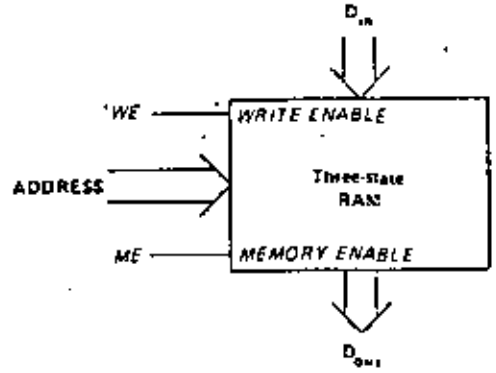


Fig. 7-12. Symbol for three-state static RAM.

As shown in Table 7-3, the output lines are floating during a hold or write operation but connected during a read operation. Memorize Table 7-3 for later use.

TABLE 7-3. RAM ACTION

ME	*WE*	*Operation*	*Output*
0	X	Hold	Floating
1	0	Read	Connected
1	1	Write	Floating

A static RAM is *asynchronous*, meaning unclocked. Instead, during a read or write operation, data is transferred without a clock signal.

During a write operation, the address and data inputs of Fig. 7-12 should be applied before *WE* goes high; this allows for setup time. Also, the address and data inputs must be held after *WE* goes low; this satisfies hold time. Data sheets usually specify the setup and hold times of RAMs.

EXAMPLE 7-7.

What are the following setup and hold times?

Address to *WE* = 5 ns (setup)
Data to *WE* = 5 ns (setup)
Address from *WE* = 0 ns (hold)
Data from *WE* = 10 ns (hold)

SOLUTION.

The address-to-*WE* setup time is 5 ns. This means the address must be applied at least 5 ns before *WE* goes high; otherwise, the manufacturer does not guarantee that the correct register is written into.

The data-to-*WE* setup time is 5 ns. Therefore, the data has to be applied at least 5 ns before the *WE* goes high. Unless this condition is satisfied, we cannot be sure the data will be correctly entered.

The address-from-*WE* hold time is 10 ns. This means the address must be held at least 10 ns after *WE* goes low.

The data-from-*WE* hold time is also 10 ns. So for a valid write operation, the data must be held at least 10 ns after *WE* goes low.

EXAMPLE 7-8.

How does the RAM of Fig. 7-13 differ from the one of Fig. 7-12?

SOLUTION.

It's still a static RAM, but the control inputs are inverted. Those bubbles at the *WE* and *ME* inputs mean complementary operation. In other words, the *WE* and *ME* entries of Table 7-3 are inverted: 1X gives a hold, 01 produces a read, and 00 gives a write.

7-4. THE MEMORY DATA REGISTER

In general, data has to be set up before *WE* goes high and held after *WE* goes low. This suggests the *memory data register* (MDR), a register that temporarily stores the data to be written into the memory. In other words, the MDR is loaded before a write operation; to satisfy hold time, the MDR retains its contents after a write operation.

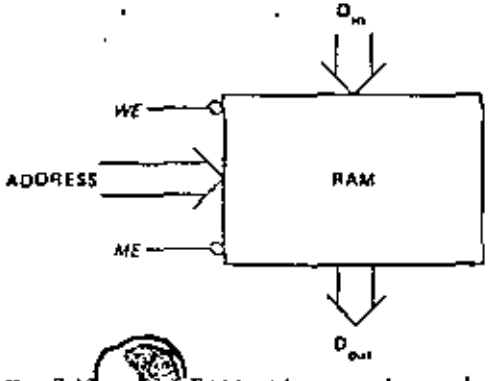


Fig. 7-13. RAM with inverted control inputs.

The SAP-2 computer uses a 256 × 12 RAM (see Fig. 7-14). The MAR again supplies the address word, but this time it contains 8 bits rather than 4. This means the binary addresses are from

ADDRESS = 0000 0000

to

ADDRESS = 1111 1111

The equivalent decimal addresses therefore are 0 to 255.

The 256 × 12 RAM can store 256 words of 12 bits each. These words are loaded into R_0, R_1, R_2, . . . , R_{255}. Noninverted *WE* and *ME* signals control the operation of the RAM; a low *ME* creates the hold condition with a floating output, a high *ME* and low *WE* result in a read onto the W bus, a high *ME* and high *WE* produce a write operation with a floating output.

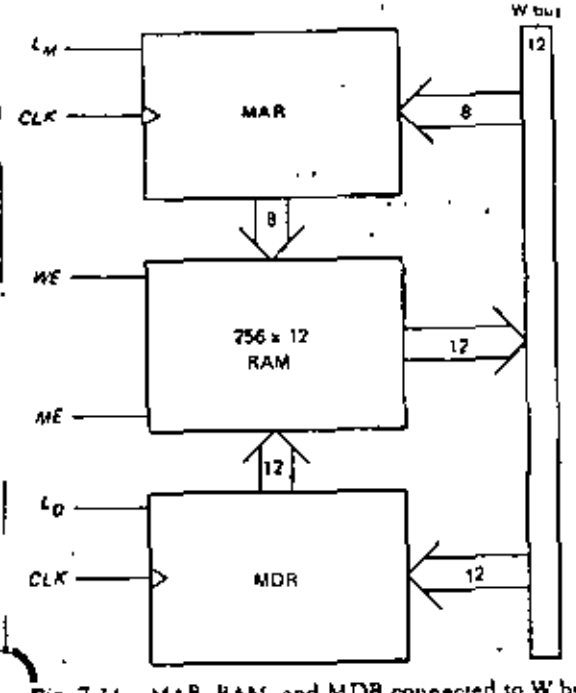


Fig. 7-14. MAR, RAM, and MDR connected to W bus.

The memory data register of Fig. 7-14 is a 12-bit buffer register. Its two-state output goes to the RAM data inputs. Therefore, the MDR supplies the word to be stored in the RAM. As indicated, the MDR has a *LOAD* (L_D) input. A low L_D prevents the bus word from entering the MDR; a high L_D and positive clock edge load the bus word into the MDR.

Incidentally, the memory data register (MDR) is also known as the memory buffer register (MBR). In this book, we prefer using MDR because it pinpoints the function of the register; the D in MDR stands for "data"; this is more concrete than using B, which stands for "buffer."

EXAMPLE 7-9.

Figure 7-15 shows part of a computer. A three-state program counter delivers an 8-bit address to the W bus when E_P is high. As usual, a high C_P and positive clock edge increment the PC.

An *input register* (I) receives instruction and data words from the outside world. At the moment, the input word is I_0. This word comes from a *peripheral device* such as a punched-card reader, a magnetic-tape reader, a teletypewriter, etc.

Initially, a *CLR* pulse resets the program counter to 0s. If the input word is

$$I_0 = 1100\ 0001\ 1001$$

what will happen if a positive clock edge occurs once during each of the following control words?

$$C_P E_P L_M W_E\ M_E L_D L_I E_I = 0110\ 0000$$
$$C_P E_P L_M W_E\ M_E L_D L_I E_I = 0000\ 0010$$
$$C_P E_P L_M W_E\ M_E L_D L_I E_I = 0000\ 0101$$
$$C_P E_P L_M W_E\ M_E L_D L_I E_I = 0001\ 1000$$
$$C_P E_P L_M W_E\ M_E L_D L_I E_I = 1000\ 0000$$

SOLUTION.

In the first control word, E_P and L_M are high. Therefore, the contents of the program counter set up the MAR via the W bus. After the positive clock edge strikes,

$$MAR = PC$$

Since the PC was initially cleared,

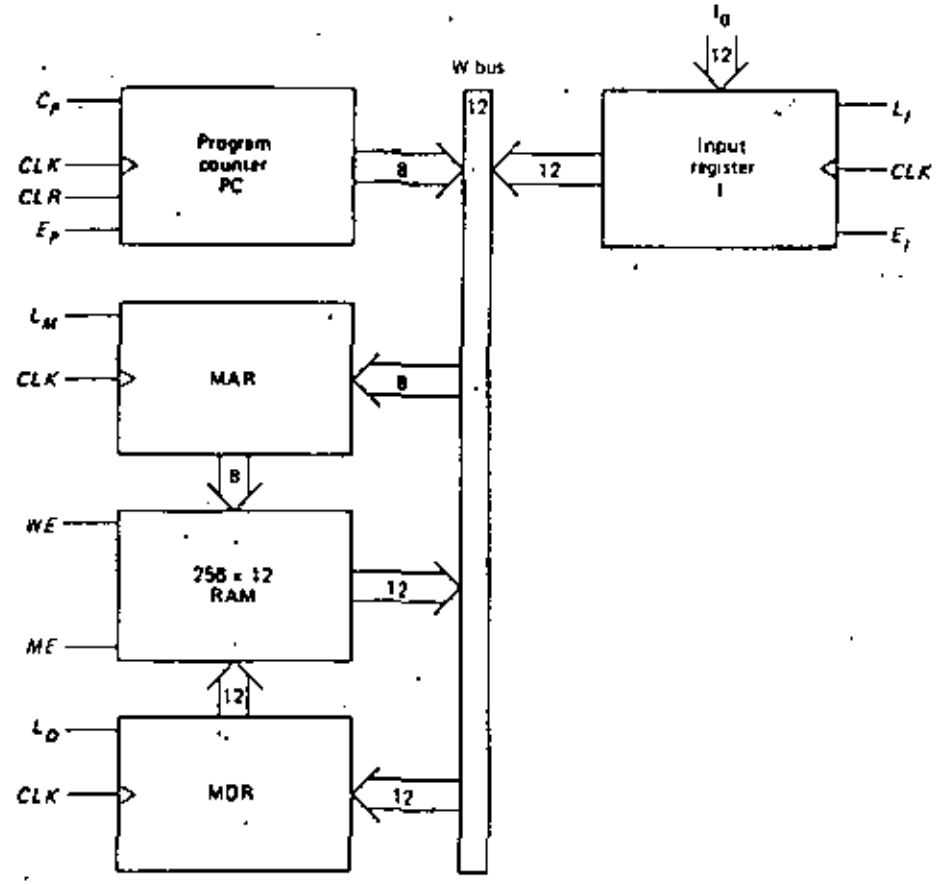


Fig. 7-15. Part of the SAP-2 computer.

In the second control word, L_I is high. When the next positive clock edge hits, the instruction or data word from the peripheral device is loaded into the input register to get

$$I = 1100\ 0001\ 1001$$

L_D and E_I are high in the third control word. This means the contents of the input register set up the memory data register via the W bus. The next positive clock edge therefore loads the MDR with the input word to get

$$MDR = 1100\ 0001\ 1001$$

In the fourth control word, WE and ME are high. This produces a write operation. Since the MAR is addressing the R_0 register,

Computer architecture depends on the application. If a computer is to be *timeshared* (used by many people at the same time), it will be large, powerful, and expensive. On the other hand, a computer *dedicated* to only one program is usually small, limited, and inexpensive (like a microcomputer).

The SAP (simple as possible) computer has been designed for you, the beginner. The main purpose of SAP is to show you all the crucial ideas behind computer operation without burying you in unnecessary detail. But even a simple computer like SAP covers many advanced concepts. Rather than bombard you with too much all at once, we will examine three different generations of the SAP computer.

SAP-1 is the first phase in the evolution toward modern computers. Although primitive, SAP-1 is a big step for a beginner. Really dig into this chapter; master SAP-1, its architecture, its programming, and its internal circuit operation. Then you will be ready for SAP-2.

8-1. ARCHITECTURE

Figure 8-1 shows the *architecture* (structure) of SAP-1. All register outputs to the W bus are three-state; this allows orderly communication and data transfer. All register outputs not connected to the W bus are two-state; these outputs continuously drive the boxes they are connected to.

Most of the registers of Fig. 8-1 were discussed earlier. What follows is a brief description of each box; detailed explanations come later.

Program Counter

The program counter (PC) counts from 0000 to 1111, equivalent to hexadecimal 0 through F. The PC is reset to 0000 before each computer run. After an instruction has been fetched, the PC is incremented by one. Therefore, at the beginning of each fetch cycle, the PC holds the address of the current instruction.

Memory Address Register

The memory address register (MAR) receives binary addresses from the program counter. Because the PC counts from 0000 upward, the MAR selects the ROM words stored at hexadecimal addresses 0 upward. Note that the output of the MAR is two-s

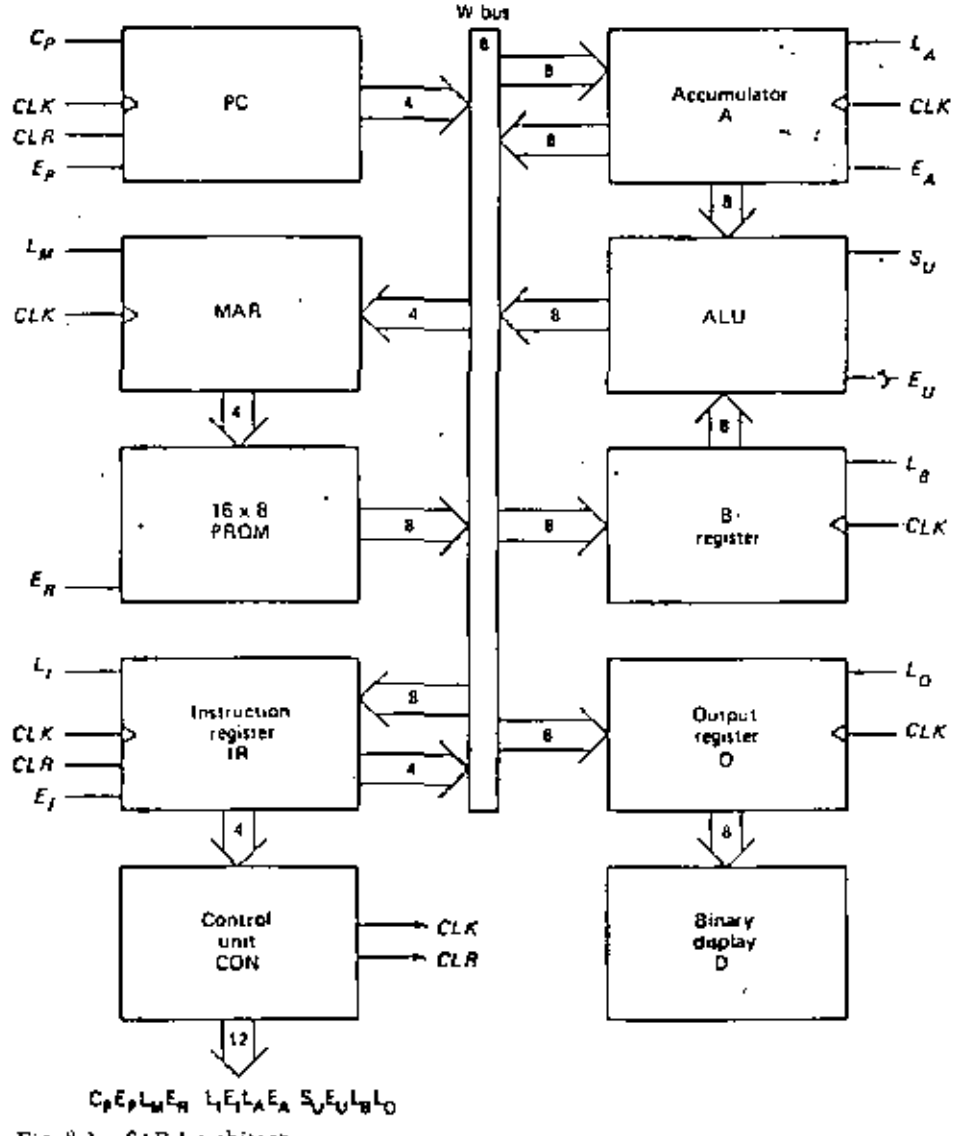


Fig. 8-1. SAP-1 architecture.

Read-only Memory

The 16 × 8 PROM is switch programmable; therefore, by closing and opening switches, you can store any 16 words of 8 bits each. These words, located at addresses 0 through F, are symbolized by R_0, R_1, R_2, . . . , R_F. When E_R is high, the addressed word is read onto W bus.

Instruction Register

Examples 7-5 and 7-6 showed you how ROM instructions are read onto the W bus. Each ROM instruction is then loaded into the *instruction register* (IR) by applying a high L_I and positive clock edge.

The contents of the instruction register are split into two smaller words. The four MSBs (most significant bits) are a two-state output that goes directly to the control unit. The four LSBs, on the other hand, are a three-state output that is read onto the W bus when E_I is high.

Control Unit

The *control unit* (CON) makes a computer what it is: an automatic data-processing machine. Before each computer run, CON sends a *CLR* signal to the instruction register and program counter. This wipes out the last instruction in the IR and resets the PC to 0000.

CON also sends a clock signal to all registers; this synchronizes the operation of the computer, ensuring well-defined internal states. In other words, all register transfers and changes occur on the positive edge of a common clock signal.

The bits from the bottom of CON form a 12-bit control word:

$$\text{CON} = C_P E_P L_M E_R \; L_I E_I L_A E_A \; S_U E_U L_B L_O$$

This word determines how the registers will react to the next positive clock edge. For instance, a high C_P means the program counter will be incremented, a high E_P and L_M mean the contents of the PC are loaded into the MAR, etc.

CON is like the conductor of a symphony. It synchronizes and orchestrates all the different pieces in the SAP-1 computer. By generating different routines (similar to Examples 7-5 and 7-9), CON makes the other circuits fetch an instruction, load the accumulator with a number, add two numbers, etc.

Accumulator

Chapter 6 introduced the accumulator (A), a group of flip-flops that store intermediate answers during a computer run. In Fig. 8-1, the accumulator has two outputs. The two-state output goes directly to the arithmetic-logic unit. The three-state output connects to the W bus. Therefore, the 8-bit accumulator word continuously drives the ALU; the same word appears on the W bus when E_A is high.

Arithmetic-Logic Unit

The arithmetic-logic unit (ALU) contains a 2's complement adder/subtractor (review Secs. 4-7 through 4-9 if necessary). When the subtract (S_U) input is low, the ALU forms the algebraic sum; therefore, its contents are

$$\text{ALU} = \text{A} + \text{B}$$

When S_U is high, the algebraic difference appears:

$$\text{ALU} = \text{A} + \text{B}'$$

(Recall the 2's complement is equivalent to decimal sign change.)

The ALU is an *asynchronous* (unclocked) circuit; this means its contents can change as soon as the input words change. When E_U is high, these contents appear on the W bus.

B Register

A high L_B and positive clock edge load the word on the W bus into the B register. The two-state output of the B register drives the ALU, supplying the number to be added or subtracted from the contents of the accumulator.

Output Register

At the end of a computer run, the accumulator contains the answer to the problem being solved. At this point, we need to transfer the answer to the outside world. This is where the *output register* (O) is used. When E_A and L_O are high, the next positive clock edge loads the accumulator word into the output register.

Typical computers have several output registers, which are connected to *interface circuits* that drive peripheral devices. In this way, processed data can drive printers, cathode-ray tubes, teletypewriters, etc. (An interface circuit tailors the data for each device.)

Binary Display

The binary display (D) is a horizontal row of eight light-emitting diodes (LEDs). Because each LED connects to one flip-flop of the output regis-

Therefore, after we've transferred an answer from the accumulator to the output register, we can see the answer in binary form.

8-2. INSTRUCTION SET

A computer is a useless pile of hardware until someone programs it. This means loading step-by-step instructions into the memory before the start of a computer run. But before you can program a computer, you must learn its *instruction set*, the basic operations it can perform. The SAP-1 instruction set follows.

LDA

LDA stands for "load the accumulator." A complete LDA instruction includes a ROM word. LDA R_B, for example, means "load the accumulator with R_B." Therefore, given

$$R_B = 1111\ 0000$$

the execution of LDA R_B results in

$$A = 1111\ 0000$$

Similarly, LDA R_A means "load the accumulator with R_A," LDA R_F means "load the accumulator with R_F," and so on.

ADD

ADD is another SAP-1 instruction. A complete ADD instruction includes a ROM word. For instance, ADD R_B means "add R_B to the accumulator contents"; the sum in the ALU replaces the original contents of the accumulator.

Here's a numerical example. Suppose decimal 2 is in the accumulator and decimal 3 is in the R_B register. Then,

$$A = 0000\ 0010$$
$$R_B = 0000\ 0011$$

During the execution of ADD R_B, the following things happen. First, R_B is loaded into the B register to get

and almost instantly the ALU forms the sum of A and B:

$$ALU = 0000\ 0101$$

Second, the ALU contents are loaded into the accumulator to get

$$A = 0000\ 0101$$

As you see, the accumulator winds up with the sum.

The foregoing routine is used for all ADD instructions; the addressed ROM word goes to the B register and the ALU output to the accumulator. This is why the execution of LDA R_B adds R_B to the accumulator contents, the execution of LDA R_F adds R_F to the accumulator contents, and so on.

SUB

SUB is another SAP-1 instruction, and it too is incomplete without a ROM word. For example, SUB R_C means "subtract R_C from the contents of the accumulator"; the difference formed in the ALU replaces the original contents of the accumulator.

For a concrete example, assume decimal 7 is in the accumulator and decimal 3 is in the ROM R_C register. Then,

$$A = 0000\ 0111$$
$$R_C = 0000\ 0011$$

The execution of SUB R_C takes place as follows. First, R_C is loaded into the B register to get

$$B = 0000\ 0011$$

and almost instantly the ALU forms the difference of A and B:

$$ALU = 0000\ 0100$$

Second, the ALU contents are loaded into the accumulator to get

$$A = 0000\ 0100$$

So, the accumulator ends up with the difference.

The foregoing routine applies to all SUB instructic the ad-

dressed ROM word goes to the B register and the ALU output to the accumulator. This is why the execution of SUB R_C subtracts R_C from the contents of the accumulator, the execution of SUB R_E subtracts R_E from the accumulator, and so on.

OUT

The instruction OUT tells the SAP-1 computer to transfer the accumulator contents to the output register. After OUT has been executed, you can see the answer to the problem being solved.

OUT is complete by itself; that is, you do not have to include a ROM word when using OUT because the instruction does not involve the memory.

HLT

HLT stands for halt. This instruction tells the computer to stop processing data. HLT marks the end of a program, similar to the way a period marks the end of a sentence. You must use a HLT instruction at the end of every SAP-1 program; otherwise, you get *computer trash* (meaningless answers caused by runaway processing).

HLT is complete by itself; that is, you don't have to include a ROM word when using HLT because this instruction does not involve the memory.

Memory-Reference Instructions

LDA, ADD, and SUB are called *memory-reference instructions* (MRIs); you must include a ROM word when using these instructions. OUT and HLT, on the other hand, are not MRIs because they do not involve the memory.

Mnemonics

LDA, ADD, SUB, OUT, and HLT are the instruction set for SAP-1. Abbreviated instructions like these are called *mnemonics* (memory aids). Mnemonics are popular in computer work because typical computers have hundreds of instructions in their instruction sets. Table 8-1 summarizes the SAP-1 instruction set.

TABLE 8-1. INSTRUCTION SET

Mnemonic	*Operation*
LDA	Load accumulator with ROM word
ADD	Add ROM word to accumulator
SUB	Subtract ROM word from accumulator
OUT	Load output register with accumulator word
HLT	Halt

EXAMPLE 8-1.

Here's a SAP-1 program:

LDA R_B
ADD R_A
ADD R_B
ADD R_C
SUB R_D
OUT
HLT

What does each instruction do?

SOLUTION.

The first instruction loads the accumulator with R_B:

$$A = R_B$$

The second instruction adds R_A to the accumulator contents:

$$A = R_B + R_A$$

Similarly, the next two instructions add R_B and R_C:

$$A = R_B + R_A + R_B + R_C$$

The SUB instruction subtracts R_D from the accumulator contents:

$$A = R_B + R_A + R_B + R_C - R_D$$

The OUT instruction loads this accumulator word into the output register; therefore, the binary display shows

D = A

The HLT instruction stops the data processing.

8-3. PROGRAMMING

The X's of Fig. 8-2 are switches on the operator panel of SAP-1. These switches program the contents of the memory. A closed switch loads a binary 1; an open switch, a binary 0. Since there are 16 ROM registers, we can store 16 words at addresses 0 through F.

To program instruction and data words into the memory of SAP-1, use the *operation code* (op-code) of Table 8-2. The op-code bits go into the four MSB positions and the address bits into the four LSB positions.

TABLE 8-2. OPERATION CODE

Mnemonic	*Op-Code*
LDA	0000
ADD	0001
SUB	0010
OUT	1110
HLT	1111

Here's how to program LDA R_F into the R_0 register. The op-code for LDA is 0000, and the binary address for R_F is 1111. Therefore, you must program the switches in the R_0 register like this:

op-op-op-op cl-cl-cl-cl

Since op (open) loads a binary 0 and cl (closed) loads a binary 1, the R_0 register has been programmed with the word

$R_0 = 0000\ 1111$ (LDA R_F)

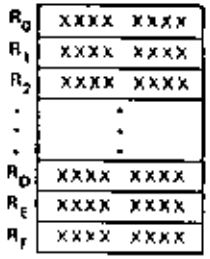


Fig. 8-2. Programming switches for 16 × 8 ROM.

How would you program ADD R_E in the R_1 register and SUB R_D in the R_2 register? Again, put the op-code bits in the MSB positions and the address bits in the LSB positions. After the switches are programmed, the stored words are

$R_1 = 0001\ 1110$ (ADD R_E)
$R_2 = 0010\ 1101$ (SUB R_D)

The OUT and HLT instructions don't use address bits; therefore, we can disregard the switches in the LSB positions. For example, to store OUT in the R_3 register and HLT in the R_4 register, set the switches to get

$R_3 = 1110\ XXXX$
$R_4 = 1111\ XXXX$

Because the address bits are not used, the LSB switches can be open or closed.

EXAMPLE 8-2.

Machine language is the binary words a particular computer responds to. Translate the program of Example 8-1 into SAP-1 machine language and store the coded instructions at hexadecimal addresses 0 upward.

SOLUTION.

All we have to do is replace the mnemonics by op-code bits and the ROM words by address bits. For instance, LDA R_9 translates into 0000 1001. After translating the entire program of Example 8-1 and loading it into the ROM, we have

$R_0 = 0000\ 1001$ (LDA R_9)
$R_1 = 0001\ 1010$ (ADD R_A)
$R_2 = 0001\ 1011$ (ADD R_B)
$R_3 = 0001\ 1100$ (ADD R_C)
$R_4 = 0010\ 1101$ (SUB R_D)
$R_5 = 1110\ XXXX$ (OUT)
$R_6 = 1111\ XXXX$ (HLT)

Incidentally, any program like the foregoing that's written in machine language is called an *object program*. The original program with mnemonics and ROM words is called a *source program*

the operator translates the source program into an object program when programming the ROM switches.

A final point. The four MSBs of a machine-language instruction in SAP-1 specify the operation and the four LSBs give the address; from now on, we will refer to the MSBs as the *instruction field* and to the LSBs as the *address field*. Symbolically,

$$R_n = \text{XXXX XXXX}$$

Instruction field — MSBs
Address field — LSBs

EXAMPLE 8-3.

How can we simplify the appearance of the source program given in Example 8-1?

SOLUTION.

The R in each memory-reference instruction is redundant information. In other words, when we translate to machine language, all we need is the subscript. For instance, LDA 9 gives the same information as LDA R_9.

By deleting all Rs from the memory-reference instructions, the source program of Example 8-1 becomes

```
LDA 9
ADD A
ADD B
ADD C
SUB D
OUT
HLT
```

This is the standard practice in industry; from now on, we will write source programs like this.

EXAMPLE 8-4.

How would you program SAP-1 to solve this arithmetic problem?

$$16 + 20 + 24 + 28 - 32$$

SOLUTION.

You can use the program of the preceding example provided decimal 16 is loaded into the R_9 register, decimal 20 into the R_A register, decimal 24 into the R_B register, and so on.

After programming the ROM switches, the memory words are

$R_0 = 0000\ 1001$	(LDA 9)
$R_1 = 0001\ 1010$	(ADD A)
$R_2 = 0001\ 1011$	(ADD B)
$R_3 = 0001\ 1100$	(ADD C)
$R_4 = 0010\ 1101$	(SUB D)
$R_5 = 1110\ XXXX$	(OUT)
$R_6 = 1111\ XXXX$	(HLT)
$R_9 = 0001\ 0000$	(16_{10})
$R_A = 0001\ 0100$	(20_{10})
$R_B = 0001\ 1000$	(24_{10})
$R_C = 0001\ 1100$	(28_{10})
$R_D = 0010\ 0000$	(32_{10})

Notice the program is stored ahead of the data. That is, the initial program instruction goes into the R_0 register, and remaining instructions go into successive addresses. This is essential in SAP-1 because the program counter starts with binary address 0000. Since the PC holds the address of the instruction to be fetched next from the memory, we must load the program ahead of the data.

Incidentally, a *stored-program* computer is one in which an object program is stored in the memory before a computer run. SAP-1 is an example of a stored-program computer. It uses front-panel switch programming, the simplest but slowest way to load a program into a computer memory.

EXAMPLE 8-5.

Show the contents of the accumulator after the first five instructions are executed in the program of the preceding example. Show the contents of the output register after the OUT instruction is executed.

SOLUTION.

For convenience, the source program is repeated along with the decimal data:

```
R0 = LDA 9
R1 = ADD A
```

$R_2 = \text{ADD B}$
$R_3 = \text{ADD C}$
$R_4 = \text{SUB D}$
$R_5 = \text{OUT}$
$R_6 = \text{HLT}$
$R_9 = 16_{10}$
$R_A = 20_{10}$
$R_B = 24_{10}$
$R_C = 28_{10}$
$R_D = 32_{10}$

(In these equations, the = sign stands for "is equivalent to.")

After LDA 9 is executed, the contents of the accumulator are

$$A = 0001\ 0000 \qquad (16_{10})$$

The execution of ADD A gives

$$A = 0010\ 0100 \qquad (36_{10})$$

After ADD B is executed,

$$A = 0011\ 1100 \qquad (60_{10})$$

When ADD C is executed,

$$A = 0101\ 1000 \qquad (88_{10})$$

Executing SUB D gives

$$A = 0011\ 1000 \qquad (56_{10})$$

The accumulator contains the answer to the problem. After OUT is executed, the contents of the output register are

$$O = 0011\ 1000 \qquad (56_{10})$$

At this point, HLT stops the data processing.

8-4. FETCH CYCLE

The control unit is the key to a computer's automatic operation. After the program and data are loaded into the memory, a start signal turns control over to CON. This unit then generates the control words that fetch and execute each instruction.

While each instruction is fetched and executed, the computer passes through different *machine phases*, periods during which register contents change. Let's find out more about these phases.

Ring Counter

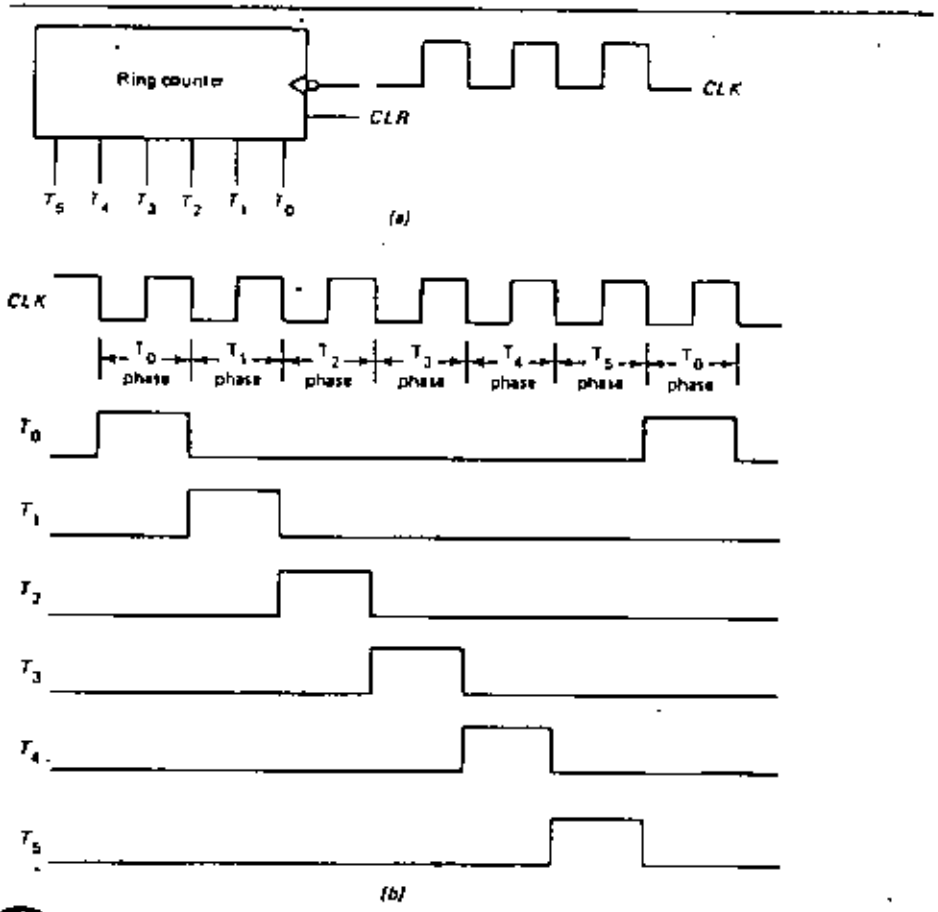
The ring counter of Fig. 8-3*a* (part of SAP-1's control unit) has an output of

$$T = T_5T_4T_3T_2T_1T_0$$

At the beginning of a computer run, the ring word is

$$T = 000001$$

Successive negative clock edges produce ring words of



8-3. Ring counter. (*a*) Symbol. (*b*) Clock and timing pulses.

T = 000010
T = 000100

.
.
.

T = 100000

Then, the ring counter resets to 000001 and the cycle repeats. Each ring word represents one machine phase.

Figure 8-3*b* shows the timing pulses out of the ring counter. The initial phase T_0 starts with a negative clock edge and ends with the next negative clock edge. During this phase, the T_0 bit out of the ring counter is high.

During the next phase, T_1 is high; the following phase has a high T_2; then a high T_3, and so on. As you can see, the ring counter produces six phases. Each instruction is fetched and executed during these six phases.

Address Phase

The T_0 phase is called the *address phase* because the address in the PC is transferred to the MAR during this phase. Figure 8-4*a* shows the computer sections that are active during this phase. (Active parts are light; inactive parts are dark.)

During the address phase, E_P and L_M are high; all other control bits are low. This means CON is sending out a control word of

$$\begin{aligned}\text{CON} &= C_P E_P L_M E_R\ L_I E_I L_A E_A\ S_U E_U L_B L_O \\ &= 0110\ 0000\ 0000\end{aligned}$$

during this phase.

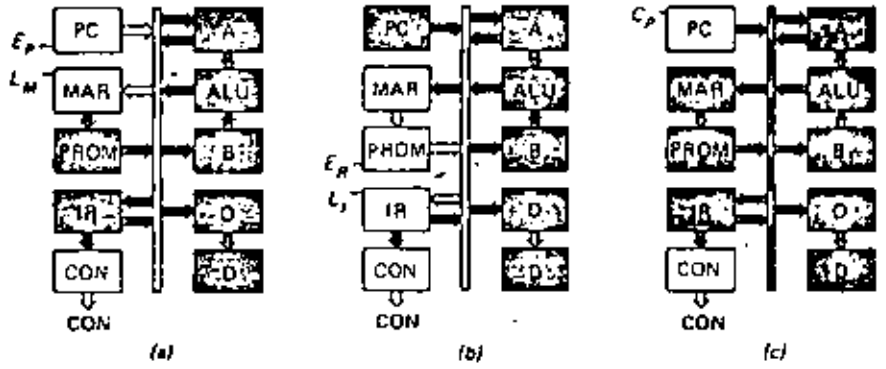


Fig. 8-4. Fetch routine. (*a*) Address phase. (*b*) Memory phase. (*c*) Increment phase.

Memory Phase

Figure 8-4*b* shows the active parts of SAP-1 during the T_1 phase. This phase is called the *memory phase* because the addressed ROM instruction is transferred from the memory to the instruction register. During the memory phase, CON is producing a control word of

$$\begin{aligned}\text{CON} &= C_P E_P L_M E_R\ L_I E_I L_A E_A\ S_U E_U L_B L_O \\ &= 0001\ 1000\ 0000\end{aligned}$$

As you see, the E_R and L_I bits are high.

Increment Phase

The T_2 phase is called the *increment phase* because the program counter is incremented during this phase. Figure 8-4*c* shows the active parts of SAP-1 during the increment phase. The only high control bit during this phase is C_P, and the word out of CON is

$$\begin{aligned}\text{CON} &= C_P E_P L_M E_R\ L_I E_I L_A E_A\ S_U E_U L_B L_O \\ &= 1000\ 0000\ 0000\end{aligned}$$

Fetch Cycle

The address, memory, and increment phases represent the *fetch cycle* of SAP-1. During the fetch cycle, CON sends out the three control words just described. Table 8-3 repeats this control routine.

TABLE 8-3. FETCH ROUTINE

Phase	$C_P E_P L_M E_R$	$L_I E_I L_A E_A$	$S_U E_U L_B L_O$	*Effect*
T_0 (address)	0 1 1 0	0 0 0 0	0 0 0 0	MAR = PC
T_1 (memory)	0 0 0 1	1 0 0 0	0 0 0 0	IR = R_{RC}
T_2 (increment)	1 0 0 0	0 0 0 0	0 0 0 0	Increment PC

During the address phase, E_P and L_M are high; this means the PC sets up the MAR via the W bus. As shown earlier in Fig. 8-3*b*, a positive clock edge occurs midway through the address phase; this loads the MAR with the contents of the PC.

During the memory phase, E_R and L_I are high. Therefore, the addressed ROM register sets up the instruction register via the W bus. Midway through the memory phase, a positive clock edge loads the IR with the addressed ROM instruction.

C_P is the only high control bit during the increment phase. This sets up the PC to count positive clock edges. Halfway through the increment phase, a positive clock edge hits the PC and advances the contents by one.

8–5. EXECUTION CYCLE

The next three phases (T_3, T_4, and T_5) are the *execution cycle* of SAP-1. The register transfers during the execution cycle depend on the particular instruction being executed. For instance, LDA 9 requires different register transfers than ADD B. What follows are the control routines for different SAP-1 instructions.

LDA Routine

For a concrete discussion, let's assume the instruction register has been loaded with LDA 9:

$$\text{IR} = 0000\ \ 1001$$

During the T_3 phase, the instruction field 0000 goes to the CON, where it is decoded; the address field 1001 is loaded into the MAR. Figure 8-5*a* shows the active parts of SAP-1 during this execution phase. Note that E_I and L_M are high; all other control bits are low.

During the second execution phase (T_4), E_R and L_A go high. This means the addressed data word in the ROM will be loaded into the accumulator on the next positive clock edge (see Fig. 8-5*b*).

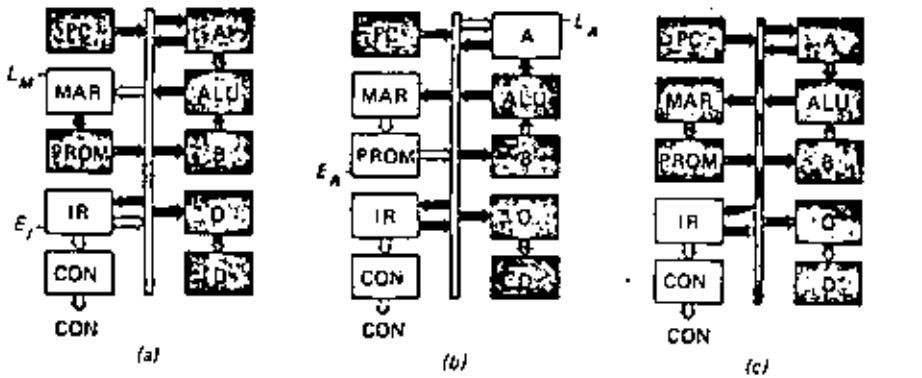


Fig. 8-5. LDA routine. (*a*) First execution phase. (*b*) Second execution phase. (*c*) Final execution phase.

T_5 is a *do-nothing* phase. During this third execution phase, all registers are inactive (Fig. 8-5*c*). This means CON is sending out a word containing all 0s.

Table 8-4 summarizes the control routine needed to *implement* (carry out) the LDA instruction. The first execution phase sends the address field in the instruction register to the MAR. The second execution phase loads the addressed data word into the accumulator. The third execution phase marks time; it does nothing.

TABLE 8-4. LDA ROUTINE

Phase	$C_P E_P L_M E_R$	$L_I E_I L_A E_A$	$S_U E_U L_B L_O$	*Effect*
T_3	0010	0100	0000	MAR = ADDRESS
T_4	0001	0010	0000	$A = B_{ADDRESS}$
T_5	0000	0000	0000	Do nothing

ADD Routine

Suppose at the end of the fetch cycle, the instruction register contains ADD B:

$$\text{IR} = 0001\ \ 1011$$

During the first execution phase (T_3), the instruction field goes to the CON and the address field to the MAR (see Fig. 8-6*a*). During this phase, E_I and L_M are high.

Control bits E_R and L_B are high during the next execution phase (T_4). This allows the addressed data word to set up the B register (Fig.

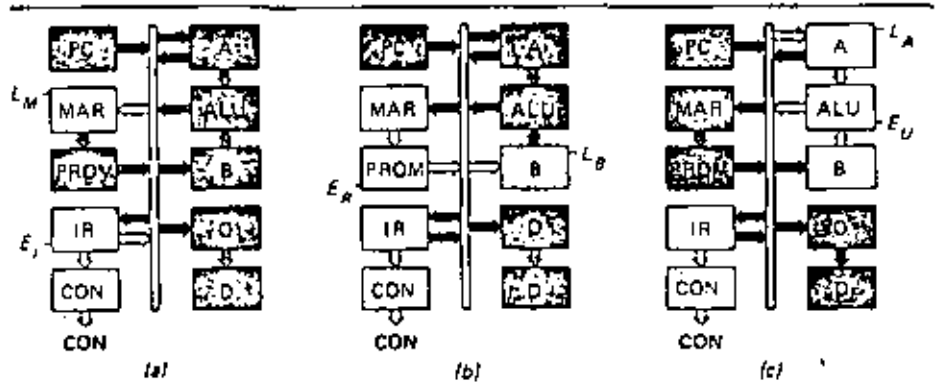


Fig. 8-6. ADD and SUB routines. (*a*) First execution phase. (*b*) Second execution phase. (*c*) Final execution phase.

8-6*b*). As usual, loading takes place midway through the phase when the positive clock edge hits the *CLK* input of the B register.

During the third execution phase (T_3), E_U and L_A are high; therefore, the ALU sets up the accumulator (Fig. 8-6*c*). Halfway through this phase, the positive clock edge loads the sum into the accumulator.

Incidentally, setup time and propagation delay time prevent racing of the accumulator during this final execution phase. When the positive clock edge hits in Fig. 8-6*c*, the accumulator contents change, forcing the ALU contents to change. These new ALU contents return to the accumulator input. But the new ALU contents don't get there until two propagation delays after the positive clock edge (one for the accumulator and one for the ALU). By then it's too late to set up the accumulator. This prevents accumulator racing (loading more than once on the same clock edge).

Table 8-5 summarizes the control routine needed to implement the ADD instruction. The first execution phase sends the address field in the instruction register to the MAR. The second execution phase loads the addressed data word into the B register. The third execution phase loads the ALU contents (SUM) into the accumulator.

TABLE 8-5. ADD ROUTINE

Phase	$C_P E_P L_M E_R$	$L_I E_I L_A E_A$	$S_U E_U L_B L_O$	*Effect*
T_2	0010	0100	0000	MAR = ADDRESS
T_4	0001	0000	0010	B = $R_{address}$
T_5	0000	0010	0100	A = SUM

SUB Routine

The SUB routine is similar to the ADD routine; the only difference is making S_U high during the final execution phase. Therefore, Figs. 8-6*a* through *c* show the active parts of SAP-1 during the three execution phases. (A high S_U drives the ALU in Fig. 8-6*c*.)

Table 8-6 summarizes the SUB routine. The first control word sends the address field from the instruction register to the MAR. The second control word loads the addressed data word into the B register. The third control word loads the ALU contents (DIFF) into the accumulator.

TABLE 8-6. SUB ROUTINE

Phase	$C_P E_P L_M E_R$	$L_I E_I L_A E_A$	$S_U E_U L_B L_O$	*Effect*
T_2	0010	0100	0000	MAR = ADDRESS
T_4	0001	0000	0010	B = $R_{address}$
T_5	0000	0010	1100	A = DIFF

OUT Routine

Suppose the instruction register contains the OUT instruction at the end of a fetch cycle. Then,

IR = 1110 XXXX

The instruction field goes to the CON. After this field is decoded, the CON sends out the control routine needed to load the accumulator contents into the output register.

Figure 8-7 shows the active sections of SAP-1 during the execution of an OUT instruction. Since E_A and L_O are high, the next positive clock edge loads the accumulator word into the output register during the first execution phase (T_3). The remaining execution phases (T_4 and T_5) are do nothings.

Table 8-7 shows the OUT routine. The first control word allows the accumulator to set up the output register via the W bus. The next two control words do nothing except mark time.

HLT

HLT does not require a control routine because no registers are involved in the execution of an HLT instruction. When the IR contains

IR = 1111 XXXX

the instruction field 1111 tells the CON to stop processing data. CON stops the computer by turning off the clock (circuitry discussed later).

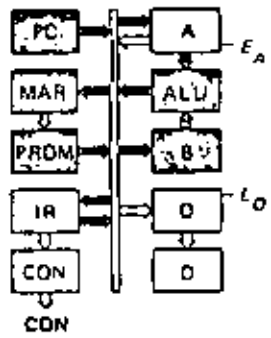


Fig. 8-7. First execution phase of OUT instruction.

TABLE 8-7. OUT ROUTINE

Phase	$C_P E_P L_M E_R$	$L_I E_I L_A E_A$	$S_U E_U L_\mu L_O$	Effect
T_3	0000	0001	0001	OUT = A
T_4	0000	0000	0000	Do nothing
T_5	0000	0000	0000	Do nothing

Machine Cycle

SAP-1 has six phases (three fetch and three execute). These six phases are called a *machine cycle* (see Fig. 8-8). It takes one machine cycle to fetch and execute each program instruction; therefore, the duration of a computer run is directly proportional to the number of program instructions.

EXAMPLE 8-6.

SAP-1 has a 1-MHz clock. How long is a machine cycle? How long is the computer run for a program with 12 instructions?

SOLUTION.

The clock has a frequency of 1 MHz; therefore, the period is

$$T = \frac{1}{1\text{ MHz}} = 1\ \mu\text{s}$$

Referring to Fig. 8-3*b*, you can see that each phase lasts for one clock period. Since there are six phases per machine cycle, each machine cycle has a duration of 6 μs.

A program with 12 instructions takes 12 machine cycles to complete. Therefore, the duration of the computer run is 72 μs.

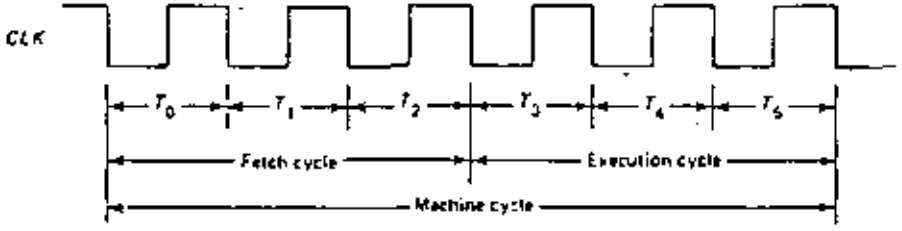


Fig. 8-8. Machine cycle.

EXAMPLE 8-7.

Figure 8-9 shows the six phases of SAP-1. The positive clock edge occurs halfway through each phase. Why is this important?

SOLUTION.

SAP-1 is a bus-organized computer (the common type nowadays). This allows its registers to communicate via a bus. But reliable loading of a register takes place only when the setup and hold time are satisfied (review Sec. 5-3 if necessary). Waiting half a cycle before loading the register satisfies the setup time; Waiting half a cycle after loading satisfies the hold time. This is why the positive clock edge is designed to strike the registers halfway through each phase (Fig. 8-9).

There's another reason for waiting half a cycle before loading a register. When the *ENABLE* input of the sending register goes high, the contents of this register are suddenly dumped onto the W bus. Stray capacitance and lead inductance prevent the bus lines from immediately reaching the correct voltage levels. In other words, we get transients (exponential, ringing, etc.) on the W bus. We have to wait for the data to settle on the bus lines if we want reliable loading. The half-cycle delay before clocking is adequate for all transients to die out.

EXAMPLE 8-8.

The following program and data are in the SAP-1 memory:

R_0 = LDA C
R_1 = ADD D
R_2 = HLT
$R_C = 15_{10}$
$R_D = 33_{10}$

Show the register changes for each phase of the R_0 machine cycle.

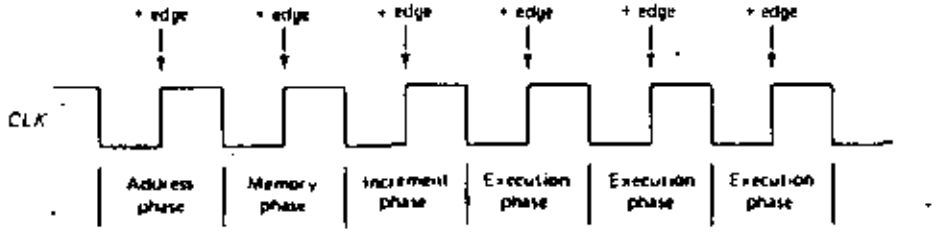


Fig. 8-9 Positive clock edges occur midway through each machine pha

SOLUTION.

Before SAP-1 can run a program, the operator has to push a *CLR* button. Among other things, this resets the program counter to 0000. The operator then pushes a start button and the computer runs the program. Here are the register changes during the first machine cycle:

T_0 (see Fig. 8-4*a*):

$$MAR = PC = 0000$$

T_1 (Fig. 8-4*b*):

$$IR = R_0 = 0000\ 1100$$

T_2 (Fig. 8-4*c*):

$$PC = 0001$$

T_3 (Fig. 8-5*a*):

$$MAR = ADDRESS = 1100$$

T_4 (Fig. 8-5*b*):

$$A = R_C = 0000\ 1111$$

T_5 (Fig. 8-5*c*):

Do nothing

EXAMPLE 8-9.

With the program and data given in the preceding example, what are the register changes during the second machine cycle?

SOLUTION.

The PC holds address 0001 at the beginning of the second machine cycle. Here are the register changes.

T_0 (Fig. 8-4*a*):

$$MAR = PC = 0001$$

SAP-1 is a computer because it stores a program and data before calculations begin; then, it automatically carries out the program instructions without human intervention. And yet, SAP-1 is primitive, like a Neanderthal man compared to a modern person. Something is missing, something found in every modern computer.

SAP-2 is the next step in the evolution toward modern computers. It includes *jump instructions*. These new instructions force the computer to repeat or skip part of a program. As you will discover, jump instructions open up a whole new world of computing power.

9-1. ARCHITECTURE

Figure 9-1 shows the architecture of SAP-2. As before, all register outputs to the W bus are three-state; those not connected to the bus are two-state. A brief description of each box is given now, with details to follow later.

Subroutine Counter

The *subroutine counter* (SC) can pinch hit for the program counter during a computer run. This occurs when a new instruction (discussed later) takes program control away from the PC and gives it to the SC. While it has control, the SC sends the binary address of the current instruction to the MAR; after the instruction has been fetched, the SC is incremented by one. Program control remains with the SC until terminated by another new instruction.

Program Counter

This time, the program counter has 8 bits; therefore, it can count from

$$PC = 0000\ 0000 \quad (0_{10})$$

to

$$PC = 1111\ 1111 \quad (FF_{16})$$

A *CLR* signal resets the PC before each computer run; so the data processing starts with the R_0 instruction.

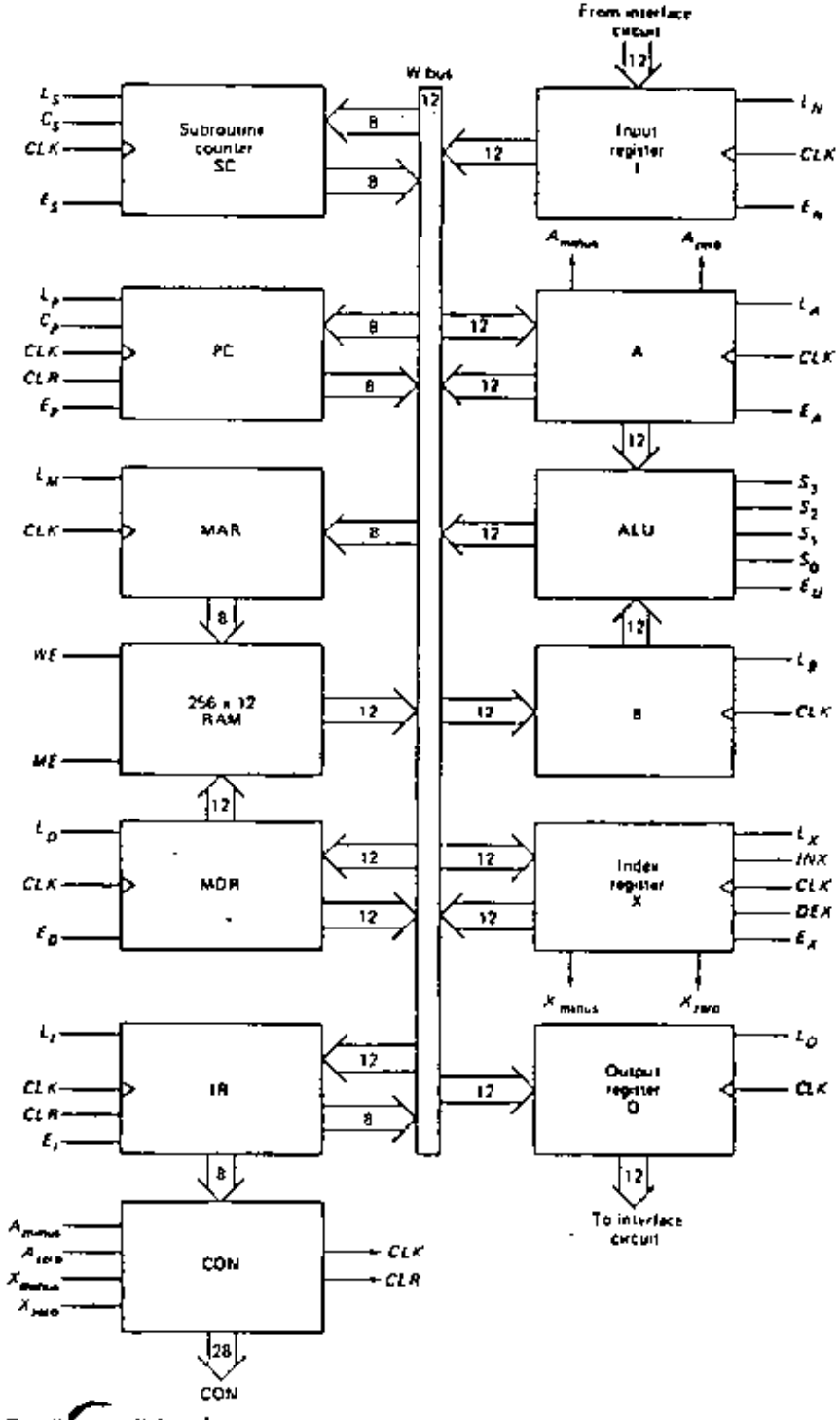


Fig. 9-4 P-2 architecture.

Memory Address Register

During the fetch cycle, the MAR receives 8-bit addresses from the program counter or subroutine counter, whichever has program control. During the execution cycle of memory-reference instructions, the MAR receives an 8-bit address field from the instruction register.

Random-Access Memory

The 256 × 12 RAM can store any 256 words of 12 bits. These words, located at addresses 0 through FF, are symbolized by R_0, R_1, R_2, . . . , R_F, R_{10}, R_{11}, R_{12}, . . . , R_{FF}. With *WE* low and *ME* high, the addressed RAM word is read onto the W bus. With *WE* and *ME* high, the contents of the MDR are written into the addressed RAM location.

Memory Data Register

A high L_D and positive clock edge load the MDR. The two-state output then sets up the RAM. The MDR also has a three-state output; therefore, a high E_D reads the contents of the MDR onto the W bus.

Instruction Register

During the execution of a memory-reference or a jump instruction, the four MSBs in the instruction register represent the instruction field and the eight LSBs are the address field. During the execution of an *operate instruction* (explained later), the eight MSBs in the IR form the instruction field and the four LSBs are don't cares.

Control Unit

The control unit produces *CLR*, *CLK*, and CON to coordinate and synchronize all registers. Because SAP-2 has a bigger instruction set, the instruction-field decoder has more gates. And because of the new registers, the control matrix has more hardware. Although the CON word is bigger, the idea is the same: The CON bits determine how the registers react to the next positive clock edge.

Input Register

Interface circuits convert signals from peripheral devices (punched-card readers, cassette-tape readers, teletypewriters, etc.) to binary signals suitable for entry into a computer. The design of an interface

depends on the peripheral: a teletypewriter (TTY) needs one kind of interface; a cassette reader, another.

In Fig. 9-1, the binary word from an interface circuit sets up the input register. Therefore, a high L_N and positive clock edge load this word into the input register. During another phase, a high E_N can read the input word onto the W bus for transfer to the accumulator. As shown in Examples 7-9 and 7-10, a control routine exists for loading up to 256 instruction and data words into the SAP-2 memory.

Accumulator

The two-state output of the accumulator goes to the ALU; the three-state output to the W bus. Therefore, the 12-bit word in the accumulator continuously drives the ALU, but this same word appears on the bus only when E_A is high. (A_{minus} and A_{zero} will be explained later.)

Arithmetic-Logic Unit

Standard ALUs are commercially available as integrated circuits. These ALUs have 4 or more select bits that determine the arithmetic or logic operation performed on words A and B.

The ALU of Fig. 9-1 is comparable to a standard ALU. Because the select bits (S_3 through S_0) cover 0000 through 1111, we can select 16 arithmetic and logic functions. SAP-2 uses three arithmetic functions (NULL, ADD, SUB) and seven logic functions (complement A, complement B, OR, AND, NOR, NAND, and XOR).

B Register

As before, a high L_B and positive clock edge load the B register with a word from the W bus, and the two-state output continuously drives the ALU.

Index Register

The *index register* (X) has a load input L_X to control loading from the bus and an enable input E_X to control readout onto the bus. Furthermore, the index register has an increment input *INX* and a decrement input *DEX*.

When *INX* is high, the next positive clock edge increases the contents of the index register by one. When *DEX* is high, the next positive clock edge decreases the contents by one. In other words, a high *INX* means the index register counts up; a high *DEX* means it counts down.[1] (X_{minus} and X_{zero} are explained later.)

Output Register

Again, the contents of the accumulator can be loaded into the output register (O). This way, an answer can move from the CPU (central processing unit) to the outside world. The output register connects to an interface circuit. The design of this circuit depends on the peripheral device being driven (printer, CRT display, teletypewriter, etc.). The interface circuit converts the contents of the output register into signals suitable for driving the peripheral device.

9-2. MEMORY-REFERENCE INSTRUCTIONS

SAP-2 has an instruction set with 28 instructions. This section is about the six memory-reference instructions.

Recognizing an MRI

The SAP-2 fetch cycle is the same as before. T_0 is the address phase (MAR = PC), T_1 is the memory phase (IR = R_{PC}), and T_2 is the increment phase (advance PC by one). All SAP-2 instructions, therefore, use the memory during the fetch cycle because a program instruction is transferred from the memory to the instruction register.

During the execution cycle, however, the memory may or may not be used; it depends on the type of instruction that has been fetched. A memory-reference instruction (MRI) is one that uses the memory during the execution cycle. You can always recognize an MRI by the registers used in the first execution phase (T_3).

Figure 9-2 shows the active parts of SAP-2 during the T_3 phase. When an MRI is in the instruction register, CON sends out high E_I and L_M bits. This means the address field of the IR is loaded into the MAR. The T_3 phase of any MRI therefore results in

MAR = ADDRESS

Remember this. It will help you distinguish an MRI from other types of instructions discussed later.

[1] If interested in the design of up/down counters, see A. P. Malvino and D. P. Leach, *Digital Principles and Applications*, McGraw-Hill Book Company, New York, 1975, pp. 246–252.

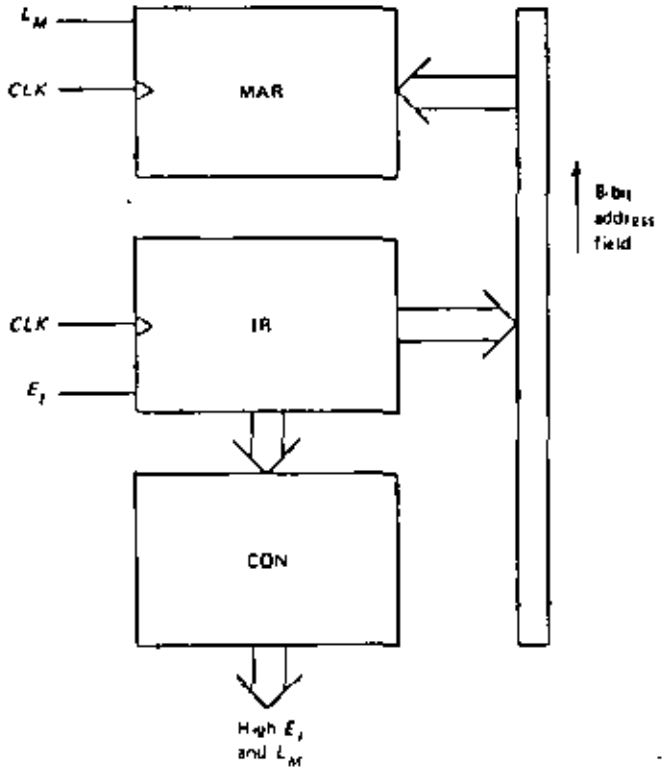


Fig. 9-2. T_3 phase of all MRIs.

LDA, ADD, and SUB

The first three MRIs in SAP-2's instruction set are the same as before, although now we can choose from 256 memory words, located at addresses 0 through FF. For instance, LDA 23 means "Load the accumulator with R_{23}," ADD 7C means "Add R_{7C} to the accumulator," and SUB FF means "Subtract R_{FF} from the accumulator."

Incidentally, any word being loaded, added, subtracted, or otherwise operated on is called an *operand*. In the foregoing examples, R_{23}, R_{7C}, and R_{FF} are operands; 23, 7C, and FF are the addresses of the operands.

STA

STA is the mnemonic for "store the accumulator word." Every STA instruction includes a memory address. STA 56, for example, means "store the accumulator word at address 56." If

$$A = 1111\ 0000\ 0000$$

the execution of STA 56 results in

$$R_{56} = 1111\ 0000\ 0000$$

LDB and LDX

The last two MRIs in SAP-2's instruction set are similar to LDA except that the addressed memory word is loaded into the B register or into the index register.

LDB is the mnemonic for "load the B register," and LDX is the mnemonic for "load the index register." To be complete, these instructions must include an address. For instance, LDB 45 means "load the B register with R_{45}," and LDX 78 means "load the index register with R_{78}."

EXAMPLE 9-1

Write a program that adds three numbers (a, b, and c) and winds up with the sum in the accumulator, the B register, and the index register.

SOLUTION.

Here's one program that will work:

R_0 = LDA 20
R_1 = ADD 21
R_2 = ADD 22
R_3 = STA 23
R_4 = LDB 23
R_5 = LDX 23
R_6 = HLT
$R_{20} = a$
$R_{21} = b$
$R_{22} = c$
R_{23} = (SUM)

Instructions R_0 through R_2 add the three numbers. Instruction R_3 stores the sum at address 23. Instructions R_4 and R_5 load the B register and index register. R_6 stops the data processing, at which point the sum appears in the accumulator, B register, and index register.

The data (a, b, and c) is loaded into addresses 20, 21, and 22 before the computer run. During the computer run, data is stored at address 23;

the parentheses around SUM indicate this data is produced by the computer.

9-3. JUMP INSTRUCTIONS

Memory-reference instructions are one type of instruction; *jump instructions* are another. SAP-2 has six jump instructions; these can change the program sequence. Instead of fetching instructions at successive addresses, the computer may repeat or skip part of the program after a jump instruction has been executed.

JMP

During the T_3 execution phase of all MRIs, the address field in the IR is transferred to the MAR (Fig. 9-2). This is how you can recognize an MRI.

Jump instructions are different. To begin with, JMP is the mnemonic for "jump the program counter." Every JMP instruction includes an address to be loaded into the PC rather than the MAR. In other words, a JMP instruction differs from an MRI because of the registers used in the T_3 execution phase (see Fig. 9-3).

When a JMP instruction is in the IR, the CON sends out high E_i and L_p bits. This means the address field of the IR goes to the PC rather than to the MAR. The T_3 phase of any JMP instruction, therefore, results in

$$\text{PC} \leftarrow \text{ADDRESS}$$

The T_4 and T_5 phases are do nothings.

Since the PC contains a *programmed* address after a JMP is executed, the instruction fetched next comes from the programmed address rather than the usual *incremented* address. This allows the computer to repeat or skip part of the program.

For instance, after the R_{33} instruction (JMP 40) is executed in Fig. 9-4*a*, the PC contains address 40. The next instruction fetched is R_{40}; therefore, the computer will repeat the program between R_{40} and R_{33}.

On the other hand, the execution of the R_{51} instruction (JMP 70) in Fig. 9-4*b* forces the computer to jump to the R_{70} instruction; therefore, all instructions between R_{51} and R_{70} are skipped.

Examples 9-2 onward give you practice with the JMP instruction and sharpen your understanding of this powerful instruction.

JAM

JAM resembles JMP because it can change the contents of the PC. But it differs from JMP because it produces a program jump only when a special condition is satisfied.

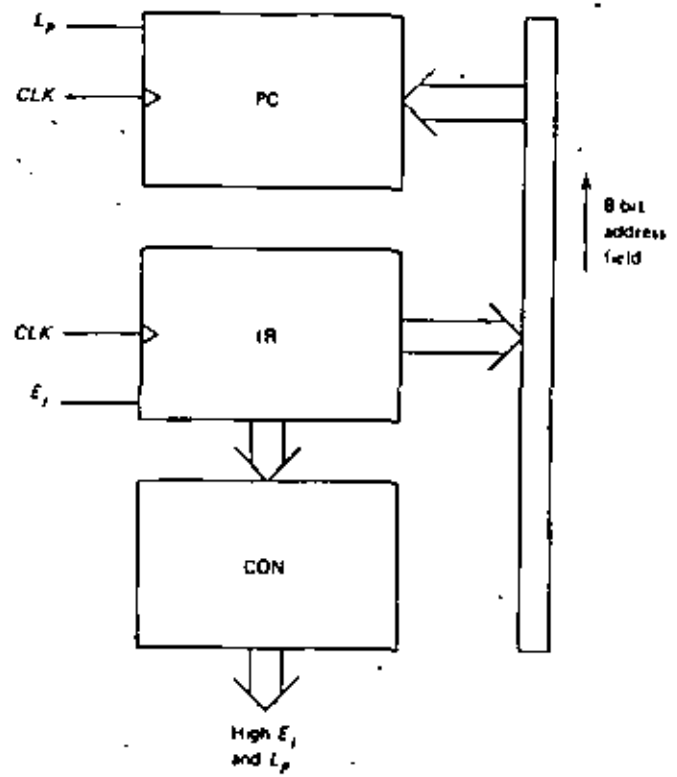


Fig. 9-3. T_3 phase of JMP instruction.

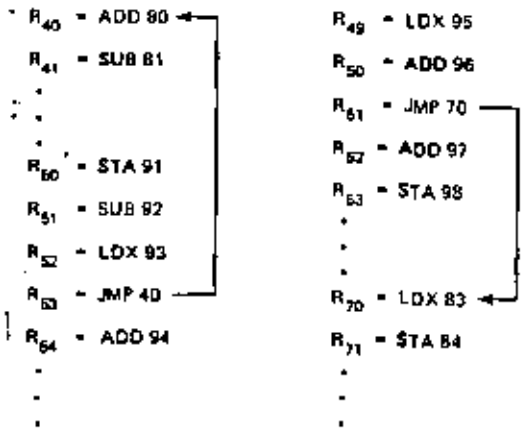


Fig 9-4. (*a*) Jumping backward (*b*) Jumping forward.

JAM is the mnemonic for "jump the program counter if the accumulator is minus"; or simply, "jump if accumulator minus." Every JAM instruction includes the address to be loaded into the PC. For instance, JAM 40 means "jump to address 40 if accumulator minus," and JAM 60 means "jump to address 60 if accumulator minus."

Figure 9-5 shows the effect of a JAM 40. During the execution of the R_{58} instruction, CON tests the contents of the accumulator. If the accumulator is minus, the program jumps back to the R_{40} instruction; otherwise, the program continues with the R_{51} instruction.

After the SUB 92 is executed, a JAM 60 appears. Again, the CON tests the accumulator contents. If these contents are minus, the program jumps ahead to the R_{60} instruction; otherwise, it continues with the R_{53} instruction.

JAZ

JAZ is the mnemonic for "jump the program counter if the accumulator is zero"; or simply, "jump if accumulator zero." Every JAZ instruction includes the address to be loaded into the PC. JAZ 32 means "jump to address 32 if accumulator zero, otherwise, continue with the next instruction."

JIM and JIZ

JIM stands for "jump the program counter if the index register is minus"; or simply, "jump if index minus." JIZ is the mnemonic for "jump the program counter if the index register is zero"; or simply, "jump if index zero."

Every JIM and JIZ instruction includes the address to be loaded into the PC. For instance, JIM 94 means "jump to address 94 if index minus"; JIZ 7F means "jump to address 7F if index zero."

JAM, JAZ, JIM, and JIZ are called *conditional jumps* because the program jump occurs only if certain conditions are satisfied. On the other hand, JMP is *unconditional;* once this instruction is fetched, the execution cycle always jumps the program to the specified address.

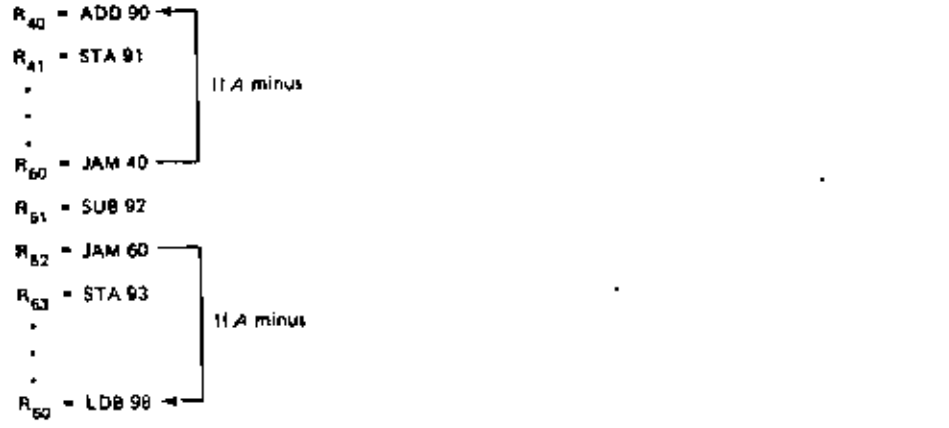


Fig. 9-5 JAM jumps.

JMS

A *subroutine* is a program stored in the memory for possible use in another program. Most computers have subroutines for finding sines, cosines, tangents, logs, square roots, etc. These subroutines are part of the *software* supplied with modern computers. ("Software" refers to the programs manufacturers prepare for use with computers. Besides subroutines, software includes *assemblers* and *compilers*, programs that can translate source programs into object programs.)

JMS is the mnemonic for "jump to subroutine." Every JMS instruction must include the *starting address* of the desired subroutine. For instance, if a square-root subroutine starts at address A2 and a log subroutine at address C5, then a JMS A2 will jump to the square-root subroutine and a JMS C5 to the log subroutine. After the subroutine is completed, the computer returns to the original program.

When a JMS is executed, program control passes from the program counter to the subroutine counter. After each subroutine instruction is fetched, the SC is incremented by one. In this way, the SC pinch hits for the program counter during the execution of a subroutine. At the end of the subroutine, program control reverts to the PC.

JMS is unconditional, like JMP. Once a JMS has been fetched into the instruction register, the computer will jump to the starting address of the subroutine.

Summary

During the execution of an MRI, the MAR receives the address field.

During the execution of a JMP, the program counter receives the address field. During the execution of JAM, JAZ, JIM, or JIZ, the program counter receives the address field if certain conditions are satisfied. During the execution of a JMS, the subroutine counter receives the address field and the program counter *saves* (retains) the incremented address.

MRIs are the workhorses of an instruction set; they load registers, add and subtract addressed words, and store words in the memory. But the jump instructions are the real magic of an instruction set; they make the computer repeat part of a program, skip part of a program, and jump to other addresses in memory where useful subroutines are stored.

Table 9-1 lists the mnemonics and op-code for SAP-2's memory-reference and jump-instructions.

TABLE 9-1. MRIs AND JUMP INSTRUCTIONS FOR SAP-2

Mnemonic	*Op-Code*	*Meaning*
LDA	0000	Load accumulator
ADD	0001	Add addressed word
SUB	0010	Subtract addressed word
STA	0011	Store accumulator word
LDB	0100	Load B register
LDX	0101	Load index register
JMP	0110	Jump
JAM	0111	Jump if accumulator minus
JAZ	1000	Jump if accumulator zero
JIM	1001	Jump if index minus
JIZ	1010	Jump if index zero
JMS	1011	Jump to subroutine
	1100	Not used
	1101	Not used
	1110	Not used
OPR	1111	Operate instruction (Sec. 9-4)

EXAMPLE 9-2.

Describe what happens during the execution of this program:

R_0 = LDA 6
R_1 = SUB 7
R_2 = JAM 5
R_3 = JAZ 5
R_4 = JMP 1
R_5 = HLT
R_6 = 25_{10}
R_7 = 9_{10}

Because of the jump instructions, part of the program will be repeated. We can summarize the data processing by listing a comment after each instruction that is executed.

First pass:

Instruction	*Comment*
R_0 = LDA 6	A = 25_{10}
R_1 = SUB 7	A = 16_{10}
R_2 = JAM 5	No jump
R_3 = JAZ 5	No jump
R_4 = JMP 1	Return to R_1

During this first pass, the LDA 6 loads the accumulator with the binary equivalent of decimal 25. The SUB 7 subtracts the equivalent of decimal 9, leaving decimal 16 in the accumulator. Because the accumulator contents are positive, the JAM 5 and JAZ 5 have no effect. Then comes the JMP 1, which returns the processing to the SUB 7 instruction (see Fig. 9-6*a*).

Second pass:

Instruction	*Comment*
R_1 = SUB 7	A = 7_{10}
R_2 = JAM 5	No jump
R_3 = JAZ 5	No jump
R_4 = JMP 1	Return to R_1

On the second pass, the SUB 7 reduces the accumulator contents to decimal 7, still positive. The JAM 5 and JAZ 5 are again ignored. The JMP 1 returns the processing to the SUB 7 instruction (Fig. 9-6*a*).

Third pass:

Instruction	*Comment*
R_1 = SUB 7	A = -2_{10}
R_2 = JAM 5	Jump to R_5
R_5 = HLT	End of program

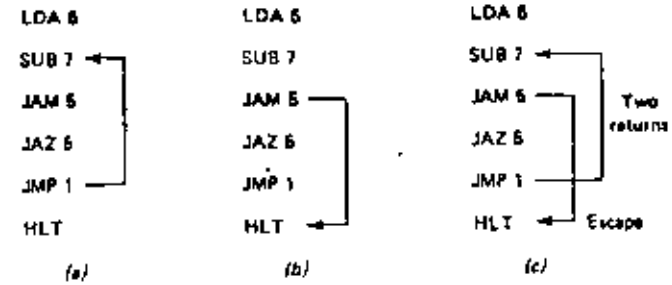


Fig. 9-6. A sample program with JAM, JAZ, and JMP.

On the third pass, the SUB 7 makes the accumulator contents negative (the sign bit changes from 0 to 1). The JAM 5 now forces the computer to jump to the R_5 instruction. At this point, the data processing stops (Fig. 9-6*b*).

The computer run therefore has two returns to SUB 7 and the final escape to HLT (Fig. 9-6*c*).

EXAMPLE 9-3.

A square-root subroutine starts at address A2 and a log subroutine at address C5. What does the following program do?

R_0 = LDA 6
R_1 = JMS A2
R_2 = ADD 7
R_3 = JMS C5
R_4 = STA 8
R_5 = HLT
$R_6 = 400_{10}$
$R_7 = 80_{10}$
R_8 = (ANSWER)

SOLUTION.

We can summarize the data processing by listing a comment after each instruction is executed.

Instruction	*Comment*
R_0 = LDA 6	A = 400_{10}
R_1 = JMS A2	A = 20_{10} after subroutine
R_2 = ADD 7	A = 100_{10}
R_3 = JMS C5	A = 2_{10} after subroutine
R_4 = STA 8	$R_8 = 2_{10}$
R_5 = HLT	End of program

The LDA 6 loads the accumulator with 400_{10}. The JMS A2 jumps the computer to the square-root subroutine. After this subroutine is finished, the accumulator contains 20_{10}. The ADD 7 increases the accumulator contents to 100_{10}. The JMS C5 results in a log subroutine, after which the accumulator contains 2_{10} (the common logarithm of 100_{10}). The STA 8 stores the accumulator word at address 8.

Figure 9-7 illustrates the program sequence. As shown, the first exit from the main program occurs after the execution of JMS A2. After completion of the square-root subroutine, we reenter the main program at ADD 7. After JMS C5 is executed, we leave the main program for a second time. Upon completion of the log subroutine, we reenter the main program at STA 8.

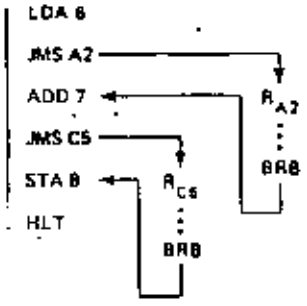


Fig. 9-7. Program sequence for subroutines.

Notice the BRB instruction at the end of each subroutine. BRB is the mnemonic for "branch back." This instruction takes program control away from the subroutine counter and returns it to the program counter. BRB is discussed further in the next section.

9-4. OPERATE INSTRUCTIONS

An *operate instruction* neither uses the memory nor alters the program counter. Instead, it operates on words already transferred out of memory into working registers (like the accumulator, B register, index register, etc.). What follows is a brief description of each SAP-2 operate instruction.

NOP

NOP is the mnemonic for "no operation." During the execution of a NOP instruction, all the phases are do nothings. Therefore, nothing happens when a NOP is executed.

Why use a NOP? After writing a long program, you almost always have to *debug* it (locate and correct the errors). This may require adding, changing, or removing instructions. If you remove an instruction, you can fill the gap with a NOP.

For instance, in Fig. 9-8*a*, the second ADD E2 is an unintended duplication. If you pull this instruction, a gap appears in the program. The easiest way to fill the gap is with a NOP (Fig. 9-8*b*).

Some microcomputer manufacturers recommend starting a program with a NOP. The reason given is that potential races exist at the be-

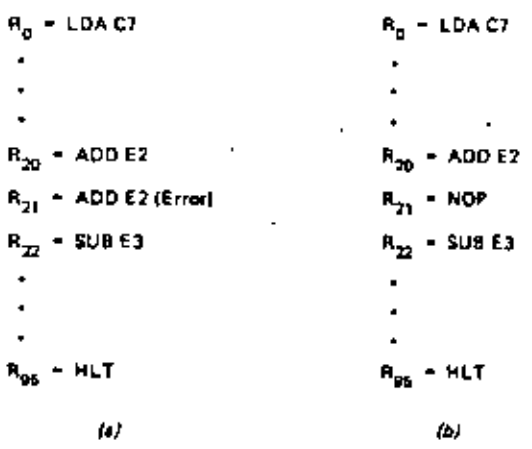


Fig. 9-8. Using a NOP to fill a gap.

ginning of a program. By leading off with a NOP, these potential races have a chance to die out.

NOPs are also used to produce delays. By repeating a NOP a few hundred times, you can delay the data processing while some activity takes place elsewhere in a computerized system.

CLA

CLA means "clear the accumulator." The execution of a CLA resets all accumulator bits to zero (Fig. 9-9*a*).

XCH

XCH is the mnemonic for "exchange accumulator and index." During the execution of an XCH, the words in the accumulator and index register are transposed (Fig. 9-9*b*). For instance, if

$$A = 0000\ 0000\ 1111 \quad (15_{10})$$
$$X = 0000\ 0000\ 1001 \quad (9_{10})$$

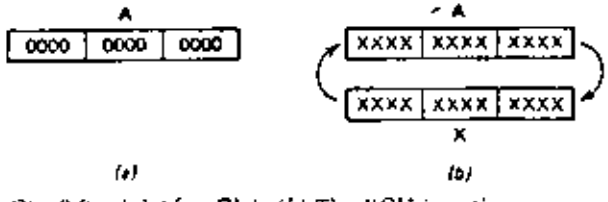


Fig. 9-9. (*a*) After CLA. (*b*) The XCH in action.

the execution of an XCH results in

$$A = 0000\ 0000\ 1001 \quad (9_{10})$$
$$X = 0000\ 0000\ 1111 \quad (15_{10})$$

DEX

DEX means "decrement the index register." The execution of a DEX decreases the contents of the index register by one. Given

$$X = 0000\ 0000\ 1001 \quad (9_{10})$$

the execution of a DEX produces

$$X = 0000\ 0000\ 1000 \quad (8_{10})$$

INX

INX is the mnemonic for "increment the index register." This instruction adds one to the index register.

CMA

CMA stands for "complement the accumulator." The execution of a CMA inverts each bit in the accumulator, producing the 1's complement.

CMB

CMB is the mnemonic for "complement the B register." This instruction inverts each bit in the B register, resulting in the 1's complement.

IOR

IOR means "inclusive OR," identical to the OR function discussed in earlier chapters. The execution of an IOR will OR the corresponding bits in the accumulator and B register (Fig. 9-10); the result appears in the accumulator. If

$$A = 1111\ 1110\ 1100 \quad (9\text{-}1)$$
$$B = 1111\ 0001\ 0000 \quad (9\text{-}2)$$

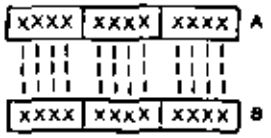


Fig. 9-10 Logic operations are bitwise.

the execution of an IOR gives

A = 1111 1111 1100

AND

Executing an AND produces bitwise ANDing of the accumulator and B register (Fig. 9-10), with the result appearing in the accumulator. With the initial values given by Eqs. (9-1) and (9-2), the execution of an AND produces

A = 1111 0000 0000

NOR

This instruction NORs the contents of the accumulator and B register on a bitwise basis. As usual, the result appears in the accumulator. With the values of Eqs. (9-1) and (9-2), the execution of a NOR gives

A = 0000 0000 0011

NAN

NAN is the mnemonic for NAND. Again, the corresponding bits of the accumulator and B register are operated on. Given the values of Eqs. (9-1) and (9-2), the execution of a NAN results in

A = 0000 1111 1111

XOR

An XOR instruction will exclusive OR the contents of the accumulator and B register bit by bit. Given the values of Eqs. (9-1) and (9-2), the execution of an XOR produces

A = 0000 1111 1100

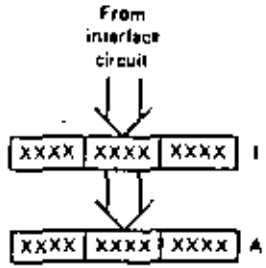


Fig. 9-11. INP transfers input word from interface circuit to accumulator.

BRB

BRB is the mnemonic for "branch back." The BRB is used at the end of each subroutine to get back to the main program. BRB is to a subroutine as HLT is to the main program. In other words, if you forget to use a BRB at the end of a subroutine, you will get computer trash when the subroutine is called.

INP

INP means "input." This instruction is executed in two phases. The first execution phase loads the input register with a word from the interface circuit. The second execution phase transfers this word to the accumulator (see Fig. 9-11).

OUT

OUT stands for "output." When this instruction is executed, the accumulator word is loaded into the output register.

HLT

What does every program end with? Right, HLT. This instruction stops the data processing.

Op-Code and Select Code

Earlier, Table 9-1 showed the op-code for all MRIs and jump instructions. 1100, 1101, and 1110 are unused words, but 1111 indicates an operate instruction. That is, all operate instructions have this format:

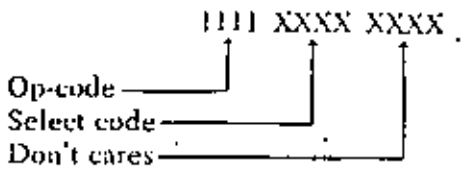


Table 9-2 shows the *select code*, the middle 4 bits used for each operate instruction. To program any operate instruction, start with four 1s, follow with the select code, and end with four don't cares. For instance, the mnemonics and equivalent machine words for the first three operate instructions are

NOP = 1111 0000 XXXX
CLA = 1111 0001 XXXX
XCH = 1111 0010 XXXX

(The = sign stands for "is equivalent to.")

Incidentally, MRIs and jump instructions use an address field but operate instructions don't, because all the operands are already in working registers.

You now have the entire SAP-2 instruction set: 6 MRIs, 6 jumps, and 16 operates; a total of 28 instructions.

TABLE 9-2. OPERATE INSTRUCTIONS

Mnemonic	*Select Code*	*Meaning*
NOP	0000	No operation
CLA	0001	Clear accumulator
XCH	0010	Exchange A and index
DEX	0011	Decrement index
INX	0100	Increment index
CMA	0101	Complement A
CMB	0110	Complement B
IOR	0111	Inclusive-OR
AND	1000	AND
NOR	1001	NOR
NAN	1010	NAND
XOR	1011	Exclusive-OR
BRB	1100	Branch back
INP	1101	Input
OUT	1110	Output
HLT	1111	Halt

EXAMPLE 9-4.

How many times is the DEX executed before the JIZ 4 jumps the processing to the HLT in the following program?

R_0 = LDX 5
R_1 = DEX
R_2 = JIZ 4
R_3 = JMP 1
R_4 = HLT
$R_5 = 3_{10}$

SOLUTION.

The LDX 5 loads the index register (hereafter called index) with 3_{10}. DEX reduces the index to 2_{10}. JIZ 4 is ignored because the index is not zero. JMP 1 returns the program to the DEX.

The second time the DEX is executed, the index drops to 1_{10}. JIZ 4 is again ignored and JMP 1 returns the program to DEX for a second time (see Fig. 9-12).

The third DEX reduces the index to zero. This time, JIZ 4 jumps the program to HLT.

A *loop* is part of a program that is repeated. In this example, we have *passed* through the loop (DEX and JIZ) three times as shown in Fig. 9-12. Note that the number of passes through the loop equals the number loaded into the index register.

If we change R_5 to 7_{10} and rerun the program, the computer will pass through the loop seven times. Similarly, if we change R_5 to 200_{10}, the computer will pass through the loop 200 times.

The point to remember is this. We can set up a loop by including an LDX, DEX, JIZ, and JMP in a program. The integer loaded into the

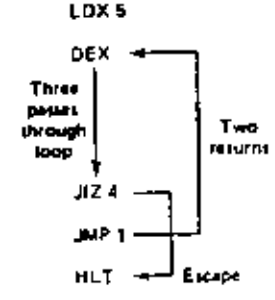


Fig. 9-12. Programming a loop.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

MODOS DE DIRECCIONAMIENTO

ING. LUIS CORDERO BORBOA

JUNIO, 1980

III. - MODOS DE DIRECCIONAMIENTO

1.- ESQUEMAS DE DIRECCIONAMIENTO.

La unidad central de proceso (CPU) en las computadoras debe realizar las siguientes funciones:

- Obtener y traer de memoria primaria al CPU la siguiente instrucción a ejecutar.
- Entender los operandos, esto es, definir la localización de los operandos necesarios para ejecutar la instrucción y traerlos al CPU.
- Ejecutar la instrucción.

Para llevar a cabo las funciones anteriores el CPU debe contar con la siguiente información:

- El código de operación de la instrucción a ejecutar.
- Las direcciones de los operandos y la del resultado.
- La dirección de la siguiente instrucción a ejecutar.

Existen diferentes soluciones que satisfacen los requerimientos anteriores, los cuales determinan la arquitectura de los procesadores que las utilizan.

Se supondrán operaciones aritméticas en las que se tienen dos operandos y un resultado ya que son las que proporcionan el caso más general.

a) Máquinas de "3+1" direcciones

El formato de instrucción en este esquema de direcciona-- miento contiene todos los elementos necesitados por el CPU

para realizar sus funciones.

Un posible formato de instrucción se muestra en la figura III.1

CODIGO DE OPERAC.	DIRECCION PRIMER OPERANDO	DIRECCION SEGUNDO OPERANDO	DIRECCION RESULTADO	DIRECCION DE LA SIGUIENTE INSTRUCCION

Palabra n de memoria

FIG. III.1

En este caso se tienen cinco campos en el formato de instrucción: Uno para el código de operación que sirve para indicar el tipo de opera---ción a realizar (suma, resta, multiplicación, etc.), tres campos para las direcciones de los operandos y resultado de las operaciones, un campo para indicar la dirección de la siguiente instrucción a ejecutar.

Las instrucciones para ésta máquina podrían ser escritas en forma simbólica en la siguiente forma: ADD A, B, C, D donde ADD representa el código de operación suma y A, B, C y D son nombres simbólicos asignados a localidades de memoria.

Suponiendo que existen las instrucciones suma (ADD), substracción---(SUB) y multiplicación (MUL), entonces una posible traducción de la expresión A=(B*C)-(D*E) en FORTRAN a lenguaje simbólico en la máquina de 3+1 direcciones sería:

L1: MUL B, C, T1, L3

L3: MUL D, E, T2, L7

L7: SUB T2, T1, A, L8

L8: Siguiente instrucción

donde T1 y T2 representan localidades temporales usadas para guardar resultados aritméticos intermedios.

Las conclusiones más importantes en este esquema son:
Los programas no necesitan estar almacenados en memoria en forma secuencial ya que el campo de dirección de la siguiente instrucción permite conocer donde fueron almacenados.

Debido a que cada instrucción contiene en forma explícita tres direcciones, no es necesario tener en el CPU hardware para guardar los resultados de las operaciones.

b) Máquinas de "3" direcciones

Considerando que los programas se escriben secuencialmente y que por consiguiente es muy lógico almacenarlos en este mismo orden, se llega a un nuevo esquema de direccionamiento en el cual se sustituyen todos los campos de dirección de la siguiente instrucción por un solo registro dentro del procesador que lleva en forma secuencial y automáticamente la dirección de la siguiente instrucción a ejecutar. Un posible formato de instrucción se muestra en la fig. III.2 .

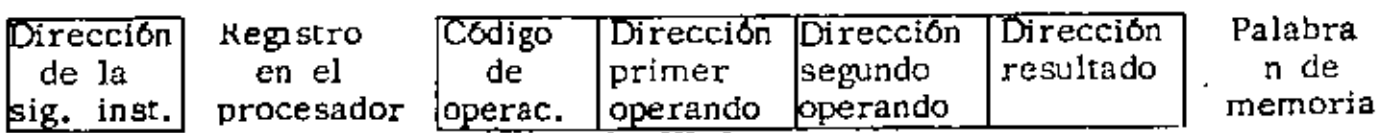


FIG. III.2

Utilizando este esquema de direccionamiento la expresión A=(B*C)-(D*E) en FORTRAN, quedaría expresada como:

```
MUL   B, C, T1
MUL   D, E, T2
SUB   T2, T1, A
Siguiente instrucción
```

Donde se ha suprimido la dirección de la siguiente instrucción ya que ésta es llevada en forma secuencial y automática por un registro del procesador conocido como contador del programa (PC).

Con el esquema de 3 direcciones se logra aprovechar la memoria en forma más eficiente y reducir la longitud de palabra lo que redunda directamente en los costos de la misma.

c) Máquinas de "2" direcciones.

En las operaciones aritméticas no siempre es necesario guardar el resultado en una localidad de memoria y preservar los operandos, por lo que se puede pensar en utilizar uno de ellos para---- guardar el resultado una vez que la operación se ha efectuado. Las consideraciones anteriores llevan a presentar un posible formato de instrucción en esta máquina, mostrado en la figura III.3

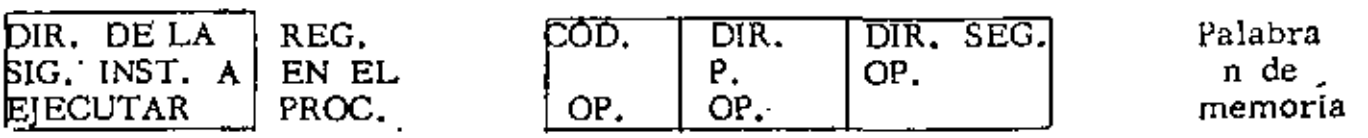


FIG. III.3

En este esquema se usará la dirección del segundo operando como la dirección del resultado una vez que la operación se haya efectuado, por lo que el segundo operando será destruído. Así pues la expresión A=(B*C)-(D*E) en FORTRAN, quedaría:

```
MUL   B, C
MUL   D, E
SUB   E, C
ADD   A, C
```

La eliminación del campo de dirección del resultado permite reducir la longitud de la palabra de memoria y los costos de la misma, lo que permite usar este esquema en máquinas medianas y chicas.

d) Máquinas de "1" dirección

Este esquema de direccionamiento permite eliminar de todas las instrucciones el campo de dirección de uno de los operando y sustituirlo por un registro dentro del procesador, el cual contendrá a uno de los operandos. A este registro se le conoce como acumulador.- El formato de instrucción para la máquina de 1 dirección se muestra en la figura III.4

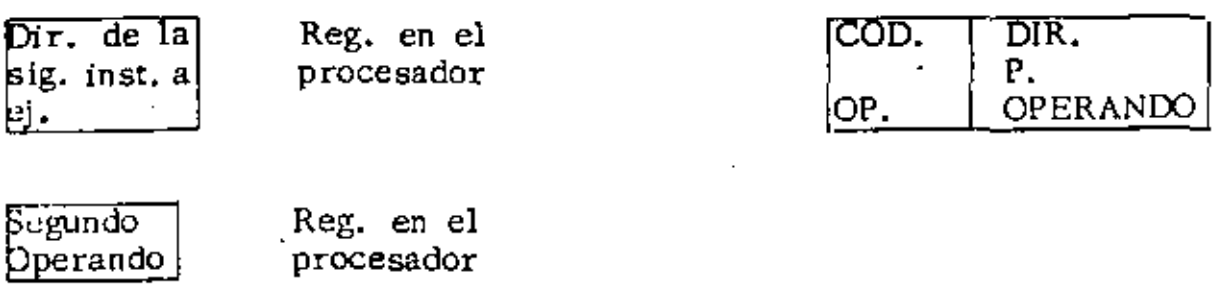


FIG. III.4

Lo anterior implica la creación de instrucciones que permitan cargar el acumulador con el segundo operando (LAC) y depositar el contenido del acumulador en memoria (DAC).

Es importante hacer notar que todas las operaciones se llevan a cabo implícitamente contra el acumulador y que éste contendrá el resultado de la operación efectuada. La expresión A=(B*C)-(D*E) en FORTRAN, podría traducirse a:

```
LAC   D
MUL   E
DAC   T1
LAC   B
MUL   C
SUB   T1
DAC   A
```

Este esquema de direccionamiento ha sido ampliamente implementado en una gran mayoría de las minicomputadoras, como por ejemplo: PDP-8, PDP-15, IBM-1130, IBM-7090 y CDC 3600.

e) Máquinas de "0" direcciones

Este esquema de direccionamiento solo utiliza el campo de código de operación, por lo que es necesario contar con algún mecanismo que implícitamente permita conocer los operandos.

El mecanismo anterior se implementa usando una pila ó stack, el cual se puede pensar como un conjunto de localidades contiguas de

memoria accesadas usando una disciplina UEPS (últimas entradas, primeras salidas). De lo anterior se concluye que en cada momento se tendrá disponible el elemento que se encuentre en el tope del stack.

El formato de instrucción para este esquema de direccionamiento se encuentra en la figura III.5

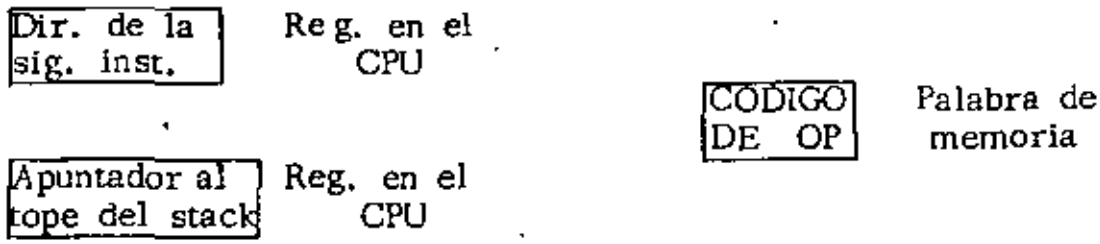
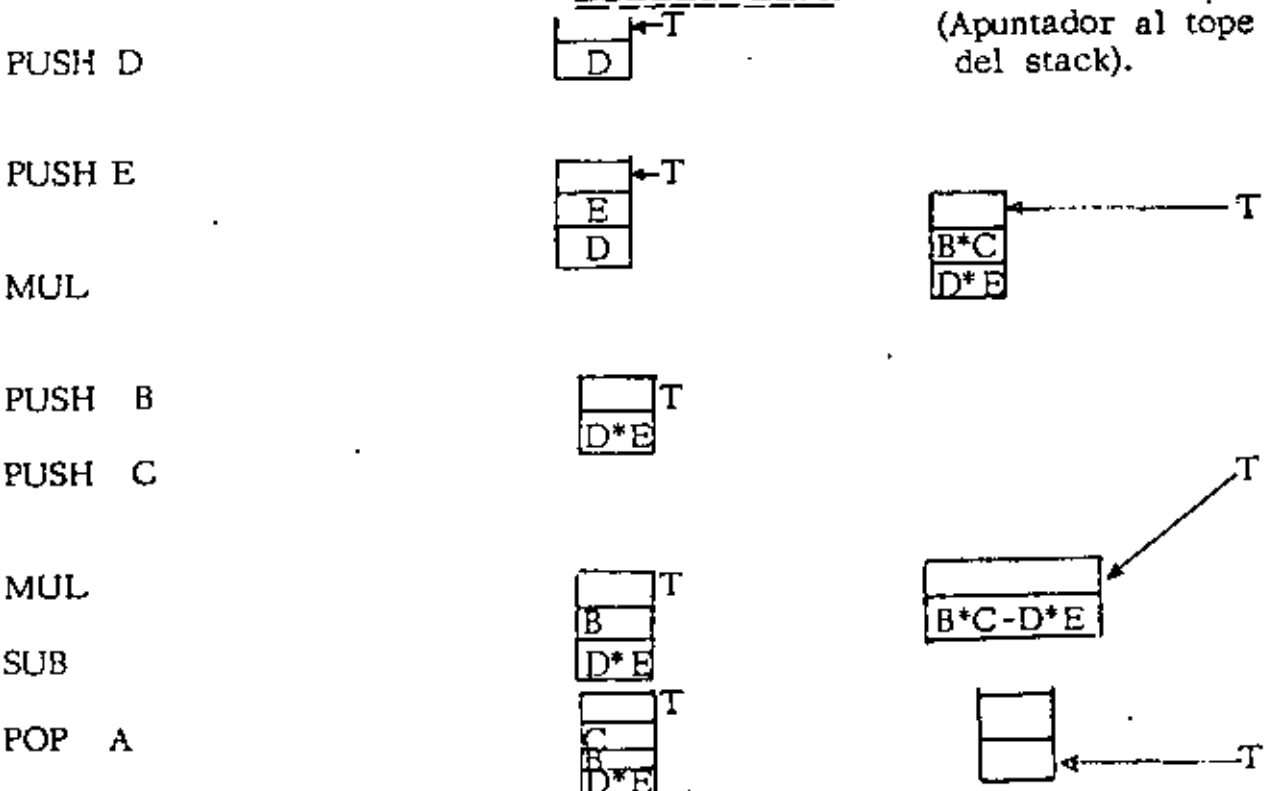


FIG. III.5

Es necesario contar con instrucciones que permitan meter elementos de memoria al stack (PUSH) y sacar elementos del stack a memoria (POP).

La expresión A=(B*C)-(D*E) en FORTRAN, podría expresarse como:

FIG. III.6



En la fig. III.6 se ilustra el estado del stack después de cada una de las inst. anteriores.

Se puede concluir que el conjunto de instrucciones de la máquina no está formado solamente por instrucciones de cero direcciones ya que también se requieren instrucciones de una dirección para meter y sacar elementos al stack.

Se requiere un registro en el procesador que apunte al tope del stack y se elimine el acumulador ya que el resultado de las operaciones -- también quedará en el stack.

2.- METODOS DE DIRECCIONAMIENTO

En las máquinas de una sola dirección el formato de las instrucciones que hace referencia a memoria consta de dos campos: el campo de código de operación y el campo de dirección del operando. Si suponemos que el campo de dirección consta de n bits, entonces la máxima capacidad de memoria direccionable será 2^n localidades. Lo anterior puede resultar bastante drástico en el caso de las minicomputadoras ya que por lo general tienen palabras de 12 ó 16 bits y si se asignan cuatro de ellos al campo de código de operación solo se pueden direccionar $2^8 = 256$ localidades de memoria en el caso de palabras de 12 bits ó $2^{12} = 4096$ localidades de memoria en el caso de palabras de 16 bits, lo cual resulta insuficiente para la gran mayoría de las aplicaciones.

Lo anterior ha ocasionado diferentes modos de direccionamiento, en los cuales el campo de dirección sirve para calcular la dirección efectiva del operando, logrando una mayor capacidad de memoria direccionable.

a) Inmediato

En este caso el operando puede estar contenido directamente en el campo de dirección ó en la localidad de memoria siguiente a la instrucción.

Será necesario dedicar un bit de la palabra para saber como se debe interpretar la instrucción.

b) Directo

Existe direccionamiento directo cuando el campo de dirección de la instrucción contiene la dirección del operando ó cuando éste campo combinado con algún registro ó palabra de memoria generan la dirección del operando.

b.1) Usando página cero

Uno de los esquemas más comunes de organización de memoria, divide ésta en n páginas de longitud fija, donde n dependerá del tamaño de la memoria y del tamaño de las páginas.

Las máquinas que usan estos esquemas generalmente usan la página cero con propósitos especiales, como son: manejo de interrupciones, traps, localidades autoincrementables, etc.

La forma de indicar si el contenido del campo de dirección se refiere a la página cero, es usando un bit para este propósito, p. ej. si este bit es cero el campo de dirección apunta a una localidad en la página cero.

b.2) Usando página actual

Si el bit de página está en uno, se asume que el campo de dirección apunta a una localidad en la página en la que se encuentra la instrucción. A esta página se le conoce como

página actual.

La dirección del operando se determina sumando los bits de orden superior del PC al campo de dirección de la instrucción.

b.3) Relativo al PC

En este modo de direccionamiento el contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al PC para obtener la dirección del operando.

b.4) Relativo a un registro índice

El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al contenido de un registro índice para obtener la dirección del operando. En caso de existir más de un registro índice es preciso asignar los bits necesarios para su identificación.

c) Indirecto

En el direccionamiento indirecto el campo de dirección de la instrucción contiene un apuntador a la dirección del operando ó este campo combinado con algún registro ó palabra de memoria genera un apuntador a la dirección del operando.

Mediante un bit en la instrucción se puede saber si el direccionamiento usado es directo ó indirecto.

c.1) Usando página cero

El campo de dirección de la instrucción apunta a una localidad en la página cero. A su vez ésta localidad contiene la dirección del operando.

c.2) Usando página actual

El campo de dirección de la instrucción apunta a una localidad en la página actual. Esta localidad contiene la dirección del operando.

c.3) Relativo al PC

El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al PC para obtener la dirección del apuntador al operando.

c.4) El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al contenido de un registro índice para obtener la dirección del apuntador al operando.

La combinación de todos los métodos de direccionamiento anteriores con registros de propósito general, permiten lograr modos de direccionamiento bastante poderosos. Cuando se usan los registros de propósito general, el campo de dirección de la instrucción especffica que registro se usa y como se interpreta la información que contiene.

3.- DIRECCIONAMIENTO EN PDP-11

a) Con dos operandos

La computadora PDP-11 es una máquina de dos direcciones por lo que su formato de instrucción tiene campos para código de operación y operandos. Lo anterior se observa en la fig. III.7

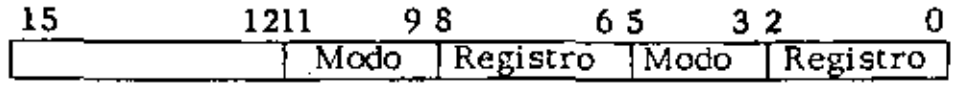


FIG. III.7

Los bits 12-15 contienen el código de operación

Los bits 6-11 contienen la dir. fuente

Los bits 0- 5 contienen la dir. destino

Las direcciones fuente y destino serán utilizadas para el cálculo de la dirección efectiva de los operandos, interpretando el modo y el registro usados.

La dirección fuente contiene dos subcampos de 3 bits cada uno, de esta forma es posible indicar cual de los ocho registros de propósito general será usado, así como la interpretación que se le dará de acuerdo a los ocho modos de direccionamiento.

El modo y registro en la dir destino se entienden en la misma forma que en la dir fuente. La dir destino también será usada para almacenar el resultado de la operación una vez que esta se haya efectuado.

b) En esta máquina existen instrucciones que solo requieren un operando en cuyo caso se utiliza un formato de instrucción con campos de código de operación y dirección destino, según se muestra en la fig. III. 8

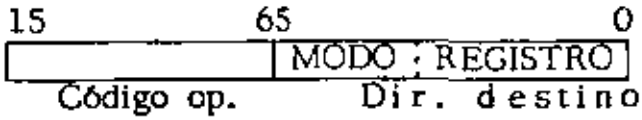


FIG. III.8

La interprelación dada a la dirección fuente es la misma que en el caso de dos operandos.

Para poder ejemplificar los modos de direccionamiento se usará el siguiente conjunto de instrucciones; así mismo se asumirá que todos los números están en octal:

Mnemonico	Código Octal	Descripción
CLR	0050DD 1050DD	Limpia (pone a ceros el destino).
INC INCB	0052DD 1052DD	Incremento (suma uno al contenido del destino)
COM COMB	0051DD 1051DD	Complementa lógicamente el destino
ADD	06SSDD	Suma

c) Direccionamiento directo

Existen cuatro modos usados en direccionamiento directo, los cuales se explican a continuación:

c.1) Registro

Forma general: OPR Rn

Descripción: El registro especificado contiene el operando requerido por la instrucción.

OPR representa un código de operación en forma general.

Modo: 0

Ejemplos: 1

c.2) Autoincremento

Forma general: OPR (Rn)+

Descripción: El contenido del registro es incrementado después de ser usado como apuntador al operando. Si la instrucción es de palabra se autoincremente en dos y si es de byte en uno.

Modo: 2

Ejemplos: 2

c.3) Autodecremento

Forma general: OPR -(Rn)

Descripción: El contenido del registro es decrementado antes de ser usado como apuntador al operando. Si la instrucción es de palabra se autodecrementa en dos y si es de byte en uno.

Modo: 4

Ejemplos: 3

c.4) Indice

Forma general: OPR X(Rn)

Descripción: La suma de X y el contenido del registro se utiliza como la dirección del operando.

Modo: 6

Ejemplos: 4

d) Direccionamiento indirecto

Existen 4 modos de direccionar en forma indirecta, los cuales utilizan los modos básicos (direccionamiento directo) en forma diferida.

d.1) Registro diferido

Forma general: OPR @Rn

Descripción: El registro contiene la dirección del operando.

Modo: 1

Ejemplos: 5

d.2) Autoincremento diferido

Forma general: OPR @(Rn)+

Descripción: El contenido del registro es incrementado después de ser usado como apuntador a la dirección del operando. El autoincremento será en dos, tanto para instrucciones de byte como de palabra.

Modo: 3

Ejemplos: 6

d.3) Autodecremento diferido

Forma general: OPR @-(Rn)

Descripción: El contenido del registro es decrementado antes de ser usado como apuntador a la dirección del operando. El autodecremento será en dos, tanto para instrucciones de byte como de palabra.

Modo: 5

Ejemplos: 7

d.4) Indice diferido

Forma general: OPR @X(Rn)

Descripción: La suma de X y el contenido del registro se utiliza como apuntador a la dirección del operando. La palabra de índice X está almacenada en la localidad de memoria siguiente a la instrucción.

El valor de Rn y X no se modifica.

Modo: 7

Ejemplos: 8

e) Uso del PC en direccionamiento

El registro siete, tiene el propósito específico de servir como contador de programa (PC), por lo cual cada vez que el procesador

usa el R7 para traer una palabra de memoria, el R7 se incrementa automáticamente en dos de tal forma que siempre apunta a la siguiente instrucción a ejecutar ó a la siguiente palabra de la instrucción que actualmente se está ejecutando.

Lo anterior permite usar el PC con propósitos de direccionamiento, permitiendo lograr ventajas cuando se utiliza con alguno de los modos 2, 3, 6 ó 7.

e.1) Inmediato

Forma general: OPR#n, DD

Descripción: El operando está en la localidad de memoria siguiente a la instrucción.

Modo: 2 usando R7

Ejemplos: 9

e.2) Absoluto

Forma general: OPR @#A

Descripción: La localidad de memoria siguiente a la instrucción contiene la dirección absoluta del operando.

Modo: 3 usando R7

Ejemplos: 10

e.3) Relativo

Forma general: OPR A

Descripción: La localidad de memoria siguiente a la ins-- trucción, sumada al PC proporcionan la dirección del operan-- do.

Modo: 6 usando R7

Ejemplos: 11

e.4) Relativo diferido

Forma general: OPR @A

Descripción: La localidad de memoria siguiente a la ins-- trucción sumada al PC proporciona el apuntador a la dirección del operando.

Modo: 7 usando R7

Ejemplos: 12

EJEMPLOS

1.1

```
005200                  INC     R0
                        ;
                        ;SUMA UNO AL CONTENIDO DE R0.
                        ;
```

Antes	Despues
001202/005200	001202/005200
_$0/000000	_$0/000001
_$7/001202	_$7/001204
_$S/000000	_$S/170020
-	-

1.2

```
105102                  COMB    R2
                        ;
                        ;COMPLEMENTO LOGICO DEL BYTE BAJO(BITS 0-7) EN R2.
                        ;LAS INSTRUCCIONES DE BYTE USADAS SOBRE LOS
                        ;REGISTROS GENERALES SOLO OPERAN EN LOS BITS 0-7.
                        ;
```

Antes	Despues
001206/105102	001206/105102
_$2/103252	_$2/103125
_$7/001206	_$7/001210
_$S/170020	_$S/170021
-	-

1.3

```
060103                  ADD     R1,R3
                        ;
                        ;SUMA EL CONTENIDO DE R1 AL CONTENIDO DE R3.
                        ;
```

Antes	Depues
001204/060103	001204/060103
_$1/000005	_$1/000005
_$3/000007	_$3/000014
_$7/001204	_$7/001206
_$S/170020	_$S/170020
-	-

2.1

```
005024                  CLR     (R4)+
                        ;
                        ;USA EL CONTENIDO DE R4 COMO LA DIRECCION DEL
                        ;OPERANDO. PONE A CEROS EL OPERANDO(PALABRA) E
                        ;INCREMENTA EL CONTENIDO DE R4 EN DOS.
                        ;
```

Antes

```
 001210/005024
_$4/000010
_000010/174216
_$7/001210
_$S/170021
-
```

Despues

```
 001210/005024
_$4/000012
_000010/000000
_$7/001212
_$S/170024
-
```

2.2

```
105024                  CLRB    (R4)+
                        ;
                        ;USA EL CONTENIDO DE R4 COMO LA DIRECCION DEL
                        ;OPERANDO. PONE A CEROS EL OPERANDO(BYTE) E
                        ;INCREMENTA EL CONTENIDO DE R4 EN UNO.
                        ;
```

Antes

```
 001212/105024
_$4/000006
_000006/173215
_$7/001212
_$S/170024
```

Despues

```
 001212/105024
_$4/000007
_000006/173000
_$7/001214
_$S/170024
-
```

2.3

```
060022                  ADD     R0,(R2)+
                        ;
                        ;EL CONTENIDO DE R0 SERA SUMADO AL OPERANDO
                        ;CUYA DIRECCION ESTA CONTENIDA EN R2. DESPUES
                        ;SE INCREMENTA R2 EN DOS.
                        ;
```

Antes

```
 001214/060022
_$0/000007
_$2/000024
_000024/000007
_$7/001214
_$S/170024
-
```

Despues

```
 001214/060022
_$0/000007
_$2/000026
_000024/000016
_$7/001216
_$S/170020
-
```

3.1

005245 INC -(R5)

```
;
;EL CONTENIDO DE R5 SE DECREMENTA EN DOS Y
;DESPUES SE USA COMO LA DIRECCION DEL OPERANDO.
;EL OPERANDO(PALABRA) SE INCREMENTA EN UNO.
;
```

Antes	Despues
001216/005245	001216/005245
_$5/000020	_$5/000016
_000016/002222	_000016/002223
_$7/001216	_$7/001220
_$8/170020	_$8/170020
_	_

3.2

105245 INCB -(R5)

```
;
;EL CONTENIDO DE R5 SE DECREMENTA EN UNO Y
;DESPUES SE USA COMO LA DIRECCION DEL OPERANDO.
;EL OPERANDO(BYTE) SE INCREMENTA EN UNO.
;
```

Antes	Despues
001220/105245	001220/105245
_$5/000347	_$5/000346
_000346/043721	_000346/043722
_$7/001220	_$7/001222
_$8/170020	_$8/170030
_	_

3.3

064401 ADD -(R4),R1

```
;
;EL CONTENIDO DE R4 SE DECREMENTA EN DOS Y
;DESPUES SE UTILIZA COMO LA DIRECCION DEL
;OPERANDO QUE SERA SUMADO AL CONTENIDO DE R1.
;
```

Antes	Despues
001222/064401	001222/064401
_$1/000017	_$1/000064
_$4/000032	_$4/000030
_000030/000045	_000030/000045
_$7/001222	_$7/001224
_$8/170020	_$8/170020

4.1

```
005063  000100          CLR     100(R3)
                        ;
                        ;SE PONE A CEROS LA LOCALIDAD(PALABRA)
                        ;DIRECCIONADA POR LA SUMA DE 100 Y EL CONTENIDO
                        ;DE R3. EL CONTENIDO DE R3 NO SE ALTERA.
```

Antes	Despues
001224/005063	001224/005063
_001226/000100	_001226/000100
_$3/000004	_$3/000004
_000104/177333	_000104/000000
_$7/001224	_$7/001230
_$S/170020	_$S/170024
-	-

4.2

```
105164  000200          COMB    200(R4)
                        ;
                        ;COMPLEMENTA LOGICAMENTE EL CONTENIDO DE LA
                        ;LOCALIDAD(BYTE) DIRECCIONADA POR LA SUMA DE
                        ;200 Y R4. EL CONTENIDO DE R4 NO SE ALTERA.
                        ;
```

Antes	Despues
001230/105164	001230/105164
_001232/000200	_001232/000200
_$4/000002	_$4/000002
_000202/174562	_000202/174615
_$7/001230	_$7/001234
_$S/170000	_$S/170031
-	-

4.3

```
066360  000010  000020        ADD     10(R3),20(R0)
                        ;
                        ;SUMA EL CONTENIDO DE LA LOCALIDAD DIRECCIONADA
                        ;POR LA SUMA DE 10 Y R3, AL CONTENIDO DE LA
                        ;LOCALIDAD DIRECCIONADO POR LA SUMA DE 20 Y R0.
                        ;
```

Antes	Despues
001234/066360	001234/066360
_001236/000010	_001236/000010
_001240/000020	_001240/000020
_$0/000030	_$0/000030
_$3/000050	_$3/000050
_000050/000037	_000050/000134
_000060/000075	_000060/000075
_$7/001234	_$7/001242
_$S/170031	_$S/170020

5.1

```
005011          CLR     @R1
                ;
                ;EL CONTENIDO DE R1 APUNTA AL OPERANDO QUE
                ;SERA PUESTO A CEROS.
                ;
```

Antes	Despues
001242/005011	001242/005011
_$1/000044	_$1/000044
_000044/035240	_000044/000000
_$7/001242	_$7/001244
_$S/170020	_$S/170024
-	-

5.2

```
105212          INCB    @R2
                ;
                ;EL CONTENIDO DE R2 APUNTA AL OPERANDO QUE
                ;SERA INCREMENTADO EN UNO.
                ;
```

Antes	Despues
001244/105212	001244/105212
_$2/000070	_$2/000070
_000070/000000	_000070/000001
_$7/001244	_$7/001246
_$S/170024	_$S/170020
-	-

6

```
005234          INC     @(R4)+
                ;EL CONTENIDO DE R4 APUNTA A LA DIRECCION
                ;DEL OPERANDO QUE SERA INCREMENTADO EN UNO,
                ;DESPUES DE LO CUAL R4 SE INCREMENTA EN DOS.
                ;
```

Antes	Despues
001246/005234	001246/005234
_$4/000036	_$4/000040
_000036/000054	_000036/000054
_000054/000007	_000054/000010
_$7/001246	_$7/001250
_$S/170020	_$S/170020
-	-

7

```
005155                  COM     @-(R5)
                        ;
                        ;EL CONTENIDO DE R5 SE DECREMENTA EN DOS.
                        ;DESPUES DE LO CUAL APUNTA A LA DIRECCION
                        ;DEL OPERANDO QUE SERA COMPLEMENTADO
                        ;LOGICAMENTE.
                        ;
```

Antes	Despues
001250/005155	001250/005155
_$5/000040	_$5/000036
_000036/000020	_000036/000020
_000020/000000	_000020/177777
_$7/001250	_$7/001252
_$S/170020	_$S/170031
-	-

8

```
067300  000200          ADD     @200(R3),R0
                        ;
                        ;LA SUMA DE 200 Y R3 DETERMINA EL APUNTADOR A
                        ;LA DIRECCION DE LA LOCALIDAD QUE SERA SUMADA A R0.
                        ;
```

Antes	Despues
001252/067300	001252/067300
_001254/000200	_001254/000200
_$0/000015	_$0/000033
_$3/000010	_$3/000010
_000210/000012	_000210/000012
_000012/000016	_000012/000016
_$7/001252	_$7/001256
_$S/170031	_$S/170020
-	-

9

```
012704  000010                  MOV     #10,R4
                        ;
                        ;MUEVE A R4 EL NUMERO 10
                        ;
```

Antes	Despues
001256/012704	001256/012704
_001260/000010	_001260/000010
_$4/000000	_$4/000010
_$7/001256	_$7/001262
_$S/170000	_$S/170020
-	-

10

```
063701  000100                  ADD     @#100,R1
                        ;
                        ;SUMA EL CONTENIDO DE LA LOCALIDAD 100 A R1.
                        ;
```

Antes	Despues
001266/063701	001266/063701
_001270/000100	_001270/000100
_$1/000033	_$1/000126
_000100/000073	_000100/000073
_$7/001266	_$7/001272
_$S/170000	_$S/170020
-	-

11

```
005267  000044                  INC     Z
                                ;
                                ;INCREMENTA EL CONTENIDO DE LA LOCALIDAD
                                ;SIMBOLICA Z EN UNO. EL CONTENIDO DE LA PALABRA
                                ;SIGUIENTE A LA INSTRUCCION SE SUMA AL PC PARA
```

Antes	Despues
001272/005267	001272/005267
_001274/000044	_001274/000044
_001342/000000	_001342/000001
_$7/001272	_$7/001276
_$S/170020	_$S/170020
-	-

12

```
005077  000040                  CLR     @Z
                                ;
                                ;LA LOCALIDAD SIMBOLICA Z APUNTA A LA
                                ;DIRECCION DEL OPERANDO QUE SERA PUESTO A CEROS.
                                ;EL CONTENIDO DE LA PALABRA SIGUIENTE A LA
                                ;INSTRUCCION SE SUMA AL PC PARA OBTENER LA
                                ;DIRECCION DE Z.
                                ;
```

Antes	Despues
001276/005077	001276/005077
_001300/000040	_001300/000040
_001342/000100	_001342/000100
_000100/000073	_000100/000000
_$7/001276	_$7/001302
_$S/170020	_$S/170024
-	-

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

CONJUNTO DE INSTRUCCIONES

M. EN C. MARCIAL PORTILLA ROBERTSON

JUNIO, 1980

CHAPTER 4

INSTRUCTION SET

4.1 INTRODUCTION

The specification for each instruction includes the mnemonic, octal code, binary code, a diagram showing the format of the instruction, a symbolic notation describing its execution and the effect on the condition codes, a description, special comments, and examples.

MNEMONIC: This is indicated at the top corner of each page. When the word instruction has a byte equivalent, the byte mnemonic is also shown.

INSTRUCTION FORMAT: A diagram accompanying each instruction shows the octal op code, the binary op code, and bit assignments. (Note that in byte instructions the most significant bit (bit 15) is always a 1.)

SYMBOLS:

( ) = contents of

SS or src = source address

DD or dst = destination address

loc = location

← = becomes

↑ = "is popped from stack"

↓ = "is pushed onto stack"

∧ = boolean AND

∨ = boolean OR

⊻ = exclusive OR

~ = boolean not

Reg or R = register

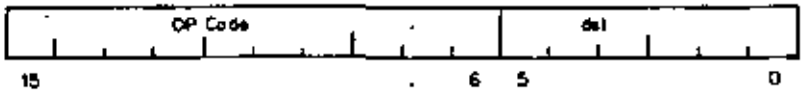
B = Byte

* = 0 for word
    1 for byte

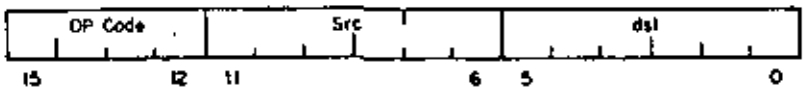
4.2 INSTRUCTION FORMATS

The major instruction formats are:

Single Operand Group

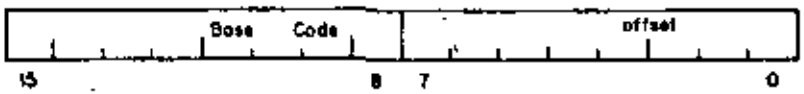


Double Operand Group



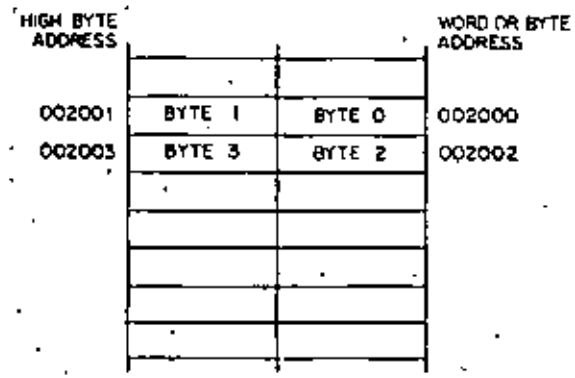
Register-Source or Destination

Branch



Byte Instructions

The PDP-11 processor includes a full complement of instructions that manipulate byte operands. Since all PDP-11 addressing is byte-oriented, byte manipulation addressing is straightforward. Byte instructions with autoincrement or autodecrement direct addressing cause the specified register to be modified by one to point to the next byte of data. Byte operations in register mode access the low-order byte of the specified register. These provisions enable the PDP-11 to perform as either a word or byte processor. The numbering scheme for word and byte addresses in core memory is:



The most significant bit (Bit 15) of the instruction word is set to indicate a byte instruction.

Example:

Symbolic	Octal	
CLR	0050DD	Clear Word
CLRB	1050DD	Clear Byte

NOTE

The term PC (Program Counter) in the Operation explanation of the instructions refers to the updated PC.

4.3 LIST OF INSTRUCTIONS

Instructions are shown in the following sequence. Other instructions are found in Chapters 9, 11, and 12.

▲—The SXT, XOR, MARK, SOB, and RTT instructions are implemented in the PDP-11/34, 11/45 and 11/55.

*—The SPL instruction is implemented only in the PDP-11/45 and PDP-11/55. The MFPS and MTPS instructions are implemented only in the PDP-11/34.

SINGLE OPERAND

Mnemonic	Instruction	Op Code	Page
General			
CLR(B)	clear destination	■050DD	4-6
COM(B)	complement dst	■051DD	4-7
INC(B)	increment dst	■052DD	4-8
DEC(B)	decrement dst	■053DD	4-9
NEG(B)	negate dst	■054DD	4-10
TST(B)	test dst	■057DD	4-11
Shift & Rotate			
ASR(B)	arithmetic shift right	■062DD	4-13
ASL(B)	arithmetic shift left	■063DD	4-14
ROR(B)	rotate right	■060DD	4-15
ROL(B)	rotate left	■061DD	4-16
SWAB	swap bytes	0003DD	4-17
Multiple Precision			
ADC(B)	add carry	■055DD	4-19
SBC(B)	subtract carry	■056DD	4-20
▲ SXT	sign extend	0067DD	4-21
MFPS	move byte from processor status	■1067DD	4-22
MTPS	move byte to processor status	■1064SS	4-23

DOUBLE OPERAND

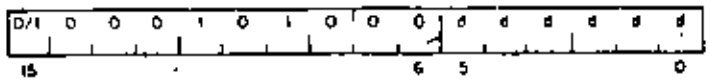
Mnemonic	Instruction	Op Code	Page
General			
MOV(B)	move source to destination	■1SSDD	4-25
CMP(B)	compare src to dst	■2SSDD	4-26
ADD	add src to dst	06SSDD	4-27
SUB	subtract src from dst	16SSDD	4-28
Logical			
BIT(B)	bit test	■3SSDD	4-30
BIC(B)	bit clear	■4SSDD	4-31
BIS(B)	bit set	■5SSDD	4-32
▲ XOR	exclusive OR	074RDD	4-33

PROGRAM CONTROL

Mnemonic	Instruction	Op Code or Base Code	Page
Branch			
BR	branch (unconditional)	000400	4-35
BNE	branch if not equal (to zero)	001000	4-36
BEQ	branch if equal (to zero)	001400	4-37
BPL	branch if plus	100000	4-38
BMI	branch if minus	100400	4-39
BVC	branch if overflow is clear	102000	4-40
BVS	branch if overflow is set	102400	4-41
BCC	branch if carry is clear	103000	4-42
BCS	branch if carry is set	103400	4-43
Signed Conditional Branch			
BGE	branch if greater than or equal (to zero)	002000	4-45
BLT	branch if less than (zero)	002400	4-46
BGT	branch if greater than (zero)	003000	4-47
BLE	branch if less than or equal (to zero)	003400	4-48
Unsigned Conditional Branch			
BHI	branch if higher	101000	4-50
BLOS	branch if lower or same	101400	4-51
BHIS	branch if higher or same	103000	4-52
BLO	branch if lower	103400	4-53
Jump & Subroutine			
JMP	jump	0001DD	4-54
JSR	jump to subroutine	004RDD	4-56
RTS	return from subroutine	00020R	4-58
▲ MARK	mark	006400	4-59
▲ SOB	subtract one and branch (if ≠ 0)	077R00	4-61
* SPL	set priority level	00023N	4-62
Trap & Interrupt			
EMT	emulator trap	104000—104377	4-63
TRAP	trap	104400—104777	4-64
BPT	breakpoint trap	000003	4-65
IOT	input/output trap	000004	4-66
RTI	return from interrupt	000002	4-67
▲ RTT	return from interrupt	000006	4-68
MISCELLANEOUS			
HALT	halt	000000	4-72
WAIT	wait for interrupt	000001	4-73
RESET	reset external bus	000005	4-74
Condition Code Operation			
CLC, CLV, CLZ, CLN, CCC	clear	000240	4-75
SEC, SEV, SEZ, SEN, SCC	set	000260	4-75

CLR
CLRB

clear destination ■050DD



Operation: (dst)←0

Condition Codes:
N: cleared
Z: set
V: cleared
C: cleared

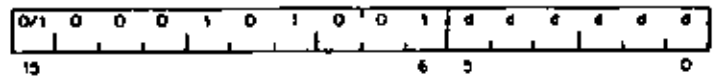
Description: Word: Contents of specified destination are replaced with zeroes.
Byte: Same

Example:

```
CLR R1

        Before              After
(R1) = 177777       (R1) = 000000

        N Z V C             N Z V C
        1 1 1 1             0 1 0 0
```


COM
COMB

complement dst ■051DD



Operation: (dst)←~(dst)

Condition Codes:
N: set if most significant bit of result is set; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: set

Description: Replaces the contents of the destination address by their logical complement (each bit equal to 0 is set and each bit equal to 1 is cleared)
Byte: Same

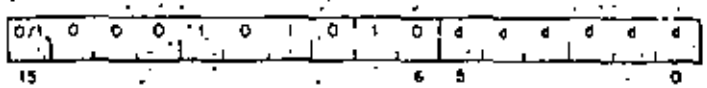
Example:

```
COM R0

        Before              After
(R0) = 013333       (R0) = 164444

        N Z V C             N Z V C
        0 1 1 0             1 0 0 1
```

INC
INCB

increment dst

#052DD



Operation: (dst)←(dst)+1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) held 077777 (word) or 177 (byte)
cleared otherwise
C: not affected

Description: Word: Add one to contents of destination
Byte: Same

Example:

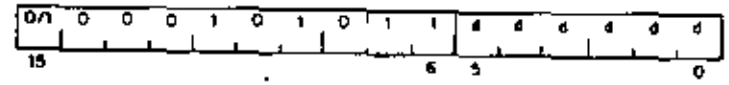
INC R2

Before	After
(R2) = 000333	(R2) = 000334
N Z V C	N Z V C
0 0 0 0	0 0 0 0

DEC
DECB

decrement dst

#053DD



Operation: (dst)←(dst)−1

Condition Codes:
N: set if result is <0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if (dst) was 100000 (word) or 200 (byte)
cleared otherwise
C: not affected

Description: Word: Subtract 1 from the contents of the destination
Byte: Same

Example:

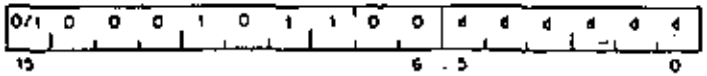
DEC R5

Before	After
(R5) = 000001	(R5) = 000000
N Z V C	N Z V C
1 0 0 0	0 1 0 0

NEG
NEGB

negate dst

#054DD



Operation: (dst)← −(dst)

Condition Codes:
N: set if the result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: set if the result is 100000 (word) or 200 (byte) cleared otherwise
C: cleared if the result is 0; set otherwise

Description:
Word: Replaces the contents of the destination address by its two's complement. Note that 100000 is replaced by itself (in two's complement notation the most negative number has no positive counterpart).
Byte: Same

Example:

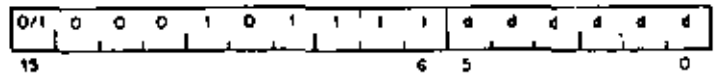
NEG R0

Before	After
(R0) = 000010	(R0) = 177770
N Z V C	N Z V C
0 0 0 0	1 0 0 1

TST
TSTB

test dst

#057DD



Operation: (dst)←(dst)

Condition Codes:
N: set if the result is < 0; cleared otherwise
Z: set if result is 0; cleared otherwise
V: cleared
C: cleared

Description:
Word: Sets the condition codes N and Z according to the contents of the destination address
Byte: Same

Example:

TST R1

Before	After
(R1) = 012340	(R1) = 012340
N Z V C	N Z V C
0 0 1 1	0 0 0 0

Shifts

Scaling data by factors of two is accomplished by the shift instructions:

ASR - Arithmetic shift right

ASL - Arithmetic shift left

The sign bit (bit 15) of the operand is replicated in shifts to the right. The low order bit is filled with 0 in shifts to the left. Bits shifted out of the C bit, as shown in the following examples, are lost.

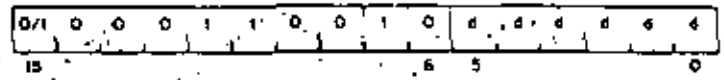
Rotates

The rotate instructions operate on the destination word and the C bit as though they formed a 17 bit "circular buffer". These instructions facilitate sequential bit testing and detailed bit manipulation.

ASR
ASRB

arithmetic shift right

#062DD



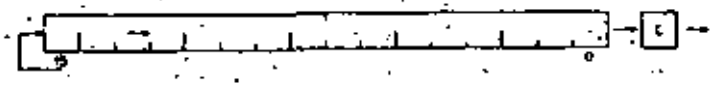
Operation: (dst)←(dst) shifted one place to the right

Condition Codes:
- N: set if the high-order bit of the result is set (result < 0); cleared otherwise
- Z: set if the result = 0; cleared otherwise
- V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
- C: loaded from low-order bit of the destination

Description:

Word: Shifts all bits of the destination right one place. Bit 15 is replicated. The C-bit is loaded from bit 0 of the destination. ASR performs signed division of the destination by two.

Word:

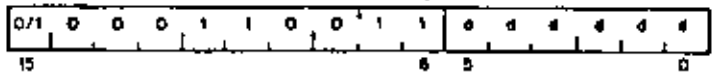


Byte:

ASL
ASLB

arithmetic shift left

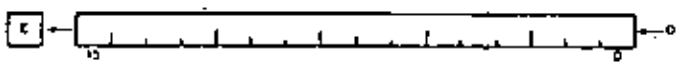
■063DD



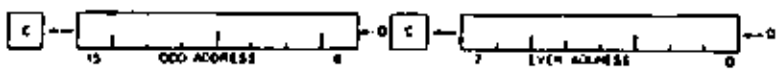
Operation: (dst)←(dst) shifted one place to the left

Condition Codes:
N: set if high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded with the exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded with the high-order bit of the destination

Description: Word: Shifts all bits of the destination left one place. Bit 0 is loaded with an 0. The C-bit of the status word is loaded from the most significant bit of the destination. ASL performs a signed multiplication of the destination by 2 with overflow indication.
Word:



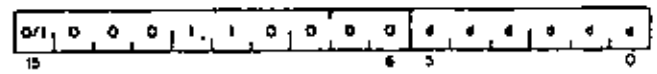
Byte:



ROR
RORB

rotate right

■060DD

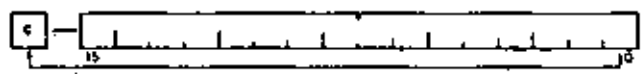


Condition Codes:
N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if all bits of result = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the low-order bit of the destination

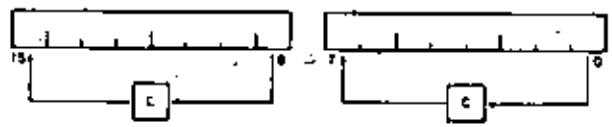
Description: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.
Byte: Same

Example:

Word:

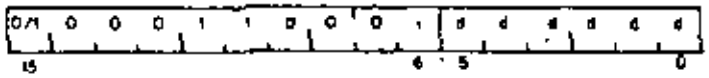


Byte:



ROL
ROLB

rotate left

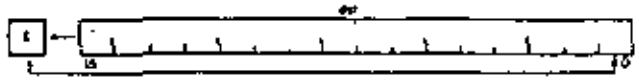


Condition Codes: N: set if the high-order bit of the destination is set (result < 0); cleared otherwise
Z: set if all bits of the destination = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the high-order bit of the destination

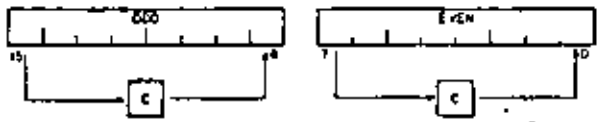
Description: Word. Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.
Byte: Same

Example:

Word:



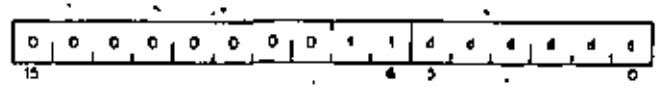
Bytes:



SWAB

swap bytes

0003DD



Operation: Byte 1/Byte 0 ← Byte 0/Byte 1

Condition Codes: N: set if high-order bit of low-order byte (bit 7) of result is set; cleared otherwise
Z: set if low-order byte of result = 0; cleared otherwise
V: cleared
C: cleared

Description: Exchanges high-order byte and low-order byte of the destination word (destination must be a word address).

Example:

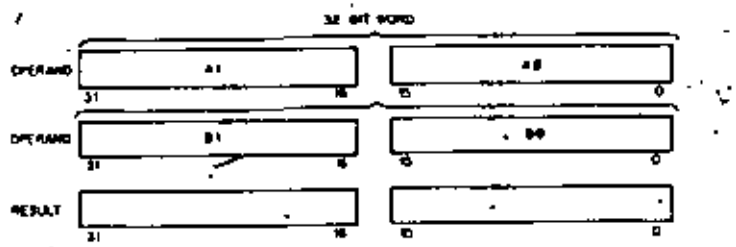
SWAB R1

Before	After
(R1) = 077777	(R1) = 177577
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Multiple Precision

It is sometimes necessary to do arithmetic on operands considered as multiple words or bytes. The PDP-11 makes special provision for such operations with the instructions ADC (Add Carry) and SBC (Subtract Carry) and their byte equivalents.

For example two 16-bit words may be combined into a 32-bit double precision word and added or subtracted as shown below:



Example:

The addition of −1 and −1 could be performed as follows:

−1 = 37777777777

(R1) = 177777 (R2) = 177777 (R3) = 177777 (R4) = 177777

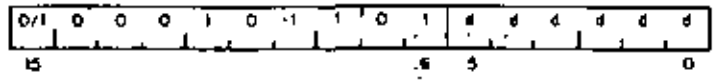
```
ADD   R1,R2
ADC   R3
ADD   R4,R3
```

1. After (R1) and (R2) are added, 1 is loaded into the C bit
2. ADC instruction adds C bit to (R3); (R3) = 0
3. (R3) and (R4) are added
4. Result is 37777777776 or −2

ADC
ADCB

add carry

#055DD



Operation: (dst)←(dst) + (C)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if (dst) was 077777 (word) or 200 (byte) and (C) was 1; cleared otherwise
C: set if (dst) was 177777 (word) or 377 (byte) and (C) was 1; cleared otherwise

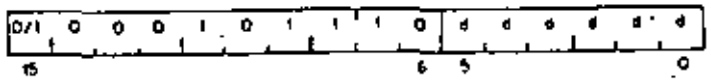
Description: Adds the contents of the C-bit into the destination. This permits the carry from the addition of the low order words to be carried into the high-order result.
Byte: Same

Example: Double precision addition may be done with the following instruction sequence

```
ADD   A0,B0     ; add low-order parts
ADC   B1        ; add carry into high-order
ADD   A1,B1     ; add high order parts
```

SBC
SBCB

subtract carry



=056DD

Operation: (dst)←(dst)-(C)

Condition Codes:
N: set if result < 0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if (dst) was 100000 (word) or 200 (byte) cleared otherwise
C: set if (dst) was 0 and C was 1; cleared otherwise

Description: Word: Subtracts the contents of the C-bit from the destination. This permits the carry from the subtraction of two low-order words to be subtracted from the high order part of the result.
Byte: Same

Example: Double precision subtraction is done by:

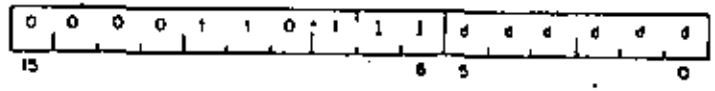
```
SUB   A0,B0
SBC   B1
SUB   A1,B1
```


SXT

Used in the PDP-11/34, 11/45 and 11/55

sign extend

0067DD



Operation: (dst) ← 0 if *N* bit is clear
(dst) ← -1 *N* bit is set

Condition Codes:
N: unaffected
Z: set if N bit clear
V: cleared
C: unaffected

Description: If the condition code bit N is set then a -1 is placed in the destination operand; if N bit is clear, then a 0 is placed in the destination operand. This instruction is particularly useful in multiple precision arithmetic because it permits the sign to be extended through multiple words.

Used in the PDP-11/34

MFPS

move byte from processor status word

1067DD

Operation: (dst) ← PS <0:7>
dst lower 8 bits

Condition Code Bits:
N = set if PS bit 7 = 1; cleared otherwise
Z = set if PS <0:7> = 0; cleared otherwise
V = cleared
C = not affected

Description: The 8 bit contents of the PS are moved to the effective destination. If destination is mode 0, PS bit 7 is sign extended through the upper byte of the register. The destination operand address is treated as a byte address.

Example:

```
MFPS R0

        before                  after
R0 [0]                  R0 [000014]
PS [000014]             PS [000014]
```


MTPS

Used in the PDP-11/34

move byte to processor status word

1064SS

Operation: PS <0:7> ← (SRC)

Condition Codes: Set according to effective SRC operand bits 0–3.

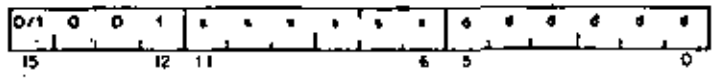
Description: The 8 bits of the effective operand replaces the current contents of the PS <0:7>. The source operand address is treated as a byte address.
Note that the T bit (PS bit 4) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority bits (PS <5:7>) in the PS.

4.5 DOUBLE OPERAND INSTRUCTIONS

Double operand instructions provide an instruction (and time) saving facility since they eliminate the need for "load" and "save" sequences such as those used in accumulator-oriented machines.

MOV
MOVB

move source to destination ■1SSDD



Operation: (dst)←(src)

Condition Codes:
N: set if (src) < 0; cleared
Z: set if (src) = 0; cleared
V: cleared
C: not affected

Description:
Word: Moves the source operand to the destination location. The previous contents of the destination are lost. The contents of the source address are not affected.
Byte: Same as MOV. The MOVB to a register (unique among byte instructions) extends the most significant bit of the low order byte (sign extension). Otherwise MOVB operates on bytes exactly as MOV operates on words.

Example:

MOV XXX,R1 ; loads Register 1 with the contents of memory location; XXX represents a programmer-defined mnemonic used to represent a memory location.

MOV #20,R0 ; loads the number 20 into Register 0; "#" indicates that the value 20 is the operand.

MOV @#20,-(R6) ; pushes the operand contained in location 20 onto the stack.

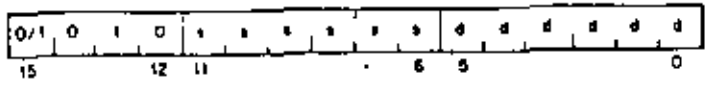
MOV (R6)+,@#177566 ; pops the operand off the stack and moves it into memory location 177566 (terminal print buffer).

MOV R1,R3 ; performs an inter register transfer.

MOVB @#177562,@#177566 ; moves a character from terminal keyboard buffer to terminal printer buffer.

CMP
CMPB

compare src to dst — 1SSDD



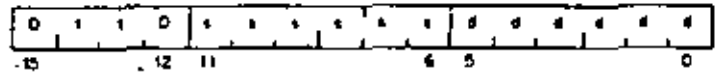
Operation: (src)–(dst)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow; that is, operands were of opposite signs and the sign of the destination was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Description: Compares the source and destination operands and sets the condition codes which may then be used for arithmetic and logical conditional branches. Both operands are unaffected. The only action is to set the condition codes. The compare is customarily followed by a conditional branch instruction. Note that unlike the subtract instruction the order of operation is (src)–(dst), not (dst)–(src).

ADD

add src to dst — 06SSDD



Operation: (dst)←(src) + (dst)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation; that is both operands were of the same sign and the result was of the opposite sign; cleared otherwise
C: set if there was a carry from the most significant bit of the result; cleared otherwise

Description: Adds the source operand to the destination operand and stores the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. Two's complement addition is performed.

Examples:

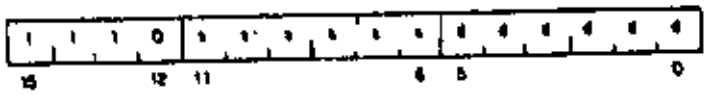
Add to register:	ADD	20,R0
Add to memory:	ADD	R1,XXX
Add register to register:	ADD	R1,R2
Add memory to memory:	ADD@	# 17750,XXX

XXX is a programmer-defined mnemonic for a memory location.

SUB

subtract src from dst

16SSDD



Operation: (dst)←(dst)−(src)

Condition Codes:
N: set if result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: set if there was arithmetic overflow as a result of the operation, that is if operands were of opposite signs and the sign of the source was the same as the sign of the result; cleared otherwise
C: cleared if there was a carry from the most significant bit of the result, set otherwise

Description: Subtracts the source operand from the destination operand and leaves the result at the destination address. The original contents of the destination are lost. The contents of the source are not affected. In double-precision arithmetic the C-bit, when set, indicates a "borrow".

Example:

SUB R1,R2

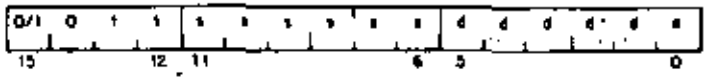
Before	After
(R1) = 011111	(R1) = 011111
(R2) = 012345	(R2) = 001234
N Z V C	N Z V C
1 1 1 1	0 0 0 0

Logical
These instructions have the same format as the double operand arithmetic group. They permit operations on data at the bit level.

BIT
BITB

bit test

#3SSDD



Operation: (src) ∧ (dst)

Condition Codes:
N: set if high-order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs logical "and" comparison of the source and destination operands and modifies condition codes accordingly. Neither the source nor destination operands are affected. The BIT instruction may be used to test whether any of the corresponding bits that are set in the destination are also set in the source or whether all corresponding bits set in the destination are clear in the source.

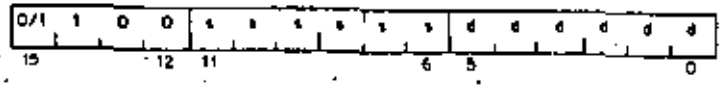
Example:

```
BIT    #30,R3        ;test bits 3 and 4 of R3 to see
                     ; if both are off

(30)=0 000 000 000 011 000
```


BIC
BICB

bit clear

#45SDD



Operation: (dst)←~(src)∧(dst)

Condition Codes:
N: set if high order bit of result set; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Clears each bit in the destination that corresponds to a set bit in the source. The original contents of the destination are lost. The contents of the source are unaffected.

Example:

```
                BIC R3,R4

          Before                After
(R3) = 001234         (R3) = 001234

(R4) = 001111         (R4) = 000101

        N Z V C               N Z V C
        1 1 1 1               0 0 0 1
```

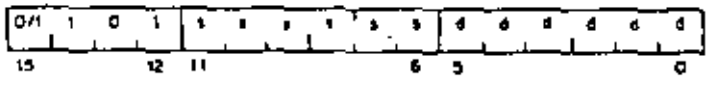
Before: (R3)=0 000 001 010 011 100
(R4)=0 000 001 001 001 001

After: (R4)=0 000 000 001 000 001

BIS
BISB

bit set

***5SSDD**



Operation: (dst)←(src) ∨ (dst)

Condition Codes:
N: set if high order bit of result set, cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: not affected

Description: Performs "inclusive OR" operation between the source and destination operands and leaves the result at the destination address; that is, corresponding bits set in the source are set in the destination. The contents of the destination are lost.

Example:

BIS R0,R1

Before	After
(R0) = 001234	(R0) = 001234
(R1) = 001111	(R1) = 001335
N Z V C	N Z V C
0 0 0 0	0 0 0 0

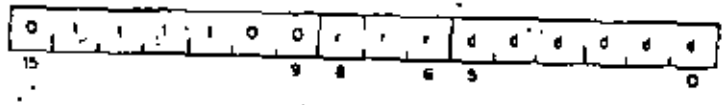
Before: (R0)=0 000 001 010 011 100
(R1)=0 000 001 001 001 001

After: (R1)=0 000 001 011 011 101

XOR

Used in the PDP-11/34, 11/45 and 11/55

exclusive OR **074ROD**



Operation: (dst)←Rv(dst)

Condition Codes:
N: set if the result <0; cleared otherwise
Z: set if result = 0; cleared otherwise
V: cleared
C: unaffected

Description: The exclusive OR of the register and destination operand is stored in the destination address. Contents of register are unaffected. Assembler format is: XOR R,D

Example:

XOR R0,R2

Before	After
(R0) = 001234	(R0) = 001234
(R2) = 001111	(R2) = 000325

Before: (R0)=0 000 001 010 011 100
(R2)=0 000 001 001 001 001

After: (R2)=0 000 000 011 010 101

4.6 PROGRAM CONTROL INSTRUCTIONS

Branches

The instruction causes a branch to a location defined by the sum of the offset (multiplied by 2) and the current contents of the Program Counter if:

a) the branch instruction is unconditional

b) it is conditional and the conditions are met after testing the condition codes (status word).

The offset is the number of words from the current contents of the PC. Note that the current contents of the PC point to the word following the branch instruction.

Although the PC expresses a byte address, the offset is expressed in words. The offset is automatically multiplied by two to express bytes before it is added to the PC. Bit 7 is the sign of the offset; if it is set, the offset is negative and the branch is done in the backward direction. Similarly if it is not set, the offset is positive and the branch is done in the forward direction.

The 8-bit offset allows branching in the backward direction by 200, words (400, bytes) from the current PC, and in the forward direction by 177, words (376, bytes) from the current PC.

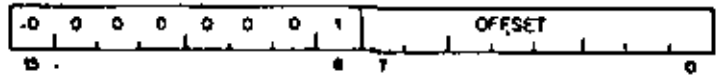
The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

Bxx loc

Where "Bxx" is the branch instruction and "loc" is the address to which the branch is to be made. The assembler gives an error indication in the instruction if the permissable branch range is exceeded. Branch instructions have no effect on condition codes

BR

branch (unconditional) 000400 Plus offset



Operation: PC ← PC + (2 x offset)

Description: Provides a way of transferring program control within a range of −128 to +127 words with a one word instruction.

New PC address = updated PC + (2 X offset)

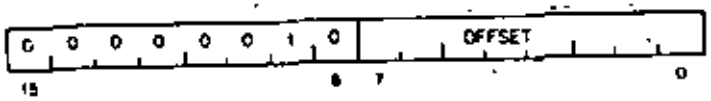
Updated PC = address of branch instruction + 2

Example: With the Branch instruction at location 500, the following offsets apply.

New PC Address	Offset Code	Offset (decimal)
474	375	−3
476	376	−2
500	377	−1
502	000	0
504	001	+1
506	002	+2

BNE

branch if not equal (to zero) 001000 Plus offset



Operation: PC ← PC + (2 x offset) if Z = 0

Condition Codes: Unaffected

Description: Tests the state of the Z-bit and causes a branch if the Z-bit is clear. BNE is the complementary operation to BEQ. It is used to test inequality following a CMP, to test that some bits set in the destination were also in the source, following a BIT, and generally, to test that the result of the previous operation was not zero.

Example:

```
CMP   A,B          ; compare A and B
BNE   C            ; branch if they are not equal
```

will branch to C if A ≠ B

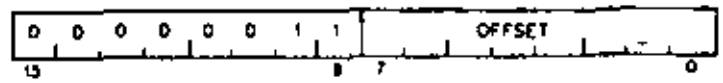
and the sequence

```
ADD   A,B          ; add A to B
BNE   C            ; Branch if the result is not
                     equal to 0
```

will branch to C if A + B ≠ 0

BEQ

branch if equal (to zero) 001400 Plus offset



Operation: PC ← PC + (2 x offset) if Z = 1

Condition Codes: Unaffected

Description: Tests the state of the Z-bit and causes a branch if Z is set. As an example, it is used to test equality following a CMP operation, to test that no bits set in the destination were also set in the source following a BIT operation, and generally, to test that the result of the previous operation was zero.

Example:

```
CMP   A,B          ; compare A and B
BEQ   C            ; branch if they are equal
```

will branch to C if A = B (A − B = 0)

and the sequence

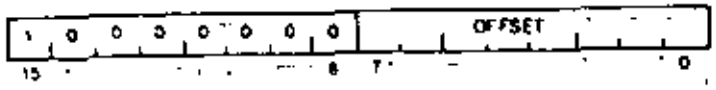
```
ADD   A,B          ; add A to B
BEQ   C            ; branch if the result = 0
```

will branch to C if A + B = 0.

BPL

branch if plus

100000 Plus offset



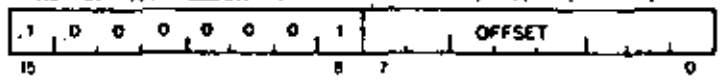
Operation: PC ← PC + (2 x offset) if N = 0

Description: Tests the state of the N-bit and causes a branch if N is clear, (positive result).

BMI

branch if minus

100400 Plus offset



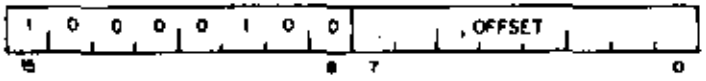
Operation: PC ← PC + (2 x offset) if N = 1

Condition Codes: Unaffected

Description: Tests the state of the N-bit and causes a branch if N is set. It is used to test the sign (most significant bit) of the result of the previous operation), branching if negative.

BVC

branch if overflow is clear — 102000 Plus offset

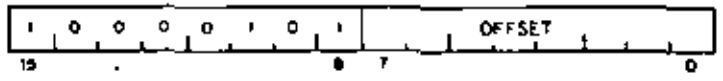


Operation: PC ← PC + (2 x offset) if V = 0

Description: Tests the state of the V bit and causes a branch if the V bit is clear. BVC is complementary operation to BVS.

BVS

branch if overflow is set — 102400 Plus offset

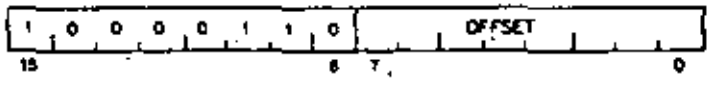


Operation: PC ← PC + (2 x offset) if V = 1

Description: Tests the state of V bit (overflow) and causes a branch if the V bit is set. BVS is used to detect arithmetic overflow in the previous operation.

BCC

branch if carry is clear — 103000 Plus offset

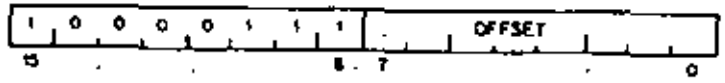


Operation: PC ← PC + (2 x offset) if C = 0

Description: Tests the state of the C-bit and causes a branch if C is clear. BCC is the complementary operation to BCS

BCS

branch if carry is set — 103400 Plus offset



Operation: PC ← PC + (2 x offset) if C = 1

Description: Tests the state of the C-bit and causes a branch if C is set. It is used to test for a carry in the result of a previous operation.

Signed Conditional Branches

Particular combinations of the condition code bits are tested with the signed conditional branches. These instructions are used to test the results of instructions in which the operands were considered as signed (two's complement) values.

Note that the sense of signed comparisons differs from that of unsigned comparisons in that in signed 16-bit, two's complement arithmetic the sequence of values is as follows:

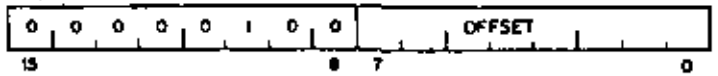
largest	077777
	077776
positive	.
	.
	000001
	000000
	177777
	177776
negative	.
	.
	100001
smallest	100000

whereas in unsigned 16-bit arithmetic the sequence is considered to be

highest	177777
	.
	.
	.
	.
	000002
	000001
lowest	000000

BGE

branch if greater than or equal (to zero) ; 002000 Plus offset

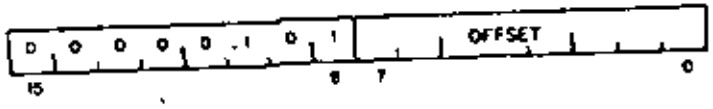


Operation: PC ← PC + (2 x offset) if N ∨ V = 0

Description: Causes a branch if N and V are either both clear or both set. BGE is the complementary operation to BLT. Thus BGE will always cause a branch when it follows an operation that caused addition of two positive numbers. BGE will also cause a branch on a zero result.

BLT

branch if less than (zero) 002400 Plus offset

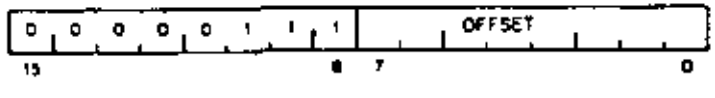


Operation: PC ← PC + (2 x offset) if N ∨ V = 1

Description: Causes a branch if the "Exclusive Or" of the N and V bits are 1. Thus BLT will always branch following an operation that added two negative numbers, even if overflow occurred. In particular, BLT will always cause a branch if it follows a CMP instruction operating on a negative source and a positive destination (even if overflow occurred). Further, BLT will never cause a branch when it follows a CMP instruction operating on a positive source and negative destination. BLT will not cause a branch if the result of the previous operation was zero (without overflow).

BLE

branch if less than or equal (to zero) 003400 Plus offset

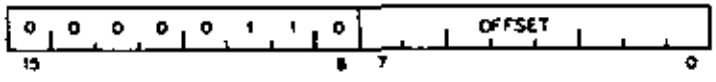


Operation: PC ← PC + (2 x offset) if Z ∨(N ∨ V) = 1

Description: Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.

BGT

branch if greater than (zero) 003000 Plus offset



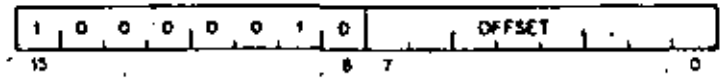
Operation: PC ← PC + (2 x offset) if Z ∨(N ∨ V) = 0

Description: Operation of BGT is similar to BGE, except BGT will not cause a branch on a zero result.

Unsigned Conditional Branches
The Unsigned Conditional Branches provide a means for testing the result of comparison operations in which the operands are considered as unsigned values.

BHI

branch if higher — 101000 Plus offset

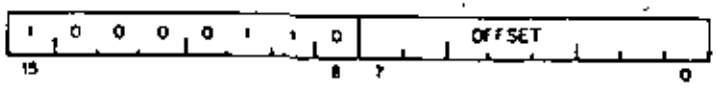


Operation: PC ← PC + (2 x offset) if C = 0 and Z = 0

Description: Causes a branch if the previous operation caused neither a carry nor a zero result. This will happen in comparison (CMP) operations as long as the source has a higher unsigned value than the destination.

BHIS

branch if higher or same — 103000 Plus offset

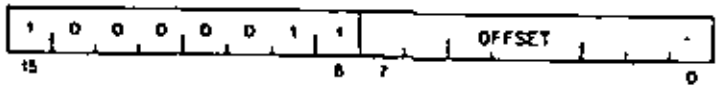


Operation: PC ← PC + (2 x offset) if C = 0

Description: BHIS is the same instruction as BCC. This mnemonic is included only for convenience.

BLOS

branch if lower or same — 101400 Plus offset

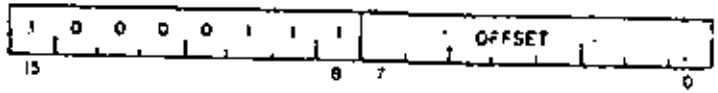


Operation: PC ← PC + (2 x offset) if C v Z = 1

Description: Causes a branch if the previous operation caused either a carry or a zero result. BLOS is the complementary operation to BHI. The branch will occur in comparison operations as long as the source is equal to, or has a lower unsigned value than the destination.

BLO

branch if lower — 103400 Plus offset

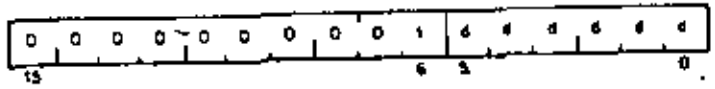


Operation: PC ← PC + (2 x offset) if C = 1

Description: BLO is same instruction as BCS. This mnemonic is included only for convenience.

JMP

jump 0001DD



Operation: PC←(dst)

Condition Codes: not affected

Description: JMP provides more flexible program branching than provided with the branch instructions. Control may be transferred to any location in memory (no range limitation) and can be accomplished with the full flexibility of the addressing modes, with the exception of register mode 0. Execution of a jump with mode 0 will cause an "illegal instruction" condition. (Program control cannot be transferred to a register.) Register deferred mode is legal and will cause program control to be transferred to the address held in the specified register. Note that instructions are word data and must therefore be fetched from an even numbered address. A "boundary error" trap condition will result when the processor attempts to fetch an instruction from an odd address.

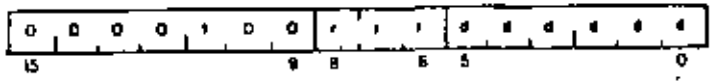
Deferred index mode JMP instructions permit transfer of control to the address contained in a selectable element of a table of dispatch vectors.

Subroutine Instructions

The subroutine call in the PDP-11 provides for automatic nesting of subroutines, reentrancy, and multiple entry points. Subroutines may call other subroutines (or indeed themselves) to any level of nesting without making special provision for storage or return addresses at each level of subroutine call. The subroutine calling mechanism does not modify any fixed location in memory, thus providing for reentrancy. This allows one copy of a subroutine to be shared among several interrupting processes. For more detailed description of subroutine programming see Chapter 5.

JSR

jump to subroutine 004RDD



Operation: (SP)←reg (push reg contents onto processor stack)

reg←PC (PC holds location following JSR; this address now put in reg)

PC←(dst) (PC now points to subroutine destination)

Description: In execution of the JSR, the old contents of the specified register (the "LINKAGE POINTER") are automatically pushed onto the processor stack and new linkage information placed in the register. Thus subroutines nested within subroutines to any depth may all be called with the same linkage register. There is no need either to plan the maximum depth at which any particular subroutine will be called or to include instructions in each routine to save and restore the linkage pointer. Further, since all linkages are saved in a reentrant manner on the processor stack, execution of a subroutine may be interrupted, the same subroutine reentered and executed by an interrupt service routine. Execution of the initial subroutine can then be resumed when other requests are satisfied. This process (called nesting) can proceed to any level.

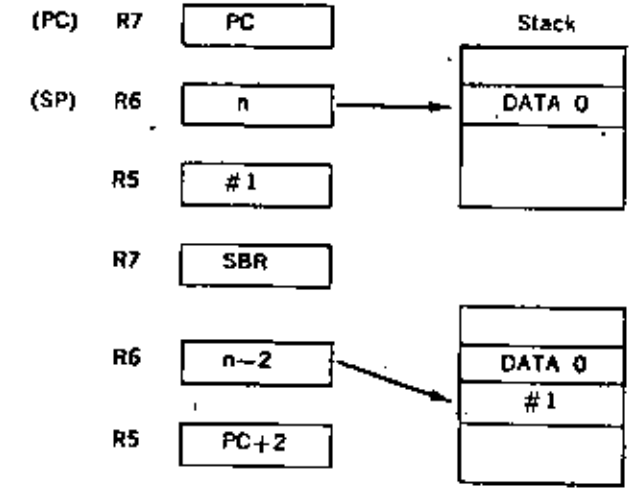
A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg)+, (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @(reg)+ and @X(reg) if the parameters are operand addresses rather than the operands themselves.

JSR PC, dst is a special case of the PDP-11 subroutine call suitable for subroutine calls that transmit parameters through the general registers. The SP and the PC are the only registers that may be modified by this call.

Another special case of the JSR instruction is JSR PC, @(SP)+ which exchanges the top element of the processor stack and the contents of the program counter. Use of this instruction allows two routines to swap program control and resume operation when recalled where they left off. Such routines are called "co-routines."

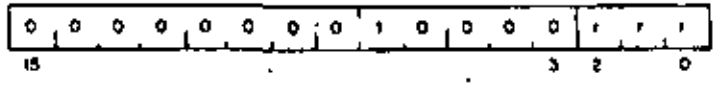
Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

Example: JSR R5, SBR



RTS

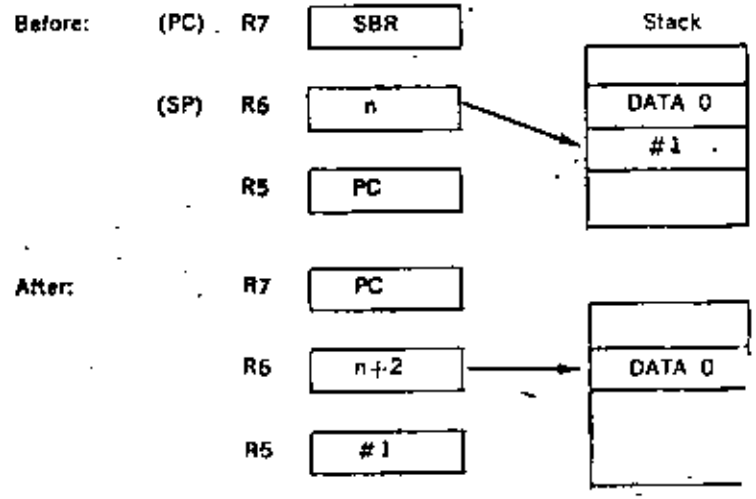
return from subroutine 00020R



Operation: PC←reg
reg←(SP)↑

Description: Loads contents of reg into PC and pops the top element of the processor stack into the specified register.
Return from a non-reentrant subroutine is typically made through the same register that was used in its call. Thus, a subroutine called with a JSR PC, dst exits with a RTS PC and a subroutine called with a JSR R5, dst, may pick up parameters with addressing modes (R5)+, X(R5), or @X(R5) and finally exits with an RTS R5

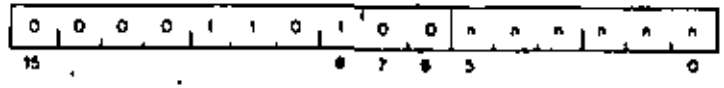
Example: RTS R5



MARK

Used in the PDP-11/34, 11/45 and 11/55

mark 00 64 NN



Operation: SP←PC + 2nn nn = number of parameters
PC←R5
R5←(SP)↑

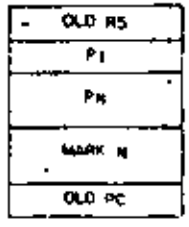
Condition Codes: unaffected

Description: Used as part of the standard PDP-11 subroutine return convention. MARK facilitates the stack clean up procedures involved in subroutine exit. Assembler format is: MARK N

Example:

```
MOV   R5,-(SP)        ;place old R5 on stack
MOV   P1,-(SP)        ;place N parameters
MOV   P2,-(SP)        ;on the stack to be
                      ;used there by the
                      ;subroutine
MOV   PN,-(SP)
MOV   #MARKN,-(SP)    ;places the instruction
                      ;MARK N on the stack
MOV   SP,R5           ;set up address at Mark N in-
                      ;struction
JSR   PC,SUB          ;jump to subroutine
```

At this point the stack is as follows:



And the program is at the address SUB which is the beginning of the subroutine

```
SUB:                    ;execution of the subroutine itself

        RTS R5          ;the return begins: this causes
```

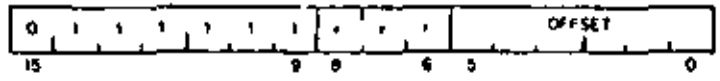
the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents of old PC are placed in R5.

MARK N causes (1) the stack pointer to be adjusted to point to the old R5 value; (2) the value now in R5 (the old PC) to be placed in the PC; and (3) contents of the the old R5 to be popped into R5 thus completing the return from subroutine.

SOB

Used in the PDP-11/34, 11/45 and 11/55

subtract one and branch (if ≠ 0) 077R00 Plus offset



Operation: R ← R −1 if this result ≠ 0 then PC ← PC −(2 × offset)

Condition Codes: unaffected

Description: The register is decremented. If it is not equal to 0, twice the offset is subtracted from the PC (now pointing to the following word). The offset is interpreted as a sixbit positive number. This instruction provides a fast, efficient method of loop control. Assembler syntax is:

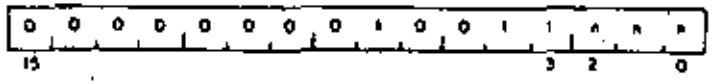
```
SOB     R,A
```

Where A is the address to which transfer is to be made if the decremented R is not equal to 0. Note that the SOB instruction can not be used to transfer control in the forward direction.

SPL

Used in the PDP-11/45 and 11/55

Set Priority Level 00023N



Operation: PS (bits 7-5) ← Priority (priority = n n n)

Condition Codes: not affected

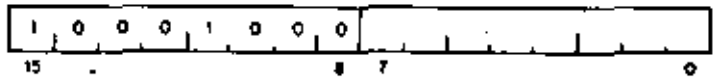
Description The least significant three bits of the instruction are loaded into the Program Status Word (PS) bits 7-5 thus causing a changed priority. The old priority is lost.
Assembler syntax is: SPL N

Note: This instruction is a no op in User and Supervisor modes.

Traps
Trap instructions provide for calls to emulators, I/O monitors, debugging packages, and user-defined interpreters. A trap is effectively an interrupt generated by software. When a trap occurs the contents of the current Program Counter (PC) and Program Status Word (PS) are pushed onto the processor stack and replaced by the contents of a two-word trap vector containing a new PC and new PS. The return sequence from a trap involves executing an RTI or RTT instruction which restores the old PC and old PS by popping them from the stack. Trap vectors are located at permanently assigned fixed addresses.

EMT

emulator trap 104000—104377

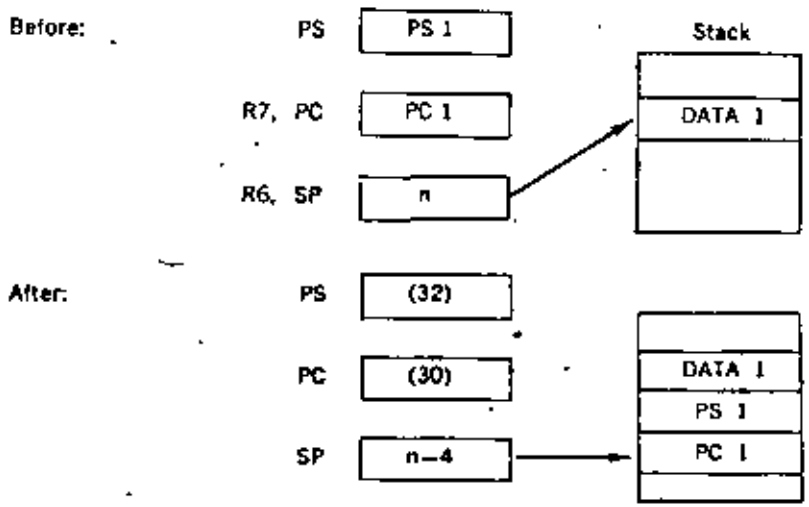


Operation:
↓(SP) ← PS
↓(SP) ← PC
PC ← (30)
PS ← (32)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

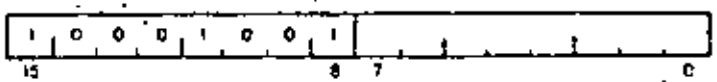
Description: All operation codes from 104000 to 104377 are EMT instructions and may be used to transmit information to the emulating routine (e.g., function to be performed). The trap vector for EMT is at address 30. The new PC is taken from the word at address 30; the new central processor status (PS) is taken from the word at address 32.

Caution: EMT is used frequently by DEC system software and is therefore not recommended for general use.



TRAP

trap 104400—104777



Operation:
↓(SP)←PS
↓(SP)←PC
PC←(34)
PS←(36)

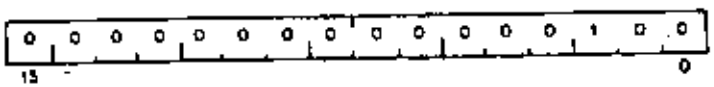
Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:
Operation codes from 104400 to 104777 are TRAP instructions. TRAPs and EMTs are identical in operation, except that the trap vector for TRAP is at address 34.

Note: Since DEC software makes frequent use of EMT, the TRAP instruction is recommended for general use.

IOT

input/output trap 000004



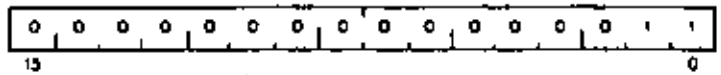
Operation:
↓(SP)←PS
↓(SP)←PC
PC←(20)
PS←(22)

Condition Codes:
N:loaded from trap vector
Z:loaded from trap vector
V:loaded from trap vector
C:loaded from trap vector

Description:
Performs a trap sequence with a trap vector address of 20. Used to call the I/O Executive routine IOX in the paper tape software system, and for error reporting in the Disk Operating System.
(no information is transmitted in the low byte)

BPT

breakpoint trap 000003



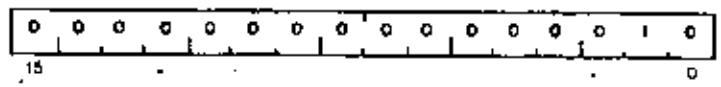
Operation:
↓(SP)←PS
↓(SP)←PC
PC←(14)
PS←(16)

Condition Codes:
N: loaded from trap vector
Z: loaded from trap vector
V: loaded from trap vector
C: loaded from trap vector

Description:
Performs a trap sequence with a trap vector address of 14. Used to call debugging aids. The user is cautioned against employing code 000003 in programs run under these debugging aids.
(no information is transmitted in the low byte.)

RTI

return from interrupt 000002



Operation:
PC←(SP)↑
PS←(SP)↑

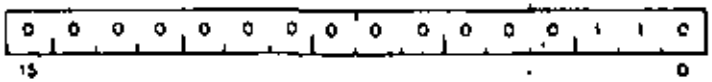
Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description:
Used to exit from an interrupt or TRAP service routine. The PC and PS are restored (popped) from the processor stack

RTT

Used in the PDP-11/34, 11/45 and 11/55

return from interrupt 000006



Operation: $PC \leftarrow (SP)_4$
$PS \leftarrow (SP)_4$

Condition Codes:
N: loaded from processor stack
Z: loaded from processor stack
V: loaded from processor stack
C: loaded from processor stack

Description: This is the same as the RTI instruction except that it inhibits a trace trap. While RTI permits a trace trap if a trace trap is pending, the first instruction after the RTT will be executed prior to the next "T" trap. In the case of the RTI instruction the "T" trap will occur immediately after the RTI.

Reserved Instruction Traps - These are caused by attempts to execute instruction codes reserved for future processor expansion (reserved instructions) or instructions with illegal addressing modes (illegal instructions). Order codes not corresponding to any of the instructions described are considered to be reserved instructions. JMP and JSR with register mode destinations are illegal instructions. Reserved and illegal instruction traps occur as described under EMT, but trap through vectors at addresses 10 and 4 respectively.

Stack Overflow Trap

Bus Error Traps - Bus Error Traps are:

1. Boundary Errors - attempts to reference instructions or word operands at odd addresses.
2. Time-Out Errors - attempts to reference addresses on the bus that made no response within a certain length of time. In general, these are caused by attempts to reference non-existent memory, and attempts to reference non-existent peripheral devices.

Bus error traps cause processor traps through the trap vector address 4.

Trace Trap - Trace Trap enables bit 4 of the PS and causes processor traps at the end of instruction executions. The instruction that is executed after the instruction that set the T-bit will proceed to completion and then cause a processor trap through the trap vector at address 14. Note that the trace trap is a system debugging aid and is transparent to the general programmer.

The following are special cases and are detailed in subsequent paragraphs:

1. The traced instruction cleared the T-bit.
2. The traced instruction set the T-bit.
3. The traced instruction caused an instruction trap.
4. The traced instruction caused a bus error trap.
5. The traced instruction caused a stack overflow trap.
6. The process was interrupted between the time the T-bit was set and the fetching of the instruction that was to be traced.
7. The traced instruction was a WAIT.
8. The traced instruction was a HALT.
9. The traced instruction was a Return from Trap.

Note: The traced instruction is the instruction after the one that sets the T-bit.

An instruction that cleared the T-bit - Upon fetching the traced instruction an internal flag, the trace flag, was set. The trap will still occur at the end of execution of this instruction. The stacked status word, however, will have a clear T bit.

An instruction that set the T-bit - Since the T-bit was already set, setting it again has no effect. The trap will occur.

An instruction that caused an Instruction Trap. The instruction trap is sprung and the entire routine for the service trap is executed. If the service routine exits with an RTI or in any other way restores the stacked status word, the T-bit is set again, the instruction following the traced instruction is executed and, unless it is one of the special cases noted above, a trace trap occurs.

An instruction that caused a Bus Error Trap. This is treated as an Instruction Trap. The only difference is that the error service is not as likely to exit with an RTI, so that the trace trap may not occur.

An instruction that caused a stack overflow. The instruction completes execution as usual—the Stack Overflow does not cause a trap. The Trace Trap Vector is loaded into the PC and PS, and the old PC and PS are pushed onto the stack. Stack Overflow occurs again, and this time the trap is made.

An interrupt between setting of the T-bit and fetch of the traced intruction. The entire interrupt service routine is executed and then the T-bit is set again by the exiting RTI. The traced instruction is executed (if there have been no other interrupts) and, unless it is a special case noted above, causes a trace trap.

Note that interrupts may be acknowledged immediately after the loading of the new PC and PS at the trap vector location. To lock out all interrupts, the PS at the trap vector should raise the processor priority to level 7.

A WAIT. The trap occurs immediately.

A HALT. The processor halts. When the continue key on the console is pressed, the instruction following the HALT is fetched and executed. Unless it is one of the exceptions noted above, the trap occurs immediately following execution.

A Return from Trap. The return from trap instruction either clears or sets the T-bit. It inhibits the trace trap. If the T-bit was set and RTT is the traced instruction the trap is delayed until completion of the next instruction.

Power Failure Trap. is a standard PDP-11 feature. Trap occurs whenever the AC power drops below 95 volts or outside 47 to 63 Hertz. Two milliseconds are then allowed for power down processing. Trap vector for power failure is at locations 24 and 26

Trap priorities, in case multiple processor trap conditions occur simultaneously the following order of priorities is observed (from high to low):

11/04

1. Odd Address
2. Timeout
3. Trap Instructions
4. Trace Trap
5. Power Failure

11/34

1. Odd Address
2. Memory Management Violation
3. Timeout
4. Parity Error
5. Trap Instruction
6. Trace Trap
7. Stack Overflow
8. Power Fail
9. Interrupt
10. HALT From Console

11/45, 11/55

1. Odd Address
2. Fatal Stack Violation
3. Segment Violation
4. Timeout
5. Parity Error
6. Console Flag
7. Segment Management Trap
8. Warning Stack Violation
9. Power Failure

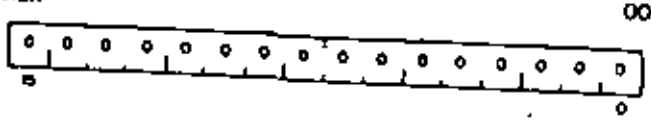
The details on the trace trap process have been described in the trace trap operational description which includes cases in which an instruction being traced causes a bus error, instruction trap, or a stack overflow trap.

If a bus error is caused by the trap process handling instruction traps, trace traps, stack overflow traps, or a previous bus error, the processor is halted.

If a stack overflow is caused by the trap process in handling bus errors, instruction traps, or trace traps, the process is completed and then the stack overflow trap is sprung.

HALT

halt 000000



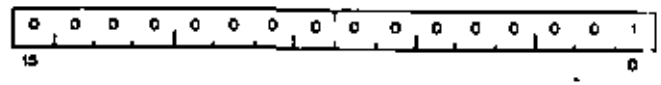
Condition Codes: not affected

Description: Causes the processor operation to cease. The console is given control of the bus. The console data lights display the contents of R0; the console address lights display the address after the halt instruction. Transfers on the UNIBUS are terminated immediately. The PC points to the next instruction to be executed. Pressing the continue key on the console causes processor operation to resume. No INIT signal is given.

Note: A halt issued in a trap

WAIT

wait for interrupt 000001

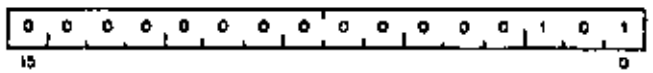


Condition Codes: not affected

Description: Provides a way for the processor to relinquish use of the bus while it waits for an external interrupt. Having been given a WAIT command, the processor will not compete for bus use by fetching instructions or operands from memory. This permits higher transfer rates between a device and memory, since no processor-induced latencies will be encountered by bus requests from the device. In WAIT, as in all instructions, the PC points to the next instruction following the WAIT operation. Thus when an interrupt causes the PC and PS to be pushed onto the processor stack, the address of the next instruction following the WAIT is saved. The exit from the interrupt routine (i.e. execution of an RTI instruction) will cause resumption of the interrupted process at the instruction following the WAIT.

RESET

reset external bus 000005



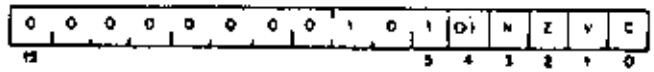
Condition Codes: not affected

Description: Sends INIT on the UNIBUS. All devices on the UNIBUS are reset to their state at power up.

Condition Code Operators

CLN SEN
CLZ SEZ
CLV SEV
CLC SEC
CCC SCC

condition code operators 0002XX



Description: Set and clear condition code bits. Selectable combinations of these bits may be cleared or set together. Condition code bits corresponding to bits in the condition code operator (Bits 0-3) are modified according to the sense of bit 4, the set/clear bit of the operator, i.e. set the bit specified by bit 0, 1, 2 or 3, if bit 4 is a 1. Clear corresponding bits if bit 4 = 0.

Mnemonic	Operation	OP Code
CLC	Clear C	000241
CLV	Clear V	000242
CLZ	Clear Z	000244
CLN	Clear N	000250
SEC	Set C	000261
SEV	Set V	000262
SEZ	Set Z	000264
SEN	Set N	000270
SCC	Set all CC's	000277
CCC	Clear all CC's	000257
	Clear V and C	000243
NOP	No Operation	000240

CC
CC

Combinations of the above set or clear operations may be ORed together to form combined instructions

centro de educación continua
división de estudios de posgrado
facultad de ingeniería - unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

MANEJO DE SUBRUTINAS

ING. LUIS CORDERO BORBOA

JUNIO, 1980

MINICOMPUTER SYSTEMS

R. H. ECKHOUSE, JR.

PROGRAMMING TECHNIQUES

Mastery of a basic instruction set is the first step in learning to program. The next step is to learn to use the instruction set to obtain correct results and to obtain them efficiently. This is best done by studying the following programming techniques. Examples, which should further familiarize the reader with the total instruction set and its use, are given to illustrate each technique.

4.1. POSITION-INDEPENDENT PROGRAMMING

Most programs written to run on a computer are written so as to occupy specified memory locations (e.g., the current location counter is used to define the location of the first instruction). Such programs are said to be absolute or *position-dependent programs*. However, it is sometimes desirable to have a standard program which is available to many different users. Since it will not be known a priori where the standard programs are to be loaded, it is necessary to be able to load the program into different areas of core and to run it there. There are several ways to do this:

1. Reassemble the program at the desired location.
2. Use a relocating loader which accepts specially coded binary from a relocatable assembler.
3. Have the program relocate itself after it is loaded.
4. Write a program that is *position-independent*.

On small machines, reassembly is often performed. When the required core is available, a relocating loader (usually called a *linking loader*) is

preferable. It generally is not economical to have a program relocate itself, since hundreds or thousands of addresses may need adjustment. Writing position-independent code is usually not possible because of the structure of the addressing of the object machine. However, on the PDP-11, position-independent code (PIC) is possible.

PIC is achieved on the PDP-11 by using addressing modes which form an effective memory address relative to the program counter (PC). Thus, if an instruction and its object(s) are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus PIC usually references locations relative to the current location. PIC programs may make absolute references as long as the locations referenced stay in the same place while the PIC program is relocated.

4.1.1. Position-Independent Modes

There are three position-independent modes or forms of instructions. They are:

1. *Branches*: the conditional branches, as well as the unconditional branch, BR, are position-independent, since the branch address is computed as an offset to the PC.

2. *Relative memory references*: any relative memory reference of the form

```
CLR     X
MOV     X, Y
ER      Y
```

is position-independent because the assembler assembles it as an offset indexed by the PC. The offset is the difference between the referenced location and the PC. For example, assume that the instruction CLR 200 is at address 100:

Line Number	Address	Contents	Symbolic Instruction	Comments
1	000100	005067 000074	CLR 200	;FIRST WORD OF INSTRUCTION ;OFFSET=200−104

The offset is added to the PC. The PC contains 104, which is the address of the word following the offset (the second word of this two-word instruction). Note that although the form CLR X is position-independent, the form CLR @X is not. We may see this when we consider the following:

Line Number	Address	Contents	Label	Symbolic Instruction	Comments
1	001000	005077 000774	S:	CLR @X	;CLEAR LOCATION A
2.	002000	003000	X:	.WORD A	;POINTER TO A
3.	003000	000000	A:	.WORD 0	

The contents of location X are used as the address of the operand, which is symbolically labeled A. The value stored at location X is the absolute address of the symbolic location A rather than the relative address or offset between location X and A. Thus, if all the code is relocated after assembly, the contents of location X must be altered to reflect the fact that location A now stands for a new absolute address.† If A, however, was the name associated with a fixed, absolute location, statements S and X could be relocated because now it is important for A to remain fixed. Thus the following code is position-independent:

Line Number	Address	Contents	Label	Symbolic Instruction	Comments
1		000036		A = 36	;FIXED ADDRESS OF 36
2	001000	005077 000774	S:	CLR @X	;CLEAR LOCATION A
3	002000	000036	X:	.WORD A	;POINTER TO A

3. *Immediate operands*: the assembler addressing form #X specifies immediate data; that is, the operand is in the instruction. Immediate data that are not addresses are position-independent, since they are a part of the instruction and are moved with the instruction. Consequently, a SUB #2,HERE is position-independent (since #2 is not an address), while MOV #A,ADRPTR is position-dependent if A is a symbolic address. This is so even though the operand is fetched, in both cases, using the PC in the autoincrement

†To verify this point the reader is encouraged to relocate the code, after assembly, into locations 4000, 5000, and 6000. By doing so he will discover that the contents of these locations are the same as for the original code and that the contents of location 5000 do not poir ocation 6000.

mode, since it is the quantity fetched that is being used rather than its form of addressing.

4.1.2. Absolute Modes

Any time a memory location or register is used as a pointer to data, the reference is absolute. If the referenced data remain always fixed in memory (e.g., an absolute memory location) independent of the position of the PIC, the absolute modes must be used.† Alternatively, if the data are relative to the position of the code, the absolute modes must not be used unless the pointers involved are modified. Restating this point in different words, if addressing is direct and relative, it is position-independent; if it is indirect and either relative or absolute, it is *not* position-independent. For example, the instruction

```
MOV     @#X,HERE
```

"move the contents of the word pointed to (indirectly referenced by) the PC (in this case absolute location X) to the word indexed relative to the PC (symbolically called HERE)" contains one operand that is referenced indirectly (X) and one operand that is referenced relatively (HERE). This instruction can be moved anywhere in memory as long as absolute location X stays the same, that is, it does not move with the instruction or program; otherwise it may not be.

The absolute modes are:

@X	Location X is a pointer.
@#X	The immediate word is a pointer.
(R)	The register is a pointer.
(R)+ and (R)	The register is a pointer.
@(R)+ and @—(R)	The register points to a pointer.
X(R) R≠6 or 7	The base, X, modified by (R), is the address of the operand.
@X(R)	The base, modified by (R), is a pointer.

The nondeferred index modes require a little clarification. As described in Chapter 3, the form X(7)†† is the normal mode in which to reference memory and is a relative mode. Index mode, using a register, is also a relative mode and may be used conveniently in PIC. Basically, the register pointer points to a dynamic storage area, and the index mode is used to access data relative to the pointer. Once the pointer is set up, all data are referenced relative to the pointer.

†When PIC is not being written, references to fixed locations may be performed with either the absolute or relative forms.

††Recall that X(7) is equivalent to X(R7), which is equivalent to ' ' where PC=R7.

4.1.3. Writing Automatic PIC

Automatic PIC is code that requires no alteration of addresses or pointers. Thus memory references are limited to relative modes unless the location referenced is fixed. In addition to the above rules, the following must be observed:

1. Start the program with .=0 to allow easy relocation using the absolute loader (see Chapter 7).

2. All location-setting statements must be of the form .=.±X or .= function of symbols within the PIC. For example, .=A+10, where A is a local label.

3. There must not be any absolute location-setting statements. This means that a block of PIC cannot set up specified core areas at load time with statements such as

```
        .=340
        .WORD   TRAPH,340       ;PRE-LOAD 340,342
```

The absolute loader, when it is relocating PIC, relocates all data by the load bias (see Chapter 7). Thus the data for the absolute location would be relocated to some other place. Such areas must be set at execution time:

```
        MOV     #TRAPH,@#340    ;PUT ADDR IN RES LOC 340
        MOV     #340,@#342      ;AND RES LOCATION 342
```

4.1.4. Writing Nonautomatic PIC

Often it is not possible or economical to write totally automated PIC. In these cases some relocation may be easily performed at execution time. Some of the required methods of solution are presented below. Basically, the methods operate by examining the PC to determine where the PIC is actually located. Then a relocation factor can be easily computed. In all examples it is assumed that the code is assembled at zero and has been relocated somewhere else by the absolute loader.

4.1.5. Setting Up Fixed Core Locations

Consider first the previous example to clear the contents of A indirectly. The pointer to A, contained in symbolic location X, must be changed if the code is to be relocated. The program segment in Fig. 4-1 recomputes the pointer value each time that it is executed. Thus the pointer value no longer depends on the value of the location counter at the time the program was assembled, but on the value of the PC where it is loaded.


```
                000000          R0=%0                   ;DEFINE R0
                000007          PC=%7                   ;DEFINE PC
000000 016700 000000 S:         MOV     PC,R0           ;R0 = (ADDR OF S)+2
000004 062700                   ADD     #A-S-2,R0       ;ADD IN OFFSET
        001770
000010 010067                   MOV     R0,X            ;MOVE POINTER TO X
        000766
000014 005077                   CLR     @X              ;CLEAR VALUE INDIRECTLY
        000762
000020 000000                   HALT                    ;STOP
                                ;
                                ;
                                ;
                001000          .=.+700
001000 002000 X:                .WORD   A               ;POINTER TO A
                                ;
                                ;
                                ;
                002000          .=.+770
002000 000000 A:                .WORD   0               ;VALUE TO BE CLEARED
                000001          .END
```

Fig. 4-1

Now if this program is loaded into locations 4000 and higher, it should be clear that none of the program values is changed. This point could be shown pictorially by taking the Fig. 4-1 material, recopying it, but changing only the values in the leftmost column, the address column. Thus if one were to look in, say, location 4010, the contents would be 766 and the value found in location 5000 would be 2000 (i.e., neither value is changed).

Given that the program data have not changed, the question is: How does it work? The answer is that the offset A−S−2 is equivalent to A−(S+2) and S+2 is the value of PC which is placed in R0 by the statement MOV PC,R0. At assembly time the offset value is $A-PC_0$, where $PC_0 = S+2$ and PC_0 is the PC that was assumed for the program when assembled beginning at location 0.

Later, after the program has been relocated, the move instruction will no longer store PC_0 in R0, but a new value, PC_n, which is the current value of PC for the executing program. However, the add instruction still adds in the immediate value $A-PC_0$, producing the final result in R0:

$$PC_n+(A-PC_0) = A+(PC_n-PC_0)$$

which is the desired value, since it yields the new absolute location of A [e.g., the assembled value of A plus the relocation factor (PC_n-PC_0)].

4.1.6. Relocating Pointers

If pointers must be used, they may be relocated as we have just shown. For example, assume that a list of data is to be accessed with the instruction

7

```
        ADD     (R0)+,R1
```

The pointer to the list, list L, may be calculated at execution time as follows:

```
M:      MOV     PC,R0           ;GET CURRENT PC
        ADD     #L-M-2,R0       ;ADD OFFSET
```

Another variation is to gather all pointers into a table. The relocation factor may be calculated once and then applied to all pointers in the table in a loop. The program in Fig. 4-2 is an example of this technique. The reader should verify (Exercise 1 at the end of this chapter) that if this program is relocated so that if it begins in location 10000, the values in the pointer table, PTRTBL, will be 10000, 10020, and 10030.

```
        000000          R0=%0                   ;DEFINE R0
        000001          R1=%1                   ;DEFINE R1
        000002          R2=%2                   ;DEFINE R2
        000007          PC=%7                   ;DEFINE PC
000000  010700  X:      MOV     PC,R0           ;RELOCATE ALL ENTRIES IN PTRTBL
000002  162700          SUB     #X+2,R0         ;CALCULATE RELOCATION FACTOR
        000002
000006  012701          MOV     #PTRTBL,R1      ;GET AND RELOCATE A POINTER
        000010
000012  060001          ADD     R0,R1           ;TO PTRTBL
000014  012702          MOV     #TBLLEN,R2      ;GET LENGTH OF TABLE
        000003
000020  060021  LOOP:   ADD     R0,(R1)+        ;RELOCATE AN ENTRY
000022  005302          DEC     R2              ;COUNT DOWN
000024  001375          BNE     LOOP            ;BRANCH IF NOT DONE
000026  000000          HALT                    ;STOP WHEN DONE
        000003          TBLLEN=3                ;LENGTH OF TABLE
000030  000000  PTRTBL: .WORD   X,LOOP,PTRTBL
000032  000020
000034  000030
        000001          .END
```

Fig. 4-2

Care must be exercised when restarting a program that relocates a table of pointers. The restart procedure must not include the relocating again (i.e., the table must be relocated exactly once after each load).

4.2. JUMP INSTRUCTION

Although mentioned earlier, the JMP instruction has been overlooked somewhat up to now. The astute reader will, no doubt, recognize that the necessity of a jump instruction is dictated by the fact that the branch instructions, although relative, are incapable of branching more than 200 words in either a positive or a negative direction. Thus to branch from one end of

memory to another, a jump instruction must be a part of the instruction set and must allow full-word addressing.

The jump instruction is indeed a part of the PDP-11 instruction set and belongs to the single-operand group. As a result, jumps may be relative, absolute, indirect, and indexed. This flexibility in determining the effective jump address is quite useful in solving a particular class of problems that occur in programming. This class is best illustrated by example.

4.2.1. Jump Table Problem

A common type of problem is one in which the input data represent a code for an action to be performed. For each code, the program is to take a certain action by executing a specified block of code. Such a problem would be coded in FORTRAN as

```
      .
      .
      .
      READ, INDEX
      GO TO (10,100,37,1150,...,7), INDEX
      .
      .
      .
```

In other words, based on the value of index, the program will go to the statement labeled 10, 100, 37, and so on.

The "computed GO TO" in FORTRAN must eventually be translated into machine language. One possibility in the language of the PDP-11 would be

```
        .
        .
        .
        READ    INDEX           ;A PSEUDO-INSTRUCTION
        MOV     INDEX,R1        ;PLACE IT IN R1
        DEC     R1              ;0<=INDEX<=MAX-1
        ADD     R1,R1           ;FORM 2*INDEX
        JMP     @TABLE(R1)      ;INDIRECT JUMP
TABLE:  .WORD   L10,L100,L37,L1150,...,L7
        .
        .
        .
```

The method used is called the *jump table method*, since it uses a table of addresses to jump to. The method works as follows:

1. The value of INDEX is obtained.

2. Since the range of INDEX is $1 \leq \text{INDEX} \leq \text{maximum value}$, 1 is subtracted from the index so that its range is $0 \leq \text{INDEX} \leq \text{max} - 1$.

3. The value of index is doubled to take care of the fact that labels in the table are stored in even addresses; i.e., full words;

4. The address for the JMP instruction is utilized both as indexed and indirect, such that it points to an address to be jumped to in the table.

Although the jump instruction transfers control to the correct program label, it does not specify any way to come back. In the next section, where we shall consider subroutining, we shall see that a slight modification of the jump instructions allows for an orderly transfer of control, and a return, from one section of code to another.

4.3. SUBROUTINES

A good programming practice to get into is to separate large programs into smaller *subprograms*, which are easier to manage. These subprograms are activated either by a main program or by each other, allowing for the sharing of routines among the different programs and subprograms.

The saving in memory space resulting from having only one copy of the needed routine is a definite advantage. Equally important is the saving in time for the programmer, who needs to code the routine only once. However, in order to share common subprograms, there must be a mechanism to

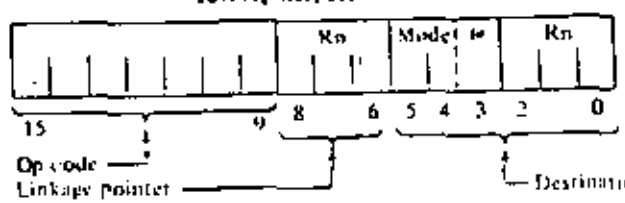
1. Allow the transfer of control from one routine to another.

2. Pass values among the various routines.

The mechanism that accomplishes these requirements is called the *subroutine linkage* and is, in general, a combination of hardware features and software conventions.

The hardware features on the PDP-11 which assist in performing the subroutine linkage are the instructions JSR and RTS. These instructions are in the subroutine call and return group and have the following assembler form and instruction format†:

JSR register, destination



†Depending on the mode of addressing, one or two words are used for the JSR instruction.

Both instructions make use of a "stack" mechanism similar to the stack mechanism described for zero-address machines in Section 1.2.8.6.

4.3.1. Stack

A *stack* is an area of memory set aside by the programmer for temporary storage or subroutine/interrupt service linkage. The instructions that facilitate stack handling (e.g., autoincrement and autodecrement) are useful features that may be found in low-cost computers. They allow a program to dynamically establish, modify, or delete a stack and items on it. The stack uses the *last-in, first-out* or *LIFO concept*; that is, various items may be added to a stack in sequential order and retrieved or deleted from the stack in reverse order (Fig. 4-3). On the PDP-11, a stack starts at the highest location reserved for it and expands linearly downward to the lowest address as items are added to the stack.

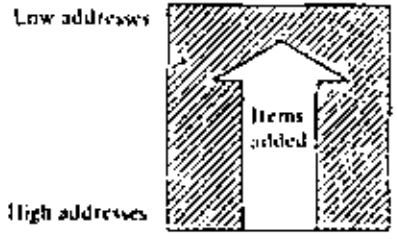


Fig. 4-3 Stack addresses.

The programmer does not need to keep track of the actual locations his data are being stacked into. This is done automatically through a *stack pointer*. To keep track of the last item added to the stack (or "where we are" in the stack), a general register always contains the memory address where the last item is stored in the stack. In the PDP-11 any register except register 7 (the PC) may be used as a stack pointer under program control; however, instructions associated with subroutine linkage and interrupt service automatically use register 6 (R6) as a hardware stack pointer. For this reason R6 is frequently referred to as the system *SP*.

Stacks in the PDP-11 may be maintained in either full-word or byte units. This is true for a stack pointed to by any register except R6, which must be

organized in full-word units only. Byte stacks (Fig. 4-4) require instructions capable of operating on bytes rather than full words (byte handling is discussed in Section 4.6).

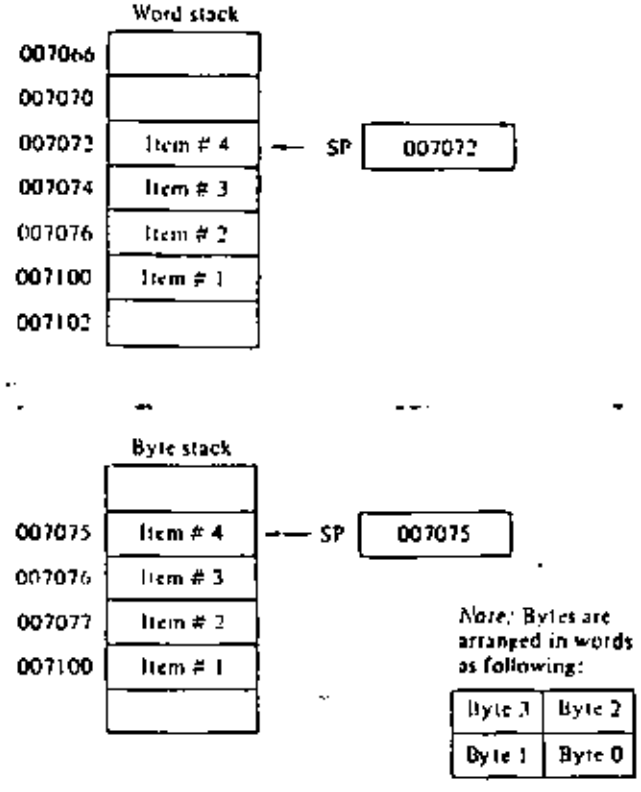


Fig. 4-4 Word and byte stacks.

Items are added to a stack using the autodecrement addressing mode with the appropriate pointer register. (See Chapter 2 for a description of the autoincrement/decrement modes.)

This operation is accomplished as follows:

```
MOV     SOURCE,-(SP)     ;MOVE SOURCE WORD ONTO THE STACK
```

or

```
MOVB    SOURCE,-(SP)     ;MOVE SOURCE BYTE ONTO THE STACK
```

This is called a "push" because data are "pushed onto the stack."

†See Section 4.6 for a discussion of byte instructions.

To remove an item from stack the autoincrement addressing mode with the appropriate SP is employed. This is accomplished in the following manner:

```
MOV     (SP)+,DEST      ;MOVE DESTINATION WORD OFF STACK
```

or

```
MOVB    (SP)+,DEST      ;MOVE DESTINATION BYTE OFF STACK
```

Removing an item from a stack is called a *pop*, for "popping from the stack." After an item has been popped, its stack location is considered free and available for other use. The stack pointer points to the last-used location, implying that the next (lower) location is free. Thus a stack may represent a pool of shareable temporary storage locations.

4.3.2. Subroutine Calls and Returns

When a JSR is executed, the contents of the linkage register are saved on the system R6 stack as if a MOV reg,—(SP) has been performed. Then the same register is loaded with the memory address following the JSR instruction (the contents of the current PC) and a jump is made to the entry location specified. The effect, then, of executing one JSR instruction is the same as simultaneously executing two MOVs and a JMP; for example,

```
                MOV REG,-(SP)   ;PUSH REGISTER INTO THE STACK
JSR  REG,SUBR   MOV PC,REG      ;PUT RETURN PC INTO REGISTER
                JMP SUBR        ;JUMP TO SUBROUTINE
```

Figure 4-5 gives the "before" and after conditions when executing the subroutine instruction JSR R5,1064.

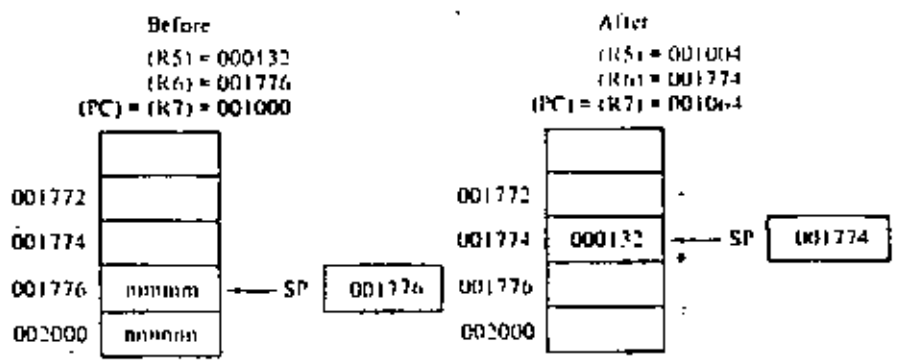


Fig. 4-5 JSR instruction.

In order to return from a subroutine, the RTS instruction is executed. It performs the inverse operation of the JSR, the unstacking and restoring of the saved register value, and the return of control to the instruction following the JSR instruction. The equivalent of an RTS is a concurrent MOV instruction pair:

```
RTS  REG        MOV REG,PC      ;RESTORE PC
                MOV (SP)+,REG   ;RESTORE REGISTER
```

The use of a stack mechanism for subroutine calls and returns is particularly advantageous for two reasons. First, many JSR instructions can be executed without the need to provide any saving procedure for the linkage information, since all linkage information is automatically pushed into the stack in sequential order. Returns can simply be made by automatically popping this information from the stack in opposite order. Such linkage address bookkeeping is called automatic *nesting* of subroutine calls. This feature enables the programmer to construct fast, efficient linkages in an easy, flexible manner. It even permits a routine to be recalled or to call itself in those cases where this is meaningful (Sections 4.3.5 and 4.3.6). Other ramifications will appear after we examine the interrupt mechanism for the PDP-11 (Section 6.4).

The second advantage of the stack mechanism is found in its ease of use for saving and restoring registers. This case arises when a subroutine wants to use the general registers, but these registers were already in use by the calling program and must therefore be returned to it with their contents intact. The called subroutine (JSRPC, SUBR) could be written, then, as shown in Fig. 4-6.

```
SUBR:   MOV     R1,TEMPS        ;SAVE R1
        MOV     R2,TEMPS+2      ;SAVE R2
        .
        .
        .
        MOV     TEMPS+2,R2      ;RESTORE R2
        MOV     TEMPS,R1        ;RESTORE R1
        RTS     PC              ;RETURN
TEMPS:  .WORD   0,0,0,0,0,0,0   ;SAVE AREA
```

or using the stack as

```
SUBR:   MOV     R1,-(R6)        ;PUSH R1
        MOV     R2,-(R6)        ;PUSH R2
        .
        .
        .
        MOV     (R6)+,R2        ;POP R2
        MOV     (R6)+,R1        ;POP R1
        RTS     PC              ;RETURN
```

Fig. 4-6 Saving and restoring registers using the stack.

The second routine uses two fewer words per register save/restore and allows another routine to use the temporary stack storage at a latter point rather than permanently tying some memory locations (TEMPS) to a particular routine. This ability to share temporary storage in the form of a stack is a very economical way to save on memory usage, especially when the total amount of memory is limited.

The reader should note that the subroutine call JSR PC,SUBR is a legitimate form for a subroutine jump. The instruction does not utilize or stack any registers but the PC. On the other hand, the instruction JSR SP,SUBR, where SP = R6, is not normally considered a meaningful combination. Later, however, utilizing register 6 will be considered (see Section 4.3.7).

4.3.3. Argument Transmission

The JSR and RTS instructions handle the linkage problem for transferring control. What remains is the problem of passing arguments back and forth to the subroutine during its invocation. As it turns out, this is a fairly straightforward problem, and the real question becomes one of choosing one solution from the large number of ways for passing values.

A very simple-minded approach for argument transmission would be to agree ahead of time on the locations that might be used. For example, suppose that there exists a subroutine MUL which multiplies two 16-bit words together, producing a 32-bit result. The subroutine expects the multiplier and multiplicand to be placed in symbolic locations ARG1 and ARG2 respectively, and upon completion, the subroutine will leave the resultant in the same locations.

The subroutine linkage needed to set up, call, and save the generated results might look like:

```
MOV     X,ARG1          ;MULTIPLIER
MOV     Y,ARG2          ;MULTIPLICAND
JSR     PC,MUL          ;CALL MULTIPLY
MOV     ARG1,RSLT       ;SAVE THE TWO
MOV     ARG2,RSLT+2     ;  WORD RESULT
```

As an alternative to this linkage, one could use the registers for the subroutine arguments and write:

```
MOV     X,R1            ;MULTIPLIER
MOV     Y,R2            ;MULTIPLICAND
JSR     PC,MUL          ;CALL MULTIPLY
...
```

This last method, although acceptable, is somewhat restricted in that a maximum of six arguments could be transmitted, corresponding to the number of general registers available. As a result of this restriction, another alternative is used which makes use of the memory locations pointed to by the linkage register of the JSR instruction. Since this register points to the first word following the JSR instruction, it may be used as a pointer to the first word of a vector of arguments or argument addresses.

Considering the first case where the arguments follow the JSR instruction, the subroutine linkage would be of the form:

```
JSR     R0,MUL          ;CALL MULTIPLY
.WORD   XVALUE,YVALUE   ;ARGUMENTS
```

These arguments could be accessed using autoincrement mode:

```
MUL:    MOV     (R0)+,R1        ;GET MULTIPLIER
        MOV     (R0)+,R2        ;GET MULTIPLICAND
        .
        .
        .
        RTS     R0              ;RETURN
```

At the time of return, the value (address pointer) in R0 will have been incremented by 4 so that R0 contains the address of the next executable instruction following the JSR.

In the second case, where the addresses of the arguments follow the subroutine call, the linkage looks like

```
JSR     R0,MUL          ;CALL MULTIPLY
.WORD   XADDR,YADDR     ;ARGUMENTS
```

For this case, the values to be manipulated are fetched indirectly:

```
MUL:    MOV     @(R0)+,R1       ;FETCH MULTIPLIER
        MOV     @(R0)+,R2       ;FETCH MULTIPLICAND
        .
        .
        .
        RTS     R0              ;RETURN
```

Another method of transmitting arguments is to transmit only the address of the first item by placing this address in a general-purpose register. It is not necessary to have the actual argument list in the same general area as the subroutine call. Thus a subroutine can be called to work on data located anywhere in memory. In fact, in many cases, the operations performed by the subroutine can be applied directly to the data located on or pointed to by a stack (Fig. 4-7) without ever actually needing to move these data into the subroutine area.

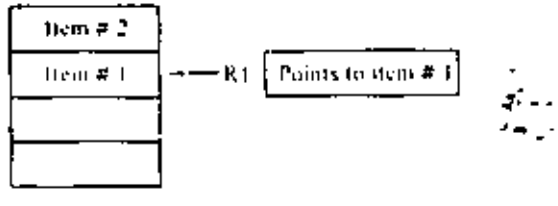


Fig. 4-7 Transmitting stacks as arguments.

Calling program:

```
        MOV     #POINTER,R1     ;SET UP POINTER
        JSR     PC,SUBR         ;CALL SUBROUTINE
```

Subroutine:

```
        ADD     (R1)+,(R1)      ;ADD ITEM #1 TO ITEM #2
                                ;PLACE RESULT IN ITEM #2  R1
                                ;POINTS TO ITEM #2 NOW.
        ...
```

or

```
        ADD     (R1),2(R1)      ;SAME EFFECT AS ABOVE EXCEPT
                                ;THAT R1 STILL POINTS TO
                                ;ITEM #1
        ...
```

Given these many ways to pass arguments to a subroutine, it is worthwhile to ask, why have so many been presented and what is the rationale for presenting them all? The answer is that each method was presented as being somewhat "better" than the last, in that

1. Few registers were used to transmit arguments.
2. The number of parameters passed could be quite large.
3. The linkage mechanism was simplified to the point where only the address of the subroutine was needed to transfer control and pass parameters.

Point 3 requires some additional explanation. Since subroutines, like any other programs, may be written in position-independent code, it is possible to write and assemble them independently from the main program that uses them. The problem is filling in the appropriate address for the JSR instruction.

Filling in the address field in the JSR instruction is the job of the linking loader, since it can not only relocate PIC programs but also fill in subroutine addresses, i.e., *link* them together. The result is that a relocatable subroutine may be loaded anywhere in memory and be linked with one or more calling programs and/or subprograms. There will be only one copy of the routine, but it may be used in a repetitive manner by other programs located anywhere else in memory.

Another point not to be overlooked in recapping argument passing is the significant difference in the methods used. The first techniques presented used the simple method of passing a *value* to the subroutine. The later techniques passed the *address* of the value. The difference in these two techniques, *call by value* and *call by address*, can be quite important, as illustrated by the following FORTRAN-like program example:


```
PROGRAM TRICKY          SUBROUTINE SWAP(X,Y)
A=1.                    TEMP=X
B=2.                    X=Y
PRINT,A-B               Y=TEMP
CALL SWAP(1.,2.)        RETURN
A=1.                    END
B=2.
PRINT,A-B
END
```

If the real constants are passed in by value, both print statements will print out a −1. This occurs because subroutine SWAP interchanges the values that it has received, not the actual contents of the arguments themselves.

However, if the real constants are passed in by address, the two print statements will produce −1. and 1., respectively. In this case the subroutine SWAP references to real constants themselves, interchanging the actual argument values.

Higher-level language, such as FORTRAN, can pass parameters both by value and by address. Often the normal mode is by address, but when the argument is an expression, the address represents the location of the evaluated expression. Therefore, if one wished to call SWAP by value, it could be performed as

```
CALL SWAP(1.*1.,2.-0.)
```

causing the contents of the expressions, but not the constants themselves, to be switched.

These techniques for passing parameters are easy to understand at the assembly language level because the programmer can see exactly what method is being used. In higher-level languages, however, where the technique is not so transparent, interesting results can occur. Thus the knowledgeable higher-level language programmer must be aware of the techniques used if he is to avoid unusual or unexpected results.

4.3.4. Subroutine Register Usage

A subroutine, like any other program, will use the registers during its execution. As a result, the contents of the registers at the time that the subroutine is invoked may not be the same as when the subroutine returns. The sharing of these common resources (e.g., the registers) therefore dictates that on entry to the subroutine the registers be saved and, on exit, restored.

The responsibility for performing the save and restore function falls either on the calling routine or the called routine. Although arguments exist for making the calling program save the registers (since it need save only the ones in current use), it is more common for the subroutine itself to save and

restore all registers used. On the PDP-11 the save and restore routine is greatly simplified by the use of a stack, as was illustrated in Fig. 4-6.

As pointed out previously, stacks grow downward in memory and are traditionally defined to occupy the memory space immediately preceding the program(s) that use them. One of the first things that any program which uses a stack (in particular one that executes a JSR) must do is to set the stack pointer up. For example, if SP (i.e., R6) is to be used, the program should begin with

```
                                ;BEG IS THE FIRST
                                ;INSTRUCTION OF THE PROGRAM
BEG:    MOV     PC,SP           ;SP=ADDR BEG+2
        TST     -(SP)           ;DECREMENT SP BY 2
                                ;A PUSH ONTO THE STACK WILL
                                ;STORE THE DATA AT BEG-2
```

This initialization routine is written in PIC form, and had it been assembled beginning at location 0 (.=0), the program could be easily relocated. The routine uses a programming trick to decrement the state: It uses the test instruction in autodecrement mode and ignores the setting of the condition codes. The alternative to using the TST instruction would be to SUB L2,SP, but this would require an extra instruction word.

4.3.5. Reentrancy

Further advantages of stack organization become apparent in complex situations which can arise in program systems that are engaged in the concurrent handling of several tasks. Such multitask program environments may range from relatively simple single-user applications which must manage an intermix of I/O service and background computation to large complex multiprogramming systems that manage a very intricate mixture of executive and multiuser programming situations. In all these applications there is a need for flexibility and time/memory economy. The use of the stack provides this economy and flexibility by providing a method for allowing many tasks to use a single copy of the same routine and a simple, unambiguous method for keeping track of complex program linkages.

The ability to share a single copy of a given program among users or tasks is called *reentrancy*. Reentrant program routines differ from ordinary subroutines in that it is unnecessary for reentrant routines to finish processing a given task before they can be used by another task. Multiple tasks can be in various stages of completion in the same routine at any time. Thus the situation shown in Fig. 4-8 may occur.

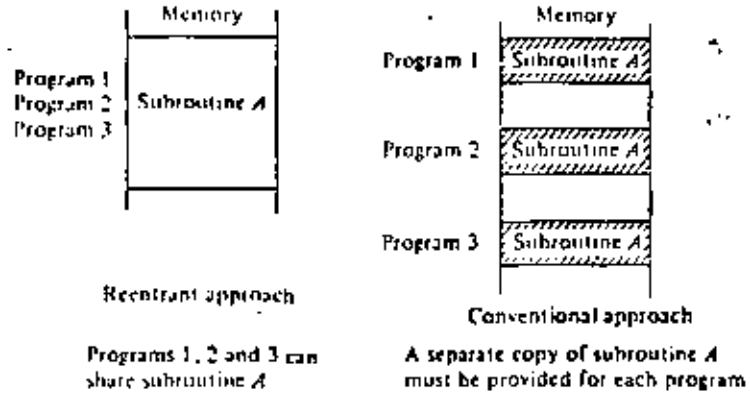


Fig. 4-8 Reentrant routines.

The chief programming distinction between a nonshareable routine and a reentrant routine is that the reentrant routine is composed solely of *pure code*; that is, it contains only instructions and constants. Thus a section of program code is reentrant (shareable) if and only if it is non-self-modifying; that is, no information within it is subject to modification. The philosophy behind pure code is actually not limited to reentrant routines. Any nonmodifying program segment that has no temporary storage or data associated with it will be

1. Simpler to debug.
2. Read-only protectable (i.e., it can be kept in read-only memory).
3. Interruptable and restartable, besides being reentrant.

Using reentrant routines, control of a given routine may be shared as illustrated in Fig. 4-9.

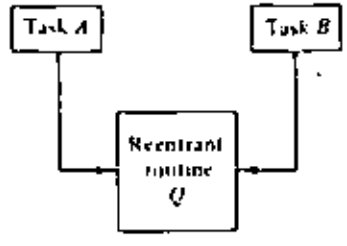


Fig. 4-9 Reentrant routine sharing.

1. Task A has requested processing by reentrant routine Q.
2. Task A temporarily relinquishes control of reentrant routine Q (i.e., is interrupted) before it finishes processing.

3. Task B starts processing in the same copy of reentrant routine Q.

4. Task B relinquishes control of reentrant routine Q at some point in its processing.

5. Task A regains control of reentrant routine Q and resumes processing from where it stopped.

The use of reentrant programming allows many tasks to share frequently used routines such as device service routines and ASCII-Binary conversion routines. In fact, in a multiuser system it is possible, for instance, to construct a reentrant FORTRAN compiler that can be used as a single copy by many user programs.

4.3.6. Recursion

It is often meaningful for a program segment to call itself. The ability to nest subroutine calls to the same subroutine is called *self-reentrancy* or *recursion*. The use of a stack organization permits easy unambiguous recursion. The technique of recursion is of great use to the mathematical analyst, as it also permits the evaluation of some otherwise noncomputable mathematical functions. This technique often permits very significant memory and speed economies in the linguistic operations of compilers and other higher-level software programs, as we shall illustrate.

A classical example of the technique of recursion can be found in computing N factorial ($N!$). Although

$$N! = N * (N - 1) * (N - 2) * \cdots * 1$$

it is also true that

$$N! = N * (N - 1)!$$

$$1! = 1$$

Written in "pseudo-FORTRAN," a function for calculating $N!$ would look like:

```
      INTEGER FUNCTION FACT(N)
      IF (N .NE. 1) GO TO 1
      FACT=1
      RETURN
1     FACT=N*FAC(N-1)
      RETURN
      END
```

This code is pseudo-FORTRAN because it cannot actually be translated by most FORTRAN compilers; the problem is that the recursive call requires

a stack capable of maintaining both the current values of FACT and the return pointers either to the function itself or its calling program. However, the function may be coded in PDP-11 assembly language in a simple fashion by taking advantage of its stack mechanism. Assuming that the value of N is in R0 and the value of $N!$ is to be left in R1, the function FACT could be coded recursively as shown in Fig. 4-10.

```
FACT:   TST     R0              ;IS R0=0?
        BEQ     EXIT            ;YES
        MOV     R0,-(SP)        ;SAVE N
        DEC     R0              ;TRY N-1
        JSR     PC,FACT         ;COMPUTE (N-1)!
RET:    MOV     (SP)+,R1        ;FETCH FROM STACK
        JSR     PC,MUL          ;MULTIPLY VALUES
EXIT:   RTS     PC              ;RETURN
```

Fig. 4-10 Recursive coding of factorial function.

The program of Fig. 4-10 calls itself recursively by executing the JSR PC,FACT instruction. Each time it does so, it places both the current value of N and the return address (label RET) in the stack. When $N = 0$, the RTS instruction causes the return address to be popped off the stack. Next an N value is placed in R1, and a nonrecursive call is made to the MUL subroutine.

The subroutine multiply (MUL) uses the value of R1 to perform a multiplication of R1 by the value of an internal number (initially 1), held in MUL, which represents the partial product. This partial product is also left in R1.

Upon returning from the multiply subroutine, the program next encounters the RTS instruction again. Either the stack contains the return address of the calling program for FACT, or else another address-data pair of words generated by a recursive call on FACT. In the latter case, R1 is again loaded with an N value that is to be multiplied by the partial product being held locally in the MUL subroutine, and the above process is again repeated. Otherwise, the return to the calling program is performed, with $N!$ held in R1.

4.3.7. Coroutines

In some situations it happens that several program segments or routines are highly interactive. Control is passed back and forth between the routines, and each goes through a period of suspension before being resumed. Because the routines maintain a symmetric relationship to each other, they are called *coroutines*.

Basically, the coroutine idea is an extension of the subroutine concept. The difference between them is that a subroutine is subordinate to a larger calling program while the coroutine is not. Consequently, passing control is different for the two concepts.

When the calling program makes a call to a subroutine, it suspends itself and transfers control to the subroutine. The subroutine is entered at its beginning, performs its function, and terminates by passing control back to the calling program, which is thereupon resumed.

In passing control from one coroutine to another, execution begins in the newly activated routine where it last left off—not at the entrance to the routine. The flow of control passes back and forth between coroutines, and each time a coroutine gains control, its computational progress is advanced until it passes control on to another coroutine.

The PDP-11, with its hardware stack feature, can be easily programmed to implement a coroutine relationship between two interacting routines. Using a special case of the JSR instruction [i.e., JSR PC,@(R6)+], which exchanges the top element of the register 6 processor stack and the contents of the program counter (PC), the two routines may be permitted to swap program control and resume operation where they stopped, when recalled. This control swapping is illustrated in Fig. 4-11.

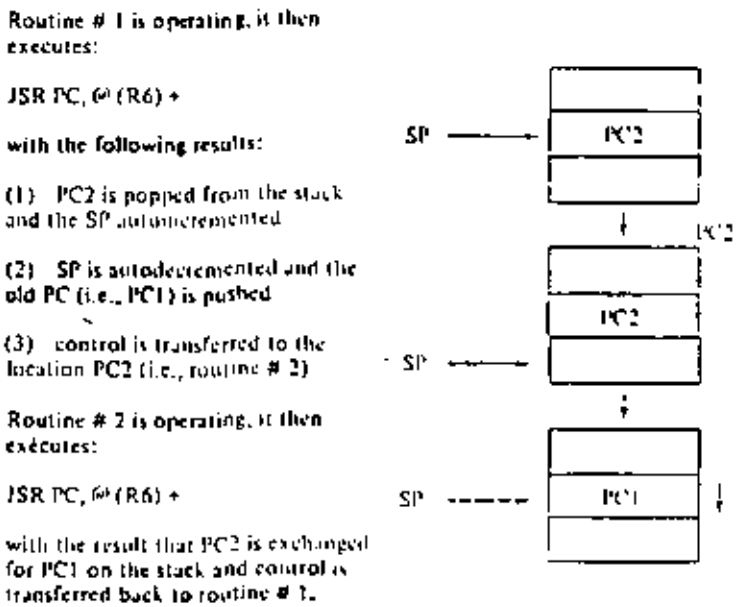


Fig. 4-11 Coroutine interaction.

The power of a coroutine structure is to be found in modern operating systems, a topic beyond the scope of this book. However, in Chapter 6 it is possible to demonstrate the use of coroutines for the double buffering of I/O while overlapping computation. The example presented in that chapter is elegant in its seeming simplicity, and yet it represents one of the most basic I/O operations to be performed in most operating systems.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCCION A LAS MINICOMPUTADORAS PDP-11

MANEJO DE ENTRADA/SALIDA

DR. ADOLFO GUZMAN ARENAS

JUNIO, 1980

Para efectuar una función de entrada salida, el programador debe especificar donde se encuentran los datos, de donde vienen o van y como el dispositivo de entrada salida debe ser manejado. A esto se le denomina programación de entrada salida.

Dependiendo de la función de entrada salida se puede requerir que el procesador espere hasta que la función de I/O sea completada o por otro lado el procesador puede continuar ejecutando tareas simultáneamente con la ejecución de la función de I/O.

El poder programar una computadora para realizar cálculos es de poca aplicación si no hubiera manera de obtener resultados de la máquina. De la misma manera se hace necesario proveer a la computadora con información a ser procesada. Por lo tanto, el programador deberá contar con medios para transferir información entre la computadora y los dispositivos periféricos que permiten cargar datos de entrada y obtener los de salida.

Para la familia PDP 11, la programación de los periféricos es extremadamente simple, ya que una instrucción especial para la entrada salida es innecesaria. La arquitectura de la máquina permite direccionar los registros de estado y datos de los periférico

ricos de manera directa como localidades de memoria. Por lo tanto, las operaciones en dichos registros como es la transferencia de información a o de ellos así como la manipulación de datos dentro de ellos es llevada a cabo con instrucciones normales de referencia a memoria.

El uso de todas las instrucciones de referencia a memoria en los registros de los periféricos incrementa gradualmente la flexibilidad de la programación de entrada salida. Todos los registros de periféricos pueden ser tratados como acumuladores.

Actualmente en la PDP-11, las direcciones correspondientes a las 4 k palabras superiores, están reservadas para los registros internos del procesador y para registros externos de entrada salida, por lo tanto, en caso de tratarse de una máquina chica, la memoria se verá limitada a 28 k palabras de memoria física y 4 k de localidades reservadas para los registros del procesador y dispositivos de entrada salida. En caso de contar con "Memory Management" lo que provee bits extra de direccionamiento 2 en el caso de la PDP 11/40 tendremos una capacidad total de 124 k palabras de memoria física aparte de los 4 k del área de registros antes mencionada.

Todos los dispositivos periféricos son especificados por un juego de registros que son direccionados como memoria y

manipulados con la flexibilidad de un acumulador. Para cada dispositivo hay 2 tipos de registros asociados:

1. Registros de control y estado
2. Registros de Datos

Cada periférico puede constar de uno o más registros de control y estado (CSR) que contienen toda la información necesaria para comunicarse con dicho dispositivo.

El unibus es una vía común que interconecta el procesador, memoria y periféricos. Debido a la arquitectura de la máquina sólo puede haber un dispositivo controlando el unibus en cualquier tiempo. A este dispositivo se le denomina Master. Los dispositivos pueden solicitar ser Masters, ya sea haciendo una solicitud de Bus o una solicitud de no procesador a la lógica de arbitraje de prioridades del procesador.

La solicitud es atendida si es la de mayor prioridad. El nuevo master asume el control del bus cuando el actual master libera el control del bus. El nuevo maestro puede solicitar que el procesador atienda el periférico o puede iniciar una transferencia de datos sin intervención del procesador.

Las interfases en la PDP-11 pueden clasificarse en 3 tipos:

4

1. Slave (esclava) - Esta interfase no está prevista para ser Master. Ella sólamente puede transferir datos a o desde el unibus por comando de un dispositivo Maestro.

2. Interrupt (interruptor) - Esta interfase tiene la habilidad de ganar el control del bus en el orden de dar al procesador la dirección de la subrutina, lo cual es usada para atender la solicitud del periférico.

DMA. Esta interfase tiene la habilidad de ganar el control del bus de manera de transferir información entre ella y algún otro periférico.

Un sola interfase puede emplear los 3 tipos anteriores.

DL 11

La interfase para línea asíncrona DL 11 es una interfase para comunicaciones designada para convertir datos de serie a paralelo. La interfase cuenta con 2 unidades independientes, (receptor y transmisor), capaces de establecer comunicación simultánea en ambos sentidos.

La interfase DL11 lleva a cabo básicamente 2 operaciones: recepción y transmición de datos asíncronos. Cuando recibe datos, la interfase convierte un caracter serie asíncrono proveniente de un dispositivo externo en un caracter en paralelo requerido para una transferencia al unibus. Este caracter puede ser mandado por el bus a la memoria, o un registro en el procesador a algún otro dispositivo. Cuando se transmiten datos en paralelo desde el bus son convertidos a serie para su transmisión a un dispositivo externo. Debido a que las 2 unidades son independientes, es posible establecer comunicación de manera simultánea en ambos sentidos. El receptor y el transmisor operan por medio de 2 registros: el registro de control y estado, para comando y monitoreo de funciones y el buffer de datos para guardar los datos antes de transferirlos al bus o a un dispositivo externo.

6

Descripción DL11 Teletype Control

Transmisión

Cuando el CPUbus direcciona el Unibus, la interfase DL 11 decodifica la dirección para determinar si el teletipo es el dispositivo externo seleccionado y si es el seleccionado qué función debe desempeñar, entrada o salida. Si por ejemplo el teletipo ha sido seleccionado para aceptar información a imprimir, datos en paralelo provenientes del unibus son cargados en el buffer de transmición del D 11. En este punto la bandera de XMIT RDY baja debido a que la lógica del transmisor ha sido activado (la bandera vuelve a estar baja una fracción de bit después si el transmisor no se encuentra activo en ese momento). La interfase genera el bit de arranque y transmite bit por bit en serie al teletipo, de nuevo pone la bandera XMIT RDY (tan pronto como el registro de buffer se encuentra vacío aún cuando el registro de corrimiento se encuentre activo. Después transmite el número requerido de bits de STOP.

Recepción

La sección de receptar la longitud del caracter es seleccionable por medio de un selector. El caracter recibido aparece justificado a la derecho en el registro buffer recepción eliminando los bits de arranque y paro.

El caracter completo es formado en el UART y es transferido al registro buffer de recepción (RBUF) en el momento en que el centro del primer bit es muestreado. En ese momento el bit de recepción efectúa el registro de entrada y control es prendido si el bit de Interrupt Enable se encontraba prendido se genera una señal de solicitud de interrupción. Los bits no usados son llenados con ceros y los bits 12-15 contienen información acerca del caracter integrado por el UART. Notece que el programa tiene un caracter completo de tiempo para retirar el caracter completo del buffer de datos antes de que el nuevo caracter sea colocado en el registro de recepción por el UART. En el caso de que el programa falle en leer este caracter anterior, se pierde y el bit de exceso y error son prendidos (bit 14-15) en el registro buffer de recepción. En el caso de que no se presente normalmente el bit de paro el UART presenta lo que supuestamente recibió, más el bit error 13y15 prendidos.

Programación

La interfase entre el programa corriendo en el procesador PDP-11 y el DL-11 se lleva a cabo mediante 4 registros. Estos son registros de estado de recepción (RCSR); 2) registro buffer de recepción (RBUF); 3) registro buffer de estado de transmición (XCSR); y 4) Registro buffer de transmisión (XBUF). La función de cada uno de estos bits se da a continuación.

CR - 11

La lectora de tarjetas CR-11, lee tarjetas perforadas de 80 columnas. La lectora está diseñada para leer secuencialmente, los datos en 80 columnas empezando con la columna 1. Cada columna tiene 12 zonas o renglones, una perforación es interpretada como un uno binario y la ausencia de perforación como un cero. Los datos son leídos de la tarjeta una columna a la vez. Los datos son presentados en dos formatos para entrada a la computadora.

Modo Comprimido.- Las 12 zonas de la tarjeta son codificadas en un byte (8bits), permitiendo un almacenamiento más eficiente de la información.

Modo no comprimido.- Un bit es empleado para presentar el estado de cada zona en la tarjeta.

La Lectora CR 11 consta de 3 registros para comunicarse con la computadora. Estos son registro de estado y dos registros de datos. Uno de los cuales presenta los datos no comprimidos y la otra comprimidos. La selección de formatos se lleva a cabo seleccionando el registro apropiado. Los datos en ambas formas se encuentran siempre presentes. A continuación se presenta la estructura de dichos registros.

RJPØ4

El RJPØ4 es un subsistema de disco de cabeza móvil el cual consiste en un controlador RH 11 y de uno a ocho drivers de disco RPØ4.

El Unibus provee la interfase entre el procesador la memoria, y el controlador RH 11. Todas las transferencias efectuadas entre la memoria y el RH 11 por medio de la facilidad de - DMA del Unibus.

El RH 11 contiene dos puertos en el Unibus: uno designado como un puerto de control y el segundo como un puerto de datos.

Los datos pueden ser transferidos a través de ambos registros. Para operación normal con memoria conectada a Unibus A como se muestra en la figura 1 sólamente es usado el puerto de - control, el puerto de datos no se usa.

El RH 11 se encuentra dividido en dos grupos funcionales, línea de registro y control y línea de DMA.

La línea de registro y control permite al programa leer y/o escribir en cualquier registro contenido en el RH 11. Hay

un total de 4 registros en el RH 11, 15 registros en cada drive y 1 registro compartido que es parcialmente compartido en el RH 11 y en el Drive seleccionado.

La línea de DMA funcionalmente consiste en una memoria FIFO de 66 palabras por 18 bits y su lógica de control.

La función primordial de esta memoria, que de aquí en adelante llamaremos SILO es el de buffer de datos para compensar fluctuaciones de retardo en el Unibus al solicitar el DMA.

Cuando una instrucción en la PDP 11 direcciona el - RH 11 para leer o escribir cualquier registro en el RH 11 o en algún Drive, se inicia un ciclo de Unibus y los datos son dirigidos al o de el RH 11. Si el registro a ser direccionado es local (se encuentra en el RH 11), la lógica de control de registros permite el acceso al registro apropiado. Si el registro direccionado es remoto (contenido en uno de los drives, la lógica de control de los registros inicia un ciclo de control de Massbus. El acceso a los registros en el - Drive por medio de la lógica de control del bus no interfiere con la transferencia DMA la que puede llevarse a cabo simultáneamente. Los registros locales del RH 11 especifican parámetros tales como dirección del Bus y contador de palabras, mientras que los registros del Drive especifican parámetros como dirección deseada en el dico, información de estado, etc.

La línea de datos de DMA funcionalmente consiste en el Bus de datos Massbus, la memoria SILO y la lógica de NPR del - Unibus.

La figura 2 presenta un diagrama de bloques simplificado de la línea de DMA con un sólo Unibus.

Los 3 comando de transferencia de datos que pueden ser llevados a cabo por el RH 11 son escritura, lectura y checado de escritura.

Antes que cualquiera de estas operaciones ocurra, el programa especifica una dirección en memoria (MA), una dirección de cilindro (CA), una dirección deseada de sector y pista (DA) y el número de palabras. La dirección de Memoria representa la localidad de memoria donde se iniciara la lectura o escritura. La dirección de cilindro deseada es la posición en la que la cabeza deberá - posicionarse.

El sector y pista deseado representa la dirección de inicio en la superficie del disco donde los datos serán escritos o - leídos.

El número de palabras a ser transferidas a o del disco.

La línea de datos de DMA funcionalmente consiste en el Bus de datos Massbus, la memoria SILO y la lógica de NPR del Unibus.

La figura 2 presenta un diagrama de bloques simplificado de la línea de DMA con un sólo Unibus.

Los 3 comando de transferencia de datos que pueden ser llevados a cabo por el RH 11 son escritura, lectura y checado de escritura.

Antes que cualquiera de estas operaciones ocurra, el programa especifica una dirección en memoria (MA), una dirección de cilindro (CA), una dirección deseada de sector y pista (DA) y el número de palabras. La dirección de Memoria representa la localidad de memoria donde se iniciara la lectura o escritura. La dirección de cilindro deseada es la posición en la que la cabeza deberá posicionarse.

El sector y pista deseado representa la dirección de inicio en la superficie del disco donde los datos serán escritos o leídos.

El número de palabras a ser transferidas a o del disco.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

A P L I C A C I O N E S

DR. VICTOR GEREZ GREISER

JUNIO, 1980

The Mini Computer as a Control Element

Dudley B. Hartung
Management Methods, Inc., Waltham, Massachusetts

Mini computers have been used to control a wide variety of processes and functions including machine tools, chemical processes, steel mills, and warehousing systems. Articles in technical journals and talks at seminars have described in some detail many of these individual applications. But what is a good application -- when do you use a hard wired control system and when should you consider a mini computer?

The decision must be based on costs dollars per function -- and reliability and maintainability. In general, reliability can be disposed of as being indirectly related to dollars. The simplest or mass-produced system is normally the cheapest system and also the most reliable system. Maintainability can be given a cost value. The decision, therefore, can be directly related to costs. Costs naturally refer to the initial capital cost of the equipment and also to recurring cost of operation, including the aforementioned maintainability cost, operator cost, quality of product value, etc. Some of these costs can only be roughly approximated and may be intuitive guesses. The cost of the equipment, however, should be fairly easy to derive by knowledgeable people during initial planning stages.

A control system -- any control system -- consists of inputs, outputs, and decision makers. In comparing hard wired systems with computer systems, the input and output devices probably stay comparable in cost. Input and output devices consist of operator switches, sensors, solenoids, servo or discrete (on-off) motor controls, etc. The decision maker is the logical system which determines the effect of input changes upon output actions. With a hard wired system, each subdecision or each function of the control system has its own logic. A mini computer time shares its logic to accomplish many functions with a relatively small logical device. The mini computer, therefore, becomes essentially the complete decision maker, even when there are hundreds of inputs and outputs with varying degrees of interrelationship. This is where the cost savings of a computer system come.

Many more decisions can be made per dollar with a computer compared to hard-wired logic.

The are some inherently costly aspects of a computer system. Holding functions must be stored externally. All inputs and outputs of data are in high speed serial word which means that switch inputs, for instance, must be held on until polled by the computer. The holding device might be the operator's finger. The outputs must have holding relays or their solid state equivalent. Inputs and outputs to the computer are always the same binary words at low levels, requiring filtering and level conversion at the interface. No power is available for force-type functions, requiring amplifiers and power relays. Of course, some of these restrictions apply to many hard-wired controls. But if the decisions are very simple, the input and output buffering, filtering, and holding may be more expensive than the complete hard-wired control.

The computer itself is limited by speed and by the size of internal memory utilized for storing data and computer program. It may be cheaper even in a computer controlled system to do some complex but frequently used and repetitive functions externally. For example, a servo loop could be performed in a computer but in most cases is done externally. Interpolation for a machine tool, which is the precise control of velocities in two or more axis to draw a straight cut or a circular cut is expensive in computer time in that it takes a large portion of a computer. A single computer can do all interpolation and control one or two high speed, high accuracy machine tools. If the interpolation is done externally, 5-20 machine tools can be similarly controlled.

So how is a decision made to go hard-wired or mini computer? The system costs must be estimated in both ways. This requires some understanding of the end of process and the requirements of both a hard-wired system and the capabilities of computers. In many cases, the computer can supply additional functions at very low cost which have to have some value placed on them to honestly compare systems. In other cases, the function to be performed is so complex that it is immediately obvious the computer is the solution. Labor costs of both of the design and building of a system and the operation must be considered.

Costs also include effects of lead time variation, set-up speed, and rejects. All of these costs vary and relate to a particular application.

Rules of thumb are dangerous and can be misleading, but there are some systems where computer control should be looked at very carefully. If many simple decisions -- or many monitoring points, such as those on a transfer line, are required or if very complex relay trees or logical decisions must be made, a computer should be considered. Complicated decisions requiring mathematical functions,

particularly if changing either between runs or over a period of time or the requirement for a great deal of stored data for look-up tables or individual parts programs suggest computer control. Finally very specialized problems or machines or processes with only one system or a few systems being built, particularly where modifications between initial concept and final operating equipment are foreseen due to technical unknowns, are particularly good applications for computer control. This is true not only because of the possible savings in hard ware costs, but more importantly, because of the normally much lower design cost.

The mini computer can be a panacea for many ills, and should be looked at by the builders and users of any controlled system. It will be found that not all systems justify on an economic basis the utilization of computers, but conversely, it will be found that what seems like an expensive and sophisticated control system can often be easily justified purely on an economic basis.

COMPUTER CONTROL OF VACUUM DEPOSITION PROCESSES

R. M. Centner and R. A. Wilson
Bendix Research Laboratories
Southfield, Michigan

ABSTRACT

With the advent of the low cost minicomputer, full automatic control of vacuum deposition processes appears both technically feasible and economically attractive. To date, vacuum deposition processes have been largely controlled manually, although simple controllers have been available for controlling portions of the process such as the vacuum pumpdown and the deposition rate during evaporation. Automatic control promises to improve process efficiency and performance, and to improve the uniformity of the resultant products, while freeing personnel from routine operating tasks. The approach to computer control of vacuum deposition processes (evaporation and sputtering) is discussed, and the conceptual design of an automatic process controller based on a minicomputer is presented. The advantages of automating these processes are reviewed.

INTRODUCTION

This paper discusses the application of a small digital computer, or minicomputer, to automatic control of vacuum deposition processes. Included are the establishment and control of the vacuum environment, control of the evaporation process, control of the sputtering process, and control of a number of lesser functions related to these processes. Emphasis is placed on demonstrating the feasibility of applying a dedicated computer to the control of a single vacuum deposition system, although of course other computer/deposition-system relationships may be preferable under certain circumstances.

In the following sections the control requirements for the vacuum deposition processes are reviewed, together with the present methods of control and some of their disadvantages. The approach to computer automation of these processes is then described and the conceptual design of an automatic controller is presented. Finally, it is shown that computer automation leads to improved system efficiency and performance, improved product uniformity, and the freeing of personnel from routine operating tasks. All of these are ultimately reflected as economic advantages.

The following discussion of control requirements and controller design concepts is specifically oriented toward the batch-type vacuum deposition system. Obviously the same general approach can be also applied to the automation of an in-line type system, although the specific control functions will differ somewhat.

VACUUM DEPOSITION PROCESS CONTROL

Control Requirements

The basic vacuum deposition processes covered in this paper are thermal evaporation and sputtering. These two basic processes encompass quite a number of different operations, including:

(1) Vacuum cycle control
(2) Pressure control
(3) Substrate conditioning
(4) Evaporation source control
(5) Sputtering control
(6) Glow discharge cleaning
(7) Substrate rotation
(8) Bell jar and base plate cooling

Each of these functions is a somewhat independent operation, although they must be appropriately grouped and coordinated to yield the desired process sequence. Each of these operations requires control functions. In some, the control is based on the behavior of a sensed parameter relative to a desired or setpoint value. Pressure control and base plate and bell jar cooling are examples of this type of control. In other cases, control is based on a timed sequence, as is generally the case for sputtering and glow discharge cleaning. Evaporation source control is an example of an operation where both bases of control are used: the soak power level is normally maintained for a timed period, whereas during actual deposition source power is usually controlled to yield a specific deposition rate until a specified film thickness is achieved. During the process control sequence, most of the items listed require only simple on-off type control of solenoid valves, power supplies in which the voltage or current levels have been preset, and motors. "Pressure control" involves adjustment of a variable valve, while substrate conditioning and evaporation source control may involve the control of variable power supplies. Thus, a vacuum deposition process may include a number of steps or operations, but each operation by itself constitutes a relatively simple control requirement which can readily be automated.

Present Control Methods

To date, vacuum deposition processes have been largely controlled manually, although simple controllers are presently available for controlling portions of the process. The latter are

Reprinted with permission from *1970 WESCON Technical Papers*, vol. 14, paper 4/4.

hardwired, modular devices or units, each controlling a single operation, and are generally limited to two areas: vacuum cycle control and evaporation source control. Evaporation source control is generally accomplished through the combined efforts of two modules or units. One is a monitor unit which determines film thickness and deposition rate, and provides a signal or contact closure when thickness reaches the set point value. The second unit provides a signal for controlling source power during the soak and deposition portion of the cycle, using signals from the monitor unit as the basis for control during the deposition portion.

These methods of control have a number of distinct disadvantages as follows: Frequent attention by an operator is required during the course of the process cycle or run. Even when the previously cited control modules are used, their operation is normally uncoordinated. When the vacuum cycle controller has established the proper environment, the operator is required to initiate the source control cycle or the actual "process". When the latter is completed, the operator must again manually initiate the return of the chamber to atmospheric conditions. Other auxiliary operations, such as glow discharge cleaning, must also be manually introduced in the cycle as required. Thus personnel who might be performing other tasks are tied up in routine equipment operation.

The high degree of operator involvement can also influence the process in at least two other ways. First, since the steps of the process must each be initiated by the operator, unnecessary delays may be incurred between the completion of one operation and the start of the next, thereby reducing the efficiency of the process and increasing the overall run time. Second, since manual control of the process involves a certain degree of operator judgment in some of the steps, the possibility exists for variations in product quality or uniformity from batch-to-batch. All of these disadvantages are ultimately reflected in cost factors which would be improved by automatic control of the process.

Approach to Computer Automation

Automatic computer control of vacuum deposition processes has been technically feasible for some time. The size and cost of the computers which have been available, however, have generally made such automation impractical and economically unsound. Exceptions to this are cases where the computer can be used to control a number of vacuum deposition systems, or where the computer can be used to control a deposition process in addition to performing other duties. The recent advent of small, low-cost minicomputers has changed the picture dramatically. Now an automatic vacuum deposition process controller based upon the use of a small dedicated computer and designed to serve a single system appears to be both technically feasible and economically attractive. It is to this approach that we now direct further attention. The next section describes an automatic controller based on this approach. The advantages of such a controller are outlined in a subsequent section.

An Automatic Vacuum Deposition Process Controller

It was noted earlier in this paper that a vacuum deposition process is made up of a number of different operations, each of which constitutes a straightforward control problem. An automatic controller based on the use of a digital computer can serve to organize, coordinate, and control the execution of these operations.

Controller Functions: The automatic vacuum deposition process controller could be capable of performing the following functions:

(1) Automatic vacuum cycle: control of the bell jar; the vent, roughing, foreline, and hi-vac valves; and the ion tube filament. Protection of the diffusion pump from overheating and/or excessive fore pressure.

(2) Automatic pressure control: control damper valve to keep chamber pressure constant at a preset value for part or all of the operating cycle.

(3) Substrate conditioning control: control heating (to bake or conditioning temperature), annealing, and cooling of the substrate.

(4) Thickness-rate functions: using thickness input signal, calculate deposition rate and determine when thickness reaches set point values. These data would be used by the evaporation source control function.

(5) Evaporation source control: control the cycle of one or two sources (power rise, soak, deposition rate, and shutter).

(6) Sputtering control: turn on preset filament, anode, and target power supplies at programmed point in cycle and maintain for timed sequence. Monitor target current while sputtering. Interrupt timed sequence and sound alarm if current drops below a preset value.

(7) Glow discharge cleaning: turn a fixed power supply on for a preset time interval at any of several pre-programmed points in the cycle.

(8) Substrate rotation: turn substrate rotation motor on and off at predetermined points in the cycle. Fixed speed (manually variable via control not provided).

(9) Bell jar and base plate cooling: turn coolant system on and off. Turn on whenever sensed temperature exceeds a set point value.

Controller Description: The automatic vacuum deposition controller would be based on a small digital minicomputer with a read-only memory. Figure 1 is a block diagram showing the relationship of the controller to the vacuum deposition system, while Figure 2 is a simplified block diagram of the automatic controller itself.

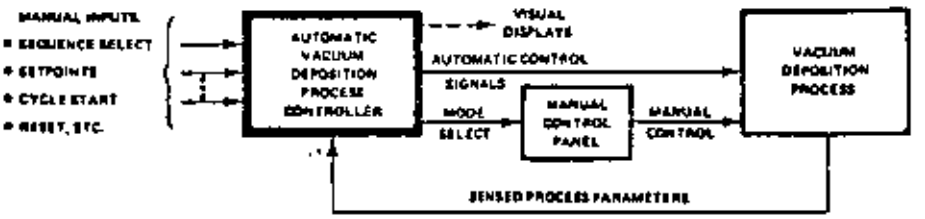


Figure 1 - System Block Diagram

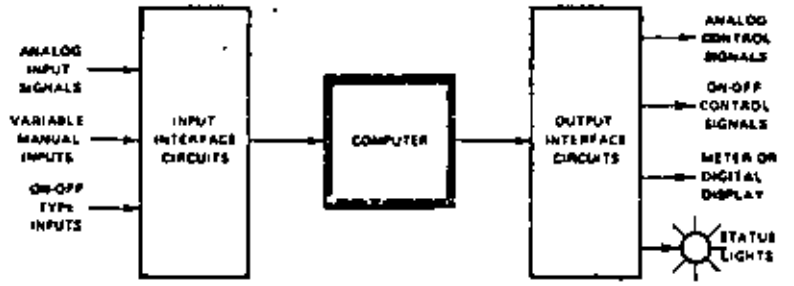


Figure 2 - Simplified Automatic Controller Block Diagram

The automatic controller would be fitted with a front panel typically containing the following: An analog meter and a digital (Nixie tube) readout, each with a function selector switch; a four-digit thumbwheel switch with function selector switch; several toggle and push-button switches; and a number of status or indicator lights.

The unit would have connections at the rear for all input signals and for analog and on-off type output (system control) signals. All variable input signals from external sources are assumed to be analog dc voltages. These would include pressure, temperature, and thickness signals. Normalizing amplifiers would be provided to adjust the relative voltage levels of those signals. The normalized signals are fed to the computer by means of a multiplexer and an analog-to-digital converter.

Variable parameters to be displayed could be read out either on the meter or on the digital display. Variable inputs which are introduced manually (set points, soak power, rise times, etc.) could be introduced via the thumbwheel switch. (Alternative methods of introducing these inputs might include: (1) potentiometers, whose output signals would be sent to the computer via the multiplexer and analog-to-digital converter and (2) a punched card and card-reader arrangement.)

On-off type manual inputs, such as "cycle start", "automatic recycle", and "reset" would be introduced to the computer by means of a status register. On-off signals from the process, such as from bell-jar hoist limit switches will be handled in the same way. Two types of control outputs are provided: Digital-to-analog converters provide analog voltages for functions where variable control signals are required. On-off type control signals or contact closures are provided for the operation of solenoid valves and solenoid-operated shutters, turning preset power supplies on and off, operating bell jar hoist and substrate rotation motors, and in fact most of the system control functions.

The input and output interface circuits would be mounted on plug-in cards and housed in unused space in the computer cabinet. The entire automatic controller could be packaged in a small bench-top cabinet, or as a small rack-mounted unit, occupying less than 24 inches of panel height.

Once the various manual inputs are set, normal operation of the system consists simply of pressing the "cycle start" button. No further attention is required until the automatic cycle has been completed and the bell jar has

been raised. Provisions for reset and other controls would be provided, however, for use when manual intervention is felt necessary.

Advantages of Computer Automation

Computer automation of the vacuum deposition processes has significant advantages with respect to either manual control or the use of separate modular units to automate the control of individual operations.

Figure 3 illustrates the cost advantage of computer automation of the vacuum deposition process, as compared with the use of a number of individual hardwired control modules to accomplish the same objective. The diagram shows relative controller cost versus the relative degree of automation. The cost versus features automated for the modular approach will rise at a fairly uniform rate. The cost of computer automation of only a single operation would be rather high, since it would include the cost of the computer itself. Automation of additional features costs relatively little, however, since this mainly involves a revision to the computer program and the addition of appropriate interface circuits. The crossover point at which the cost of computer automation drops below that of the modular controller approach occurs when only a relatively few operations are to be automated. Modular controllers are not known to be available at present for some of the features included within the scope of the automatic vacuum deposition process controller described herein.

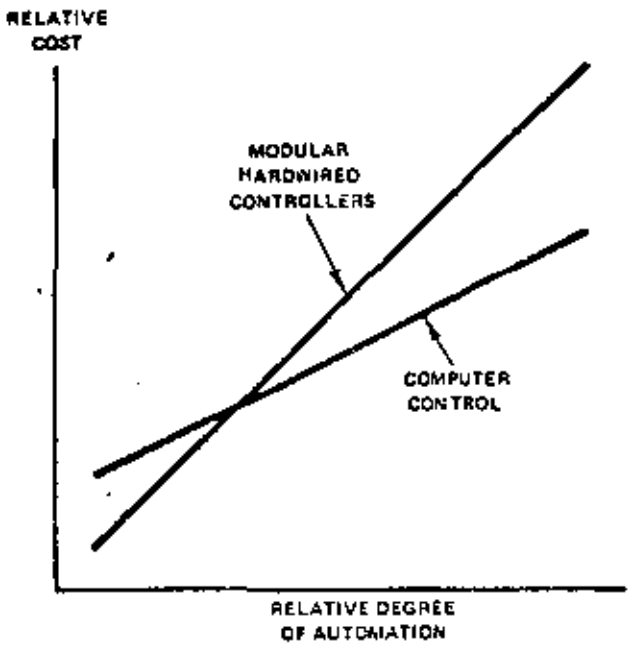


Figure 3 - Cost Versus Degree of Automation

The computer automated controller also results in a number of performance or operational advantages:

(1) Flexibility: For the user having varying process requirements it offers flexibility. The operations of the sequence can be quickly added, deleted, and otherwise altered, and set-points can be established by means of switches and other controls on the panel of the automatic controller.

(2) Process Repeatability: For the user making the same product repetitively, it offers a high degree of process repeatability, once a given sequence has been established and set-point values have been set, resulting in uniformity of the resultant product.

(3) Process Efficiency: The automatic controller will provide smooth and rapid transition from one operation or step of the process to the next, completing the cycle or run in a minimum of time and thus enhancing the efficiency of the process.

(4) Personnel Advantages: Once the sequence and set-point values have been established, the operator is only required to press the "start" button and the complete cycle will be executed unattended. Thus personnel are freed from routine operating tasks.

All of these are ultimately reflected in economic advantages of automatic computer control of vacuum deposition process.

CONCLUSIONS

Although vacuum deposition processes require a relatively large number of control functions, each function is reasonably simple and lends itself quite readily to automatic control techniques. The advent of the low-cost minicomputer now appears to make computer automation of vacuum deposition processes both technically feasible and economically attractive. Automated control offers a number of operational advantages over presently used semiautomatic control methods, many of which are ultimately reflected as additional economic advantages. Hence it may be expected that computer automation of the control of vacuum deposition processes will achieve growing importance in the near future.

Batch Control with a Minicomputer

R. YOUNG, Emery Industries, Inc. and
D. E. SVOBODA, Jackson Associates

At Emery Industries, a minicomputer controls batch production of chemicals, consisting of esterification reactions of fatty acids with alcohols. Functions of the mini-system range from simple alarm-point monitoring to ddc. The authors describe the hardware and software for a system that demonstrates the minicomputer's value as an economical, flexible, sophisticated production tool.

FOR PROCESS APPLICATIONS, it's often more economical to design the control system around a digital computer rather than hardware logic components and analog setpoint controllers. Prices of minicomputers start at $3,000 to $4,000 without core memory; therefore, for all but the simplest systems, the cost of the computer will be less than the cost of the hardware it replaces.

In addition, the overall effort required to design computer software (even with assembly-language programming) is less than that required for equivalent hardware, and the computer programs are easier to modify. Sophisticated control algorithms that can reduce operating costs—but which are difficult to implement with hardware—can usually be programmed for a computer with little difficulty.

The process control system described in this article performs a variety of functions typical of computer-based systems. These functions include "contact-closure" input and output, analog input and output, direct digital control (ddc) of analog process variables, timing and sequencing of process events, and logging of process variables and events. The computer hardware is discussed first, followed by an explanation of programming techniques.

Hardware for the mini

The computer control system (shown in the figure) is built around Digital Equipment's PDP-5L computer and Peripheral Equipment's 7520-9 magnetic tape unit. Additional equipment consists of analog-to-digital (A/D) and digital-to-analog (D/A) converters, contact closure inputs and outputs, a time-of-day clock for event logging, a 50-Hz interval timer which provides the time base for the entire system, a stall alarm, and a teletypewriter for logging. Operator messages are presented through an annunciator panel.

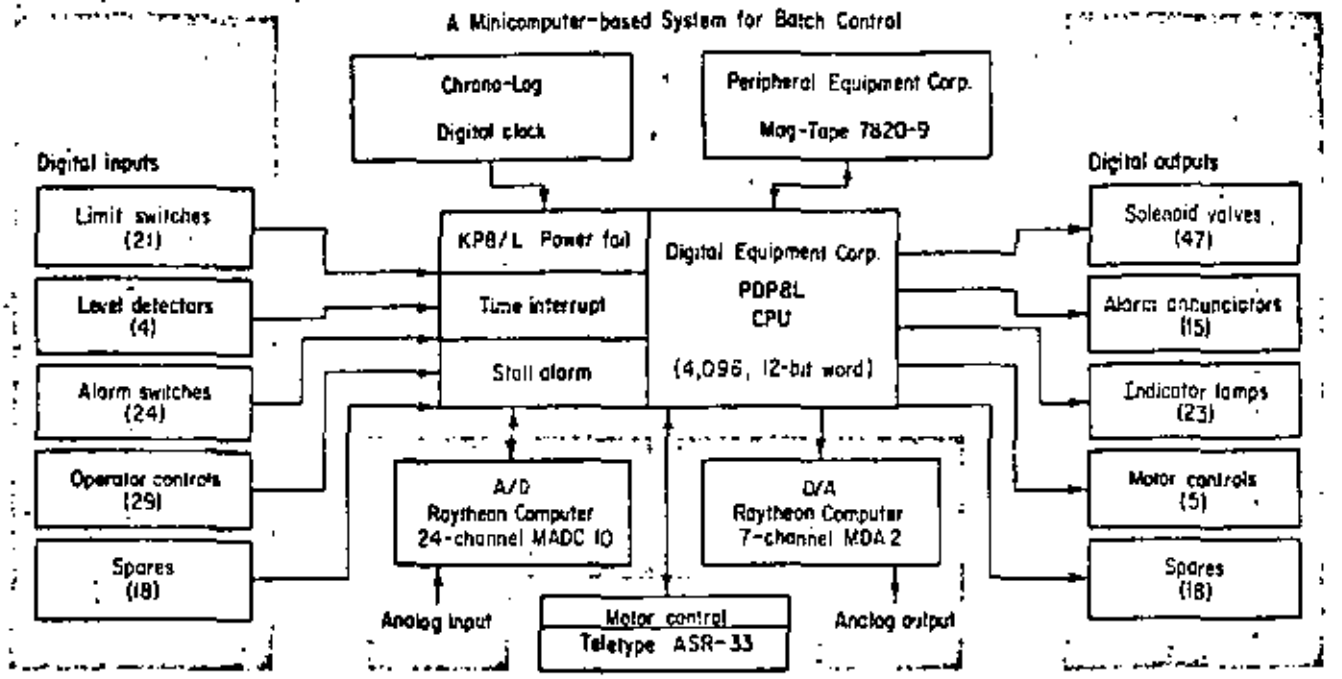
The computer memory holds the control program and the parameters associated with each product that will be manufactured. The magnetic-core memory consists of 4096 12-bit words, and has a 1.6-microsecond cycle time.

Computer I/O facilities consist of 12-input and 12-output data lines. Data can be selected and placed on the output lines or accepted from the input lines at appropriate times, under control of the program. In addition, six address lines are used by the external logic to route input or output data to or from external equipment such as contact-closure sensors and D/A converters. Control lines that can be pulsed or tested by the program synchronize the external logic with the control program.

Contact-closure inputs are arranged and addressed in groups of 12 that correspond to the 12-output data lines of the computer. DC input circuits consist of RC filters (to take out contact-bounce noise) followed by Schmitt triggers (to convert inputs to logic levels). AC inputs pass through isolation transformers and diodes for conversion to dc. Contact-closure inputs include signals from the annunciator panel, operator pushbuttons, valve-position limit switches, and level detectors.

Contact-closure output hardware consists mostly of solid-state devices, triacs for ac output and transistor switches for dc output. A few relays, driven

Reprinted with permission from *Instrum. Technol.*, vol. 17, pp. 72–76, Aug. 1970.



by transistors, are used where continuity is desirable in case of a logic power supply failure. The contact-closure outputs are also arranged in groups of 12.

Each contact-closure output has a storage flip-flop which receives and holds the data from one of the output lines. Groups of contact-closure outputs are selected to receive data by codes provided on the six address lines. The outputs include signals to the annunciator panel, motor start/stop signals, and signals to solenoid-operated pilot valves that supply air to the process valves.

Analog voltage inputs are multiplexed to an A/D converter, changed to binary numbers, and put into the input data lines of the computer. Process variables such as temperature, pressure, and flow are entered into the computer via the A/D converter. Manual setpoints (operator-adjusted potentiometers) also pass through the A/D converter.

The D/A converters accept binary output data and produce coresponding analog voltages. A separate converter is used for each analog output, with a range of ±10 volts; each D/A converter has a separate six-bit address. Analog output voltages go to panel meters which display process variables, and to electric-to-pneumatic (E/P) converters which provide air to throttling control valves.

The interval timer generates a time-interrupt signal for the computer every $\frac{1}{10}$ of a second, the basic timing for sequencing of the process. The timer also provides a reference for integral and derivative control in the ddc loops.

The stall alarm consist of two 10-millisecond timers that can be reset by the program. The timers must be reset so that at least one is always running; otherwise, an alarm signal is produced. If a program error or hardware malfunction alters the normal sequence of the program, the timers will not be reset often enough, thereby actuating a stall alarm.

The teletypewriter and clock are used in a conventional manner for event logging. Time in hours, minutes, and seconds can be read from the clock and printed by the teletypewriter with a typical message: 05:16:57 THERMINOL FROM ESTERIFIER LOW FLOW.

Software for the mini

The computer programming, or software, regulates the operation of a computer-controlled process, and constitutes a major part of the design and development effort of such a system. Some of the general tasks for Emery Industries' computer can be mentioned; they are typical for a control computer that is applied to a batch process.

Depending on the product's requirements, the software sequences the valves, provides timing, and monitors the status functions that determine when steps should be taken. The computer checks six variable and 14 logical (yes or no) "endpoints," any combination of which can control the duration of a step or the branching to one of several possible next steps. Computer software must also check a total of 50 temperatures, manual valve positions,

and other status signals that show the system is operating normally. Critical status errors can stop the chemical process.

The annunciation function of the software puts out printed messages (on the teletypewriter) concerning the status indicators and program flow. A self-checking function detects and annunciates computer malfunctions.

It's impressive that the minicomputer has sufficient capacity for all of these functions. The key to fitting them in was the careful organization of the software into subroutines. Besides the normal advantages of easier troubleshooting and simplified program changes, an important feature of subroutines for this application is that a subroutine can be called many times during a program sequence which minimizes the total required number of program statements. For a program of routine operations but unique sequence and duration of the operations, this approach greatly shortens its length.

A general principle of the software organization: every function which is routine is a subprogram, and only those functions unique to the particular chemical product remain in the main program.

One operation illustrates the application of this principal. Changing the combination of the 47 on-off valves of the process system can happen as many as 30 times during the batch process. A change takes two program statements, a valve-change subroutine call followed by an encoded combination. The valve-change subroutine decodes the combination, selects the valves to be opened or closed, and produces a valve-change message which gets printed by the annunciator subprogram. Finally, the proper valves are actuated by the update subprogram which does all I/O functions.

Time-sharing is another familiar tool that has been applied in this system. Simultaneous operation of functions such as output printing, system error detecting, endpoint detection, and ddc was deemed necessary; therefore, a time-interrupt system for time-sharing was devised. The executive control program is divided into 15 equal time slots, each of which contains parts of the programming. Four passes per second through the 15 slots are required for the execution of all statements. At the end of the first ½-second interval within each slot, the contents of the accumulator and the address of the next statement to be executed (in that particular time slot) are saved by the executive before going on to the next slot, and on to the 15th.

At the start of the corresponding time slot during the next ¼ second pass, the accumulator is restored by the executive and the program proceeds as if the interruption had not taken place.

An executive "fork control" subroutine (an unconditional jump) permits programming in one slot to alter the flow of that in another slot. Parameter values in one time slot can be read or modified from another slot.

The ddc loops go into a single time slot. These loops are the digital-computer equivalent of analog control loops that operate valves. Five valves control nine process variables. Each loop has setpoint inputs from the main program and process variable inputs from the update program. The control algorithm resembles that of a normal analog loop, except that summation replaces analog intergration and digital differentiation replaces analog.

A "tune-up" control panel permits rapid optimization of various constants for these loops. Considering the computer speed and the time constants of this application, the ddc control is indistinguishable from analog control but it is much easier to tune and modify.

Because of possible failures, safeguard procedures have been included in the software design. Manual takeover of any valve or any ddc-loop setpoint is possible; these options are designed so that automatic control can be reestablished smoothly.

For safety as well as convenience, all of the above software is stored on magnetic tape which is read into the computer by a simple loader program. Normally, all subprograms remain in the computer core and only the main program is read in at the beginning of each chemical process. Provision for updating the magnetic tape is also part of the software.

In Emery Industries' system, every phase of real-time computer usage is represented—from simple alarm-point monitoring to unattended direct digital control with self-checking features. The system has been designed so that the operator can interact with the control system to alter setpoints if necessary, or adjust the control system to handle process upsets manually if the need arises.

Some of the software concepts borrowed from computer time-sharing technology (which permit many subroutines to be activated simultaneously), contributed to the flexibility of the system. This organization permits a new main program to be written for an entirely new product with a minimum of effort, inasmuch as the main programs consist primarily of a sequence of calls to the various utility subroutines, along with their required endpoints and setpoints.

Dr. Robert Young is Director of Engineering at Emery Industries, Inc., Cincinnati, Ohio; Dr. Dean E. Svoboda is a consultant with Jackson Associates, Columbus, Ohio. Article is based on paper presented at the Conference on Solid-State Devices for Industrial Applications, sponsored by IEEE with ISA as a cooperating society, Cleveland, 1970.

Part 5
Process Control Applications Including Direct Control, Supervisory Control, and Advanced Control

Introductory Comments

Automation in the process industries has been under way for many years. The variety of applications is extensive. Early systems tended to use a rather large process control computer to implement many applications in a single plant. The advent of the minicomputer has provided an alternative, namely, the dedication of a minicomputer to a single task or, at most, a small number of related tasks. This approach to automation has been termed "islands of automation" as opposed to overall or integrated automation of a plant. This leads, of course, to an alternate set of problems, intercomputer communication, since various minicomputer applications will be required to share information concerned with resources, orders, etc.

The papers in this part describe the various control applications that arise in industrial processes. They are selected in order to illustrate the variety of problems, the variety of control theory, and technology that can be applied, and the problems of implementing such systems. The first paper by R. J. Mouly describes in some detail various applications in a typical process plant which provide opportunity for a great variety of different control theories to be applied. The organization of such a complex control system is important, for it may mean the difference between success and failure in any specific instance. Mouly describes the way plants are organized and how controls themselves must be organized in order to provide an effective system.

The second paper by J. M. Lombardo describes control at the lowest level, namely, direct control where the function of the computer is to directly manipulate valves, voltages, etc. in the plant. Of importance here is the integration of the operator into the control system as well as the constraints imposed by reliability. In particular, the design of the application must take into account the backup of the control system, the so-called set-point station, which may be used to switch between computer control and manual control and which influences greatly the organization of the direct digital control system. This paper also illustrates the variety of input output devices through which a minicomputer must communicate with human beings and the process. The third paper, by E. H. Gautier, M. R. Hurlbut, and E. A. E. Rich, gives an alternative

view of computer control. They stress the experience that has been gained from controls installed in over thirty cement plants and discuss operator communication, interfacing, and hardware and software problems in detail.

The last two papers look at smaller process control systems. G. R. Devine discusses the application of a minicomputer to the control of basis weight and moisture on a paper machine. The important result in this paper is that a dedicated application such as this still requires a rather complete hardware/software control theory system to make it effective. That is, in addition to the implementation of the feedback control algorithms themselves, additional techniques for determining appropriate parameters of the system and design of the resulting controller parameters are necessary ingredients in minicomputer software. This, coupled with operator communication requirements, implies that even in a dedicated application the overall system must be very carefully considered in the design. The techniques discussed in this paper are an illustration of one of the best applications of minicomputers in the process industries.

The last paper by C. P. Pracht illustrates the use of control theory in minicomputers, which cannot be economically applied without a digital computer. This is in contrast to many applications where the computer duplicates the function of an analog control system but perhaps at lower cost. Through the use of fast-time simulation relatively complex problems can be solved readily in an on-line manner.

The overall intent of this part is to illustrate that the variety of applications that can be implemented with minicomputers in the process control area is limited only by one's imagination. However, successful implementation demands a rather thorough systems analysis of the hardware requirements, software requirements, operator communication, hardware and software, and the theory necessary to support the application.

Introducción

Bibliography

[1] "In-plant sensors help schedule work and watch costs in brass rod mill," R. L. Aronson, *Contr. Eng.*, vol. 17, July 1970, pp. 40–43.

[2] "Control problems in papermaking," K. J. Astrom, *1968 IBM Scientific Symp. Control Theory and Applications*, pp. 136–161.

[3] "Goal: five paper machines under computer control," J. N. Bairstow, *Contr. Eng.*, vol. 16, Jan. 1969, pp. 120–123.

[4] "Dynamic control of the cement process with a digital computer system," T. Bay, C. W. Ross, J. C. Andrews, and J. L. Gilliland, *IEEE Trans. Ind. Gen. Appl.*, vol. IGA-4, May/June 1968, pp. 294–303.

[5] "Atlantic Richfield automates for safety and efficiency," W. B. Bleakley, *Oil Gas J.*, Apr. 19, 1971, pp. 110–113.

[6] "The digital computer in real-time control systems," A. S. Buchman, *IEEE Trans. Aerosp. Electron. Syst.*, July 1970.

[7] "Minicomputer grades steel strip on-line," H. S. Drewry and W. R. Edens, *Instrum. Technol.*, Jan. 1971, pp. 49–53.

[8] "True computer systems play a big role in pipeline control," *Oil Gas J.*, June 21, 1971, pp. 130–137.

[9] "Economic justification for digital control of a batch process," R. G. Fritchie and E. F. Schagrin, *Contr. Eng.*, vol. 17, July 1970, pp. 54–56.

[10] "On-line sampling saves valuable ore," F. W. Glow and S. J. Bailey, *Contr. Eng.*, vol. 16, Jan. 1969, pp. 117–123.

[11] "Direct digital control at Lone Star's Greencastle, Indiana, plant," D. L. Grammes, *IEEE Trans. Ind. Gen. Appl.*, vol. IGA-6, Sept./Oct. 1970, pp. 480–487.

[12] "Computer control of motor gasoline blending," R. J. Lasher, *1967 Comput. Conf. Proc.*

[13] "Automation in the steel industry," A. Miller, *Automation*, Nov. 1966, pp. 7–14.

[14] "Survey of real-time on-line digital computers," H. H. Rosenbrock and A. J. Young, *Proc. 1966 IFAC, 3rd Congr.*

[15] "Process performance computer for adaptive control systems," F. A. Russo and R. J. Valek, *IEEE Trans. Comput.*, vol. C-17, Nov. 1968, pp. 1027–1037.

[16] "Minicomputers rethink NC," J. E. Sanford, *Iron Age*, Feb. 18, 1971, pp. 53–57.

[17] "Sequence control for batch polymerization," F. H. Schreiner, *Contr. Eng.*, Sept. 1968, pp. 96–100.

[18] "Bumpless transfer under digital control," R. Uram, *Contr. Eng.*, vol. 18, Mar. 1971, pp. 59–60.

[19] "Modelling and programming for direct digital control," G. V. Woodley, *ISA J.*, Mar. 1966, pp. 48–54.

Systems Engineering in the Glass Industry

RAYMOND J. MOULY, SENIOR MEMBER, IEEE

Abstract—A survey of current trends of systems engineering in the glass industry is presented. The central theme in the systems engineering is the technique through which the process of our time—the information revolution exemplified by the digital computer—is exerting its impact on the industry.

Systems engineering is examined and basic concepts explained, and the production system is defined as a pyramidal, hierarchical structure. Process models which have been developed primarily for control purposes are reviewed; examples of theoretically or experimentally developed models are given. In computer control applications, a major trend is seen toward extensive integrated real-time information-processing systems consisting of several computers interconnected through a communication network. The development of the human components in the production system, particularly management structure, is considered as an essential aspect of the overall system development.

I. Introduction

ABOUT 200 years ago, the invention of the steam engine marked the beginning of the first industrial revolution. The mechanical age had begun, characterized by, in the words of Mokrhan [1], "the technique of fragmentation that is the essence of machine technology," with its emphasis on the individual control of the fragmented parts without marked concern for their interaction and the behavior of the process as a whole.

The mechanical age is now receding. We are living in the "electric age." The information revolution—the process of our time—is taking place, forcing us to reshape and restructure our processes and to move inexorably from fragmented, slow, and informal control practices to a philosophy of global, instantaneous, and systematic control.

These statements provide the background for the survey that follows. It consists of three major parts. First, in Section II, some fundamental systems engineering concepts will be reviewed. Then, in Section III, examples of the application of these concepts in the glass industry will be presented. Finally, in Section IV, the role of human factors in systems engineering will be discussed in a general way.

II. General Systems Engineering Concepts

A. Definitions

What do the terms "systems and systems engineering" mean? There are almost as many definitions as there are writers on the subject. The concept of systems is an ancient one. An early reference can be found in this quotation from St. Paul: "There are many members, yet but one body." A modern definition [2] reads as follows: "A system is any collection of interacting elements that operate to achieve a common goal." Systems engineering is the art or the technique of building systems. This, in itself, would not be a new activity were it not for two factors which characterize systems engineering and set it apart from conventional engineering. The first factor is the formal awareness of the importance of interaction between the parts of a system. The second factor is that systems engineering implies integration. It says that the whole is more than the sum of the parts.

Designing a system consists of translating a task statement into a specification of the system to be built. There are two fundamentally different approaches to the system design problem. They are, as defined by Athans [3], the direct or ad hoc approach and the usual or standard approach (Fig. 1).

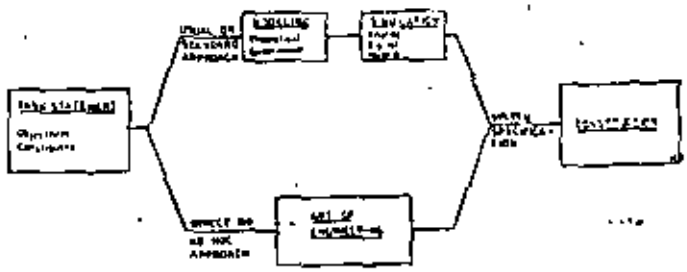


Fig. 1. Physical system design approaches.

The direct approach is often referred to as the art of engineering. It consists simply of building a system which does the job. The direct approach is acceptable for small systems, but as systems become increasingly complicated and extensive, it is frequently inadequate if optimum design is to be achieved. In addition, the risk and costs involved in extensive experimentation might be prohibitive.

The usual or standard approach is the technical or scientific approach; it begins with the replacement of the real world problem by a problem involving mathematical relationships. In other words, the first step consists of formulating a suitable model of the physical process, the system objectives, and the imposed constraints. Simulations of mathematical relationships on a computer often play a vital role in the search for a solution. Various alternative designs can be compared and evaluated. Then, and then only, a system is built.

Presently, the design of a large and complex system is often achieved through the combined use of the direct and the standard approaches. The direct approach is likely to be used in the structuring of the whole system, whereas the standard approach will be taken for the design of

Manuscript received December 21, 1968. This paper was presented at the 5th International Congress on Glass, London, England, July 3, 1968.

The author is with the Technical Staffs Division, Corning Glass Works, Corning, N. Y.

Reprinted from *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-5, pp. 300-312, Oct. 1969.

various components. The standard approach has been extensively used by engineers for the design of control systems.

The manufacturing process is the system we are interested in. I shall discuss its nature from a systems engineering viewpoint and particularly examine the role of the information network and show how it relates to the economics of process control.

B. Hierarchical Process Control [4], [5]

The manufacturing system, whether it be a major process, a plant, a multiplant operation, a company, or even a whole industry, can be looked at as the pyramidal structure shown in Fig. 2, consisting of two distinct elements: the physical process and the controller. The controller's function is to manipulate the plant in order to optimize the process with respect to the manufacturing system objectives.

Somewhat arbitrarily, a hierarchy of three interacting control functions can be identified. At the first level, we find the process control functions which include the single- and multiple-variable control activities usually associated with the control of process units. Evaluation control, at the second level, is the guidance for the utilization of production facilities; it covers such activities as scheduling, inventory control, cost control, and invoicing. The management control functions at the third level include the setting of objectives to be achieved by the system within the constraints of policy.

Paralleling the hierarchy of control levels, we can identify a hierarchy of control functions—regulation, optimization, adaptation, and self-organization—as we move toward the top of the pyramid. It can also be observed that, as we advance toward the higher levels of control, the emphasis on the physical variables decreases as the economic variables play an increasingly important role in the decision-making or control functions.

Other important characteristics of the control system are the decreasing frequency of the controller action and the increasing complexity of the decision-making process as one rises through the hierarchy of control levels. It should also be pointed out that control problems at the lowest level are essentially those of a deterministic system, whereas as one rises through the hierarchy, the nature of the problems becomes increasingly probabilistic.

This hierarchical control structure can be identified in most industrial processes although not always in a systematic form. We find that machines, such as controllers, sequential control systems, etc., are carrying out automatically some of the control functions at the lowest level of control, but also some of the control functions are still carried directly by human beings (process operators, supervisors, schedulers, and managers). All of these controllers, human beings or machines, have one common characteristic: they are processors of information and are part of the information network of the system.

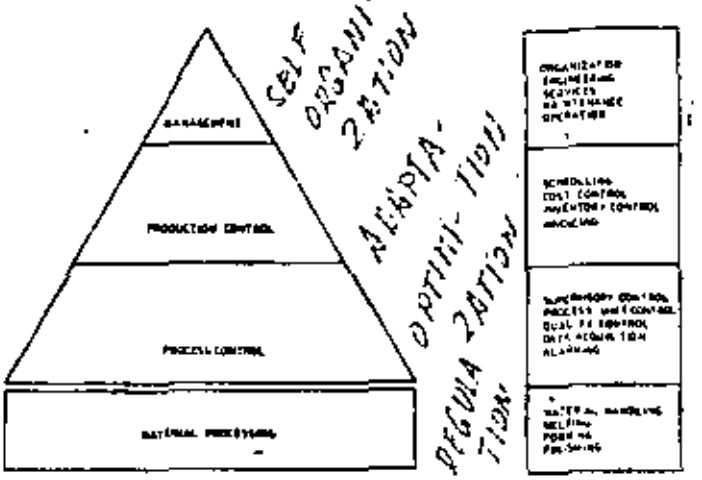


Fig. 2. Plant functions.

The importance of the information network within the manufacturing process cannot be overemphasized. It is the interconnecting tissue which relates the other five process networks: materials, orders, money, personnel, and capital equipment [6].

Efforts to automate process control functions took place initially at the first level of control with the application of process controllers. Little could be done at the higher levels until 20 years ago, when the invention of the digital computer marked the beginning of a new era. This second industrial revolution, the information revolution, which has already deeply affected our concepts of process control, has developed along two somewhat distinct paths. On the one hand, with the availability of data processing machines, attempts have been made to automate part of the control functions at the third level. On the other hand, during the past 10 years, computers have increasingly penetrated the industrial process production control field at the first and second levels.

Today, the availability of reliable on-line process control computers makes it possible to effect in real time the entire information network of the production process and to implement integrated systems that will perform control functions at all levels of the hierarchy. Such systems are technologically feasible. Why should they be implemented? Technological feasibility is not enough. Powerful economic incentives must exist if the technique is to be applied extensively by competitive industries. In order to answer the question, we should examine the nature of the relationship that exists between the processing of control information and the economics of the process.

C. Process Control and Process Economics

We know, intuitively, that there is a relationship between these two subjects, but it is only recently, however, that the quantitative nature of this relationship has been established. Trapeznikov shows in a recent paper [7] that controlling a process consists in ordering information. Any process system left to itself under natural conditions will tend to become increasingly disorderly; the

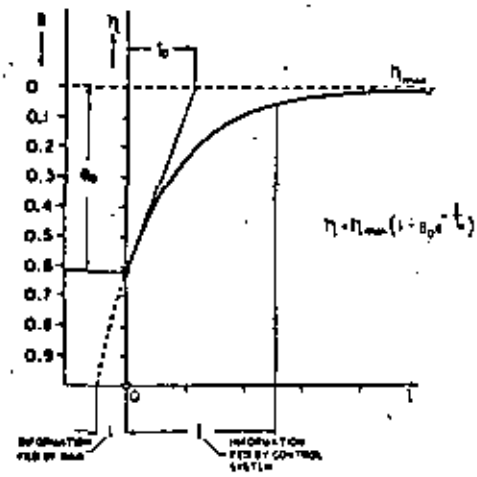


Fig. 3. Process effectiveness—control information curve.

entropy of the system will increase. The purpose of controlling the process is to counteract the growth of disordering. Control is work for ordering.

A fundamental relationship relates the system efficiency to the amount of control information I

$$\eta = \eta_{max}(1 - B_0 e^{-(I/I_0)})$$

B_0 being the measure of the degree of disorder in the system associated with the amount of control information I_0.

Efficiency should be taken here in a very general sense, and in particular, it can be looked at as profit. The relationship, illustrated in Fig. 3, can be looked at as a formal expression of the "law of diminishing returns" or of the "cost-effectiveness" relationship applied to control systems. It is quite similar to the familiar S-shaped relationship between return and effort expressed in monetary units.

Important practical conclusions can be drawn from these considerations:

1) Process effectiveness increases rapidly at first with increasing knowledge, but because of the basic nonlinearity of the relationship, the investment in control should not exceed a certain economically justifiable level.

2) In order to achieve the maximum overall effectiveness it is necessary to attain the same degree of effectiveness at all levels.

3) So far the automatic control of information at the higher levels has received little attention as traditionally, the major function of instrumentation and control engineering has been to increase the ordering of information at the process control level, the first level of the control hierarchy. The automatic coordinated control of major units has not progressed as rapidly, basically because until recently no control tools were available to process reliably control information in real time. It should, consequently, be noted that the economic potential of automatic process control at the higher levels could be high because of the inherent high information utilization usually found at these levels of control.

III. Systems Engineering in the Glass Industry

I shall now review specific examples of applications of systems engineering concepts in the glass industry. I shall focus on two subjects—process modeling and computer control systems.

A. Process Models and Modeling Techniques

The plant or process is the central and most fundamental issue. In process control, knowledge of process behavior comes first. Models which represent the essential aspects of the process are needed in order to apply the standard approach to systems design.

A model is defined as "a quantitative or qualitative representation of a process or endeavor that shows the effects of those factors which are significant for the purpose being considered" [8]. We shall not consider either physical scale models, such as tank models using viscous solutions [9]–[11], or activity models, such as PERT, but will discuss only models in which mathematics is used to describe the salient features of the process behavior and which are intended primarily for use in the synthesis of control systems. The mathematical relationships of interest are those which relate the process inputs, manipulated variables, and disturbances to the intermediate variables and outputs (Figure). It is essential for process control problem applications that these relationships account for the dynamic behavior of the system.

Models can be classified as experimental or theoretical according to the techniques through which they are developed. Experimental modeling [10] requires the observation of the process variables in order that the state of the process may be recorded under a variety of conditions. Intentional perturbation of the process through the manipulated variables and inputs is usually necessary to obtain accurate relationships. The trend is toward increasing use of automatic data acquisition and processing techniques to determine the quantitative relationships that exist between the process variables.

In theoretical modeling the mathematical description of the process is built by writing the exact equations which govern the behavior of the process, such as conservation of mass, energy and momentum and the fundamental equations of heat transfer and fluid flow.

In any case, the validity and usefulness of the model generally depend heavily upon the ingenuity of the model builder, his clear understanding of the purpose of the model and his prior knowledge of the process.

Several examples of experimental and theoretical models developed for the design of control systems in the glass industry will be reviewed in the following.

Bubbling Tubing Process Model [13]: This simple example is one of an experimental model. The problem is to develop an automatic thickness measurement for tube drawing processes in the manufacture of fluorescent tubing.

The process is shown in Fig. 5. Glass is delivered to the forming process through a refractory ring placed at the

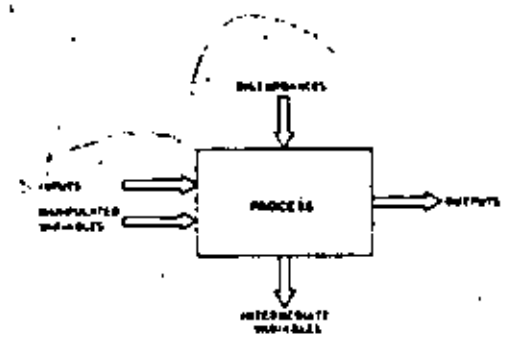


Fig. 4. Basic process.

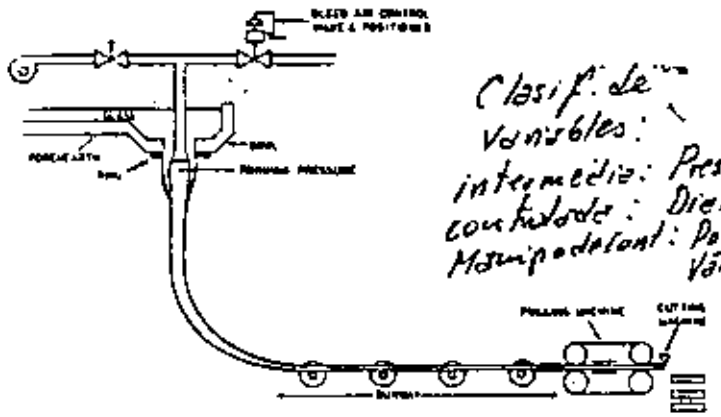


Fig. 5. Vello tubing process.

bottom of the bowl. Air is blown through a pipe in the center of the ring while the tubing is drawn by a pulling machine. At the end of the runway, a cutting machine cuts the tubing into tubes of proper length.

The experimental mathematical model used to describe this process consists of two parts. The first part is a set of linear incremental differential equations expressing the relationships between the manipulated variable, valve position, the intermediate process variable, forming pressure, and the controlled variable, diameter. The equations given below were obtained by experimental step-response techniques.

$$\frac{\Delta \text{ forming pressure}}{\Delta \text{ valve position}} = \frac{T_1 s + 1}{\alpha T_1 s + 1} \cdot \frac{K_1}{T_2^2 s^2 + 2\zeta T_2 s + 1}$$

$$\frac{\Delta \text{ diameter}}{\Delta \text{ forming pressure}} = K_2 e^{-\tau s}.$$

The second part of the model is the statistical description of the controlled variable. This description is in the form of power spectra and histograms. The power spectra, Fig. 6, characterize the way the diameter variations occur. Significant diameter variations still take place at the process natural frequency, 1.0 cycle/min. Consequently, an effective automatic control system must control diameter variations occurring up to this frequency. This information on the statistical nature of the process provides a basis for the intelligent selection of instrumentation and a means for evaluating the expected improvement in process performance that would result from the implementation of a given control system.

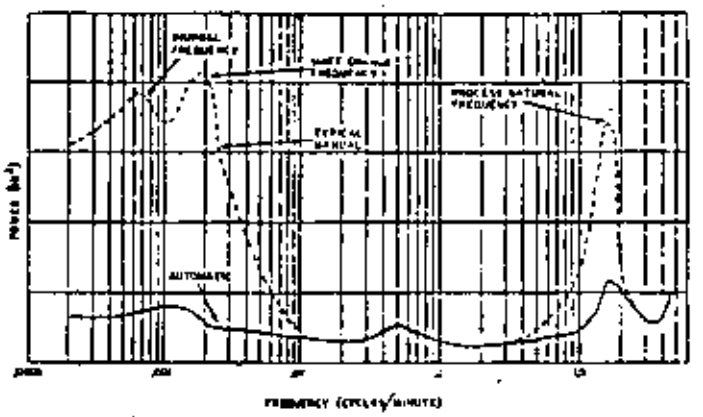


Fig. 6. Power spectra—manual and automatic control of tubing diameter.

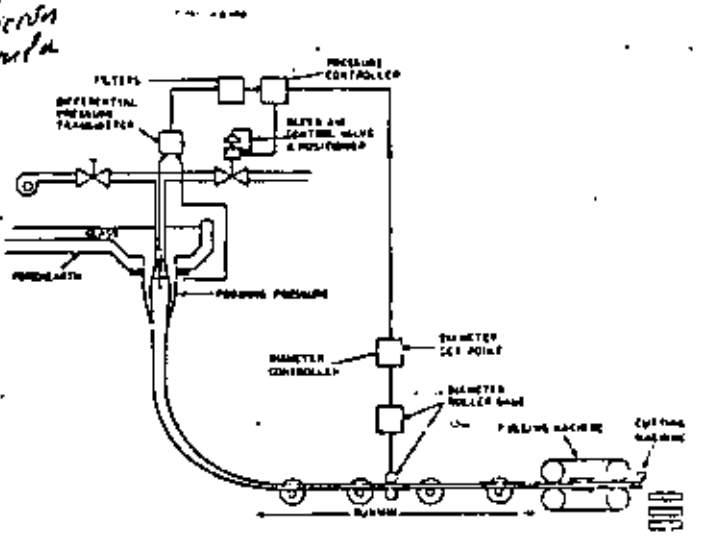


Fig. 7. Vello tubing process with automatic diameter control system.

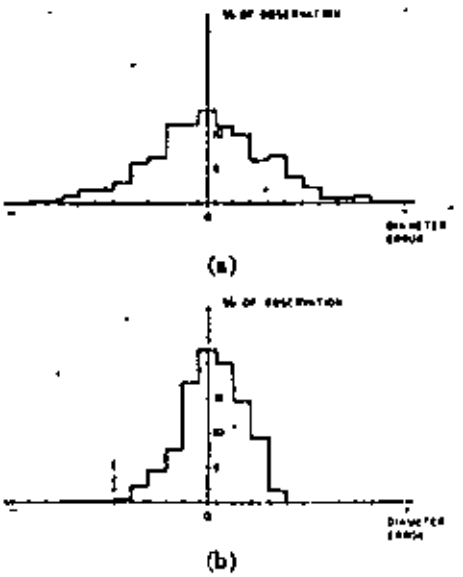


Fig. 8. Histograms of diameter error (a) Manual. (b) Automatic diameter control. (Note: σ automatic = 0.5 σ manual.)

These process models, incremental difference equations, and statistical models were used in an analog computer simulation to evaluate a number of possible control system configurations. The control system of Fig. 7 was selected; it is a cascade control system in which the forming pressure is controlled by a high-gain, large-bandwidth loop, and the diameter is controlled by a low-gain, low-bandwidth loop.

The histograms, big improvement in the performance of the system under manual and automatic control. It is seen that the automatic control system reduces the diameter variations by 50 percent.

2) *Ribbon Machine Process Model* [14]: Two models were developed in connection with the design of a computer system for the automatic control of the dimensions of bulbs made on a ribbon machine (Fig. 9). These models which account for the process behavior, including the quality control sampling procedures, were used in a digital computer simulation to evaluate alternate control strategies.

The first model is the matrix in Fig. 10. It was determined experimentally and represents the relationships that exist between the most significant process variables.

The second model was developed for the analysis of the particular control problems resulting from the fact that only small, relatively infrequent samples of the end product quality can be obtained for feedback control. This problem was investigated in a digital computer simulation study of a one-variable control loop with quality data used as the feedback measurement and a long transport delay as the significant process dynamic element. The process disturbances were simulated by the sum of an assignable periodic disturbance and a random disturbance. This study indicated that the sample mean was the best indicator of average process performance and that the stability of the system in response to the assignable disturbance depended only upon the control system design parameters.

The computer process control system schematic in Fig. 11 was developed on the basis of these studies. The automatic control of the low-frequency components of the error signal resulted in a significant reduction of the variability of the product dimensions.

3) *Glass Tank Model*: Another example of experimental modeling is given by Hoelzle in his investigation of the dynamics of a glass tank [15]. When the composition of the bath in a continuous tank is changed abruptly, there will follow a change in the glass composition at the outlet of the tank. Comparing the step response of composition to the step change in batch, the transfer function for composition of the melting furnace may be determined; two cases are considered with and without outlet return.

In general, the transfer function may be approximated by a transportation lag T_d and a first-order system with time constant T_c. Hoelzle's investigation indicated that the transfer function could have a dominant time constant T_c, which is about ten times the

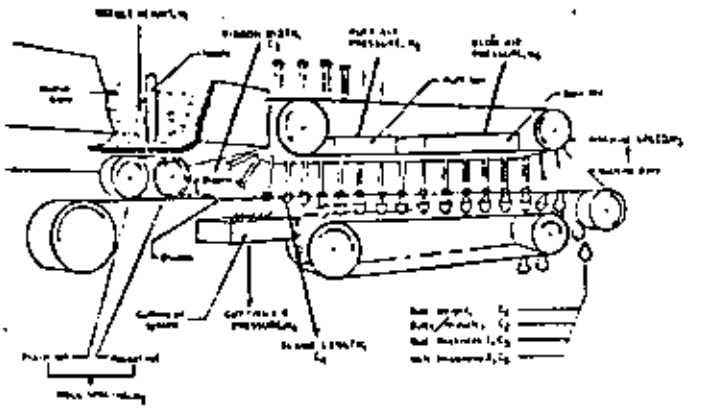


Fig. 9. Ribbon machine process.

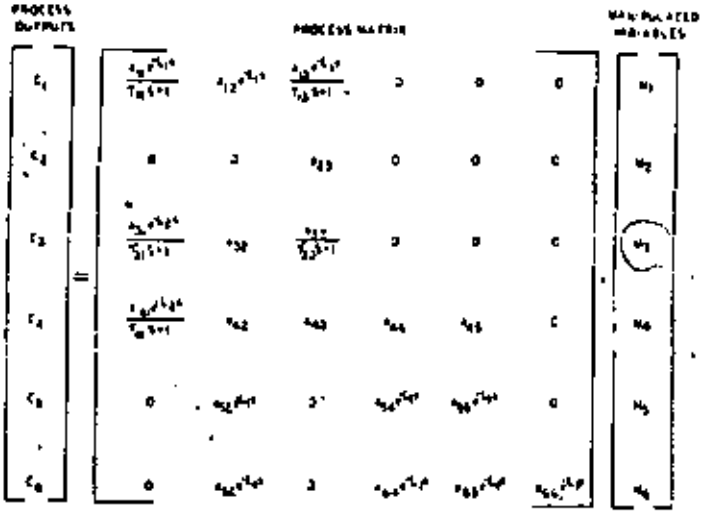


Fig. 10. Ribbon machine process model.

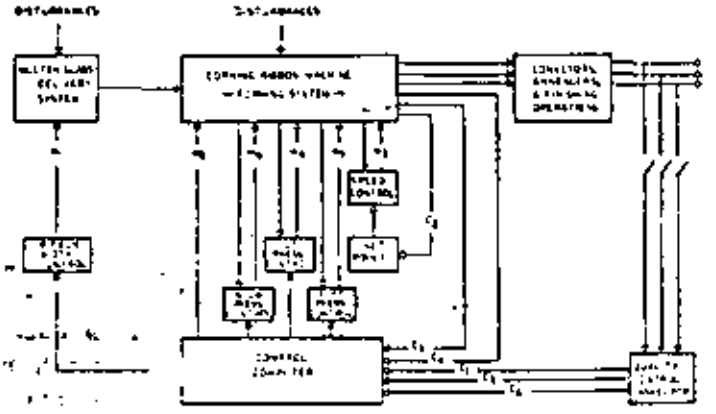


Fig. 11. Ribbon machine process computer control system.

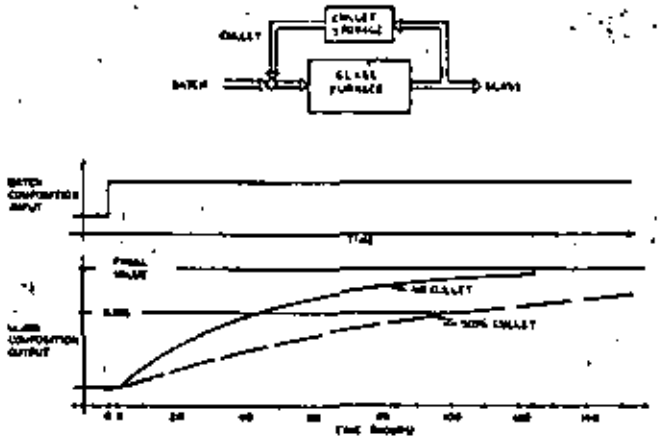


Fig. 12. Glass composition response to a step change in batch composition.

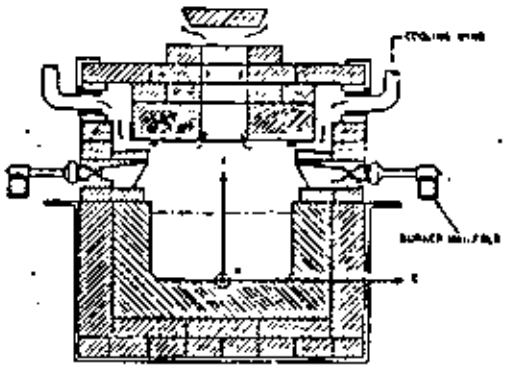


Fig. 13. Forehearth channel—cooling zone cross section.

of the glass in the furnace. The mean residence time can be estimated by dividing the furnace glass capacity M by the average glass output (pull) Q. Comparing T_L, τ, and T_M can give some idea as to what extent the glass is ideally mixed. The derivative of the step response gives the residence time distribution of the glass. Fig. 12 illustrates some experimental results.

For a furnace with a glass capacity of 200 tons and a pull of 96 tons/day, T_M = 50 hours; the transfer function without cullet return consisted of a transportation lag T_L = 3 hours and a time constant τ = 40 hours. With a cullet return of 50 percent after 20 hours, the transportation lag was 3 hours as before, but the time constant increased to 100 hours.

b) Forehearth effects: The forehearth model developed by Duffin and Johnson [4] illustrates the methodology used to construct a theoretical model based on physical laws of nature. The development of a theoretical model usually involves the following steps: 1) formulate the system equations based on physical laws, 2) apply appropriate boundary and initial conditions, and 3) solve the equations by analytical or numerical means.

The forehearth delivers the glass in an open channel from the furnace to the forming machine and conditions the glass to a predetermined delivery temperature by means of wind cooling and gas heating as shown in Fig. 13.

a) Formulation of system equations: The basic energy equation—The general differential equation for heat transfer of a flowing stream of molten glass is derived from the principle of conservation of energy. By taking an energy balance on a differential volume element of dimensions dx, dy, dz, the energy equation is

$$\underbrace{\frac{\partial}{\partial y}\left(k'\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(k'\frac{\partial T}{\partial z}\right)}_{\text{rate of energy input by conduction and radiation}}-\underbrace{\frac{\partial}{\partial x}\left(\rho C_p V_x T\right)}_{\text{rate of energy input by mass flow}}=\underbrace{\rho C_p\frac{\partial T}{\partial t}}_{\text{rate of accumulation of energy}}. \tag{1}$$

In deriving (1), the following assumptions are made.

i) Heat flow by radiation can be regarded as being due to a "radiation conductivity" of $8T^3/\alpha$, where T is the absolute temperature and α is the absorption coefficient for the energy of wavelengths corresponding to temperature T. The factor k' in (1) is defined as the true conductivity plus radiation conductivity.

ii) The effective conductivity k', density of glass, and the specific heat of glass C_p are not temperature dependent (hence not a function of the space coordinates).

iii) The velocity V_x in the x direction (direction of flow) is not a function of x. Thus (1) reduces to

$$\frac{k'}{\rho C_p}\left[\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right]-V_x\frac{\partial T}{\partial x}=\frac{\partial T}{\partial t}. \tag{2}$$

Equation (2) is applicable only in the interior of the glass. To completely specify the system, appropriate boundary and initial conditions must be supplied. These are the following.

i) The temperature distribution on the glass-refractory boundaries at the bottom ($y = 0$) and the sides ($z = \pm W$) of the channel are assumed to be time-invariant and linear functions of the space coordinates.

$$\begin{aligned} T(x,0,z) &= \phi_2(x,f) \text{ is specified} \\ T(x,y,w) &= \phi_3(x,y) \text{ is specified.} \end{aligned} \tag{3}$$

ii) At the interface between the glass and the gas ($y = d$), the boundary is a radiating boundary where the glass is exchanging radiant energy with the channel enclosure (refractory crown). Further, the gas in the space between the glass and the crown also exchanges heat with the system through convection and radiation. The equation at the glass-gas interface is again derived based on energy balance

$$k'\left.\frac{\partial T}{\partial y}\right|_{y=d}=\sigma F\left[T_{\text{crown}}{}^4-T^4\right]-h(T-T_{\text{air}}) \tag{4}$$

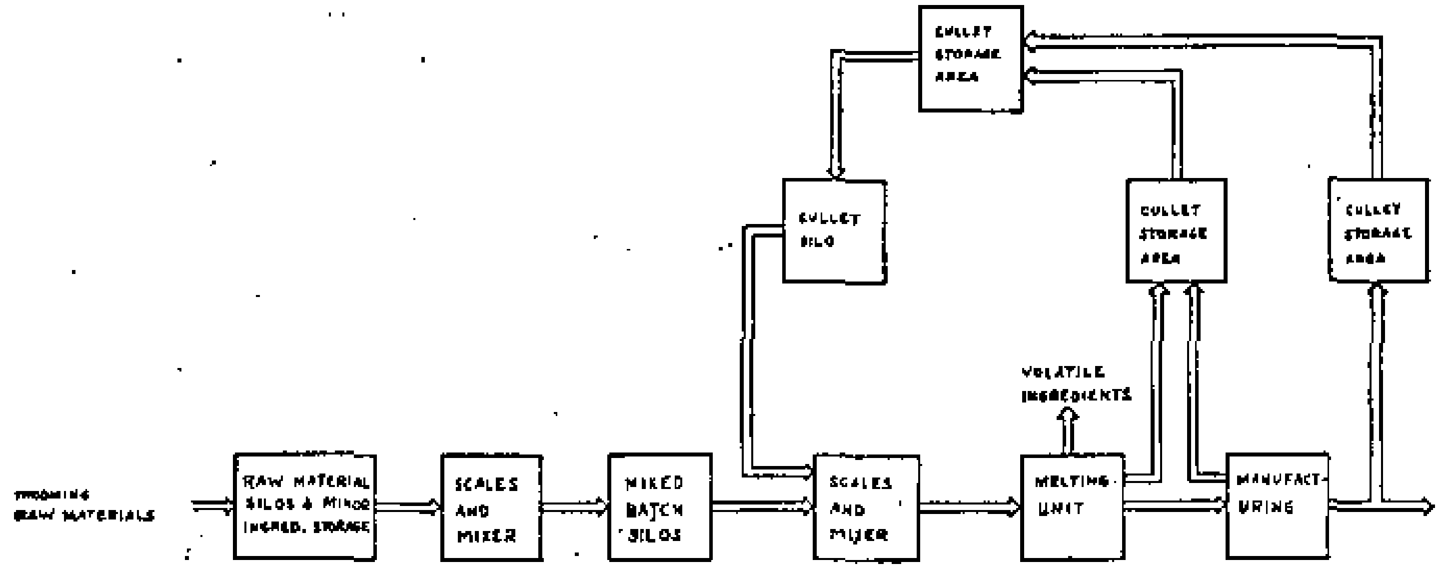


Fig. 14. Melting system schematic.

where

- σ Stefan Boltzman constant
- F view factor derived with the assumption that the glass surface and the crown are two opposite infinite parallel planes
- h gas heat transfer coefficient
- T_{crown}, T_{gas} these temperatures are inputs to the model and must be either assumed or determined by measurement on actual forehearths.

iii) Since glass temperature is symmetric with respect to the center of the channel ($z = 0$),

$$\left.\frac{\partial T}{\partial z}\right|_{z=0} = 0. \qquad (5)$$

iv) At time zero, the temperature distribution at some location X must be specified as an initial condition. Usually the temperature distribution at the inlet to the forehearth is given.

b) *Numerical solution:* Equations (3)–(5) and the appropriate initial conditions completely specify the system. The equations are partial differential equations of the parabolic type with nonlinear boundary conditions. Because of the complexity of the problem, an approximate numerical solution is the best that can be obtained. The equations are written in a finite-difference form and can be solved on a large digital scientific computer.

c) *Application of the model:* This model is applicable to the systematic design of a temperature control system for an existing forehearth. Studies can be made with the model to evaluate control systems which will deliver glass at constant temperature to the forming machine in the face of disturbances in the inlet glass temperature, ambient temperature, and glass flow rate changes.

5) *Melting system models:* One of the earliest examples of the application of modeling techniques to the analysis of control system problems in the glass industry is given by Oppelt [17]. His paper presents a conceptual elementary multivariate dynamic model of a glass tank and suggests improved control strategies using feedback and feedforward techniques.

One last example of theoretical modeling concerns the melting system illustrated in Fig. 14, consisting of raw materials input and storage, batch mixing and storage, melting, cullet recycle, and control systems. The study made by Sting [18] is important in that it develops models for process units, such as storage silos, mixers, etc., and demonstrates the use of these models in the analysis of systems design and operation through simulation.

The first step in approaching the problem is to construct mathematical models for all the process units by taking one of the most important aspects of the entire process into consideration: the physical transformation of granular material.

A general model is developed which, when specialized, can be used to model silos, mixers, and mixing tanks along with other process components. This general model will be described briefly for a silo.

A silo is defined as a temporary storage device whereby granular material is dumped into the top, stored, and at some later time removed from the bottom. The model was developed under the following reasoning.

a) The filled silo is divided into spaces of batch volume size (refer to Fig. 15).

b) Associated with each space is a corresponding batch and its describing constituent vector.

c) When a batch is removed from the bottom, all the batch constituent vectors above it move down one space.

d) When the material is either entered or extracted, it is done discretely in time.

e) Because of the mixing effect between adjacent batches, the output batch is some combination of any input batch.

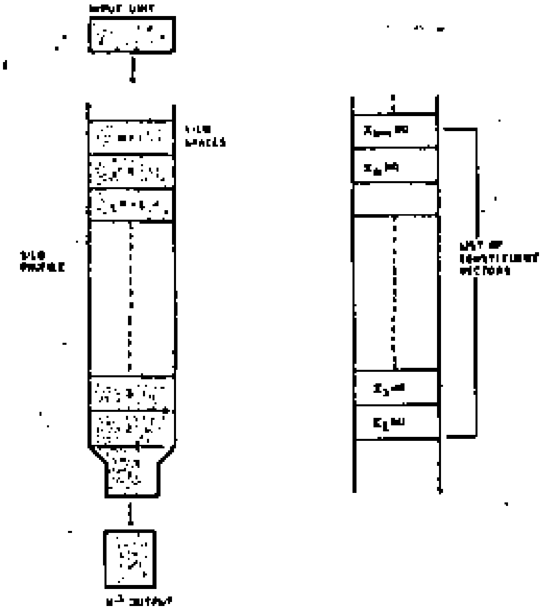


Fig. 15. Schematic silo.

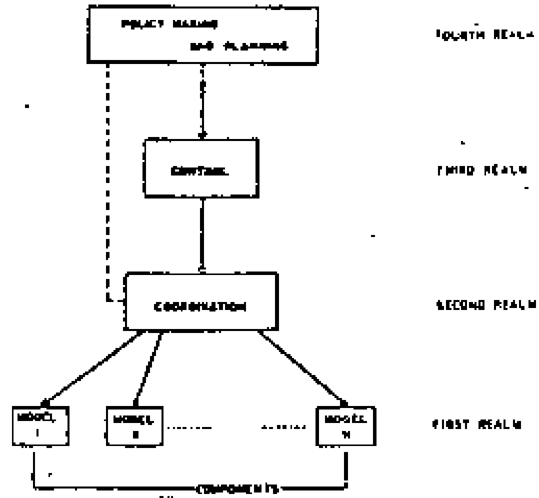


Fig. 16. Activity realms.

f) All materials which are placed in the silo together have equal or nearly equal densities.

g) A batch of materials, or any part thereof, has a maximum and a minimum length of the silo to transverse, and this transversal occurs within some maximum and minimum number of output batches.

These assumptions, together with mass and impulse balance, yield the following set of equations:

$$Y(K) = \sum_{i=1}^{m} W_i(K) - X_i(K) \tag{6}$$

$$\sum_{i=1}^{m} W(K) = 1 \tag{7}$$

$$\sum_{i=1}^{m} W_i(K - i + 1) - X_i(K - i + 1) = X_1(K) \tag{8}$$

$$X_1(K) = X_2(K - 1) = X_3(K - 2) = \cdots = X_m(K - m + 1). \tag{9}$$

By substituting (9) into (8), then rearranging it, there results

$$W_1(K) = 1 - \sum_{i=2}^{m} W_i(K - i + 1) \tag{10}$$

where

- $X_i(K)$ constituent vector of the material at the ith position in the compartmentalized silo, just prior to the kth output
- $Y(K)$ Kth output batch constituent vector
- m maximum range over which an input batch will be spread over the output batch
- $W(K)$ the weighing value which designates the percentage of inputs that are in the output at time NT.

The weighing values are assumed to be of a statistical nature. The particular disturbance associated with the random variables of the model is dependent upon the particular silo to be modeled and the material to be stored. Thus the weighing values not only must satisfy the constraints imposed by (7) and (10), but also must be generated in accordance with the information extracted from the actual data obtained by conducting experiments on a particular silo. Once the weighing values are determined, (6) can be used to express the physical transformation taking place between input and output batches within the silo.

The second step is to combine all the component models into a "multiactivity system." Broadly defined, the model is composed of four activity realms (Fig. 16). The first realm defines the functions of the components of the process. The second realm defines the interactions and performs structure coordination. The third realm defines the supervisory functions (control), and the fourth realm defines the policy making and planning functions.

The complete system model for batch systems is amenable to digital computer simulation and has been used to investigate process design and control problems.

6) *Conclusions*: As is the case in other process industries, it appears that the lack of [illegible] process models still remains the major obstacle to the implementation of advanced control systems in the glass industry. As a rule, [illegible] control concepts are applied.

Although experimental techniques probably offer the best practical short-term approach to the problem of process modeling, the [illegible] modeling of process units offers very attractive long-term advantages, especially when the [illegible] modeling can be combined with [illegible] unit design. Although the [illegible] is [illegible] the potential gain in the [illegible] process systems is high.

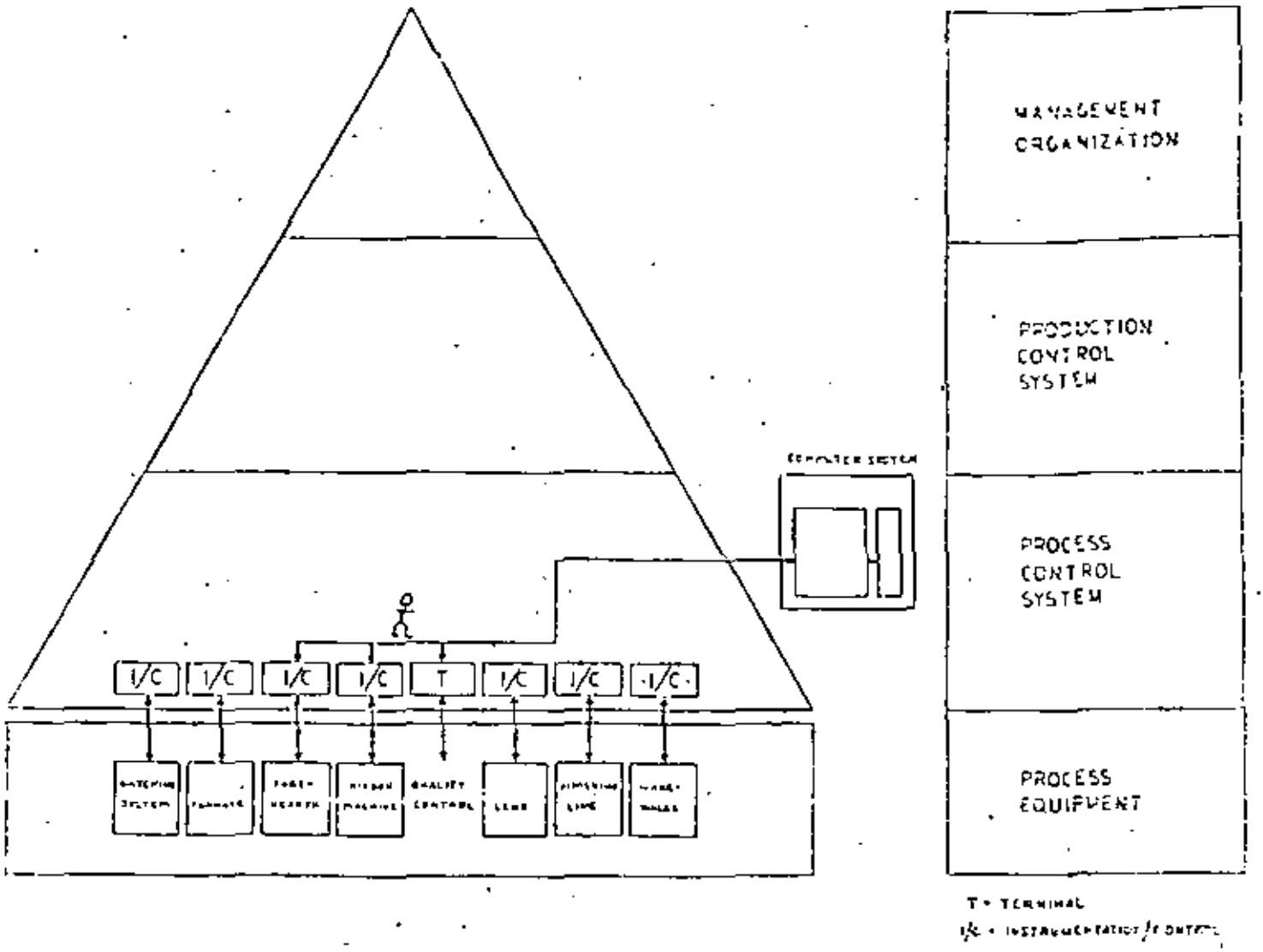


Fig. 17. Plant process control computer system.

Finally, [illegible] technology for large systems consisting of [illegible] process units. [illegible] and [illegible] operation in the [illegible] process.

[illegible] Control Systems

The essential role played by the controller of the manufacturing process, the information network, was discussed in Section II-B. It was stated that the computer technology makes it now possible to automate control functions at all levels of the hierarchy. It is within this framework that we will now survey, on the basis of scarce published information, the status of the implementation of such systems in the glass industry.

One of the first computer control systems implemented in the glass industry was mentioned in the section on process modeling (Section III-A). It is the process computer control system developed for the automatic control of a ribbon machine [14]. This system performs control functions only. The structure of the system is depicted in Fig. 17. Quality control information is entered manually and processed by a process control computer which in turn manipulates a number of variables on the forehearth and ribbon machine.

Another example of process computer control application is given by the control system used in the plants of the Owens-Corning Fiberglas Corporation. On the basis of published information, it appears that these systems are essentially process control systems performing first level control functions in the melting and delivery areas of the process, although some production scheduling might be effected in some instances [19]–[21].

Other supervisory control applications have also been announced recently by glass container manufacturers [22], [23]. Computer control systems are being used for the control of batching, melting, and inspecting operations at the Lakeland, Fla., plant of Owens, Illinois. The function of the computer is to supervise and monitor the entire process.

[illegible] process control [illegible]

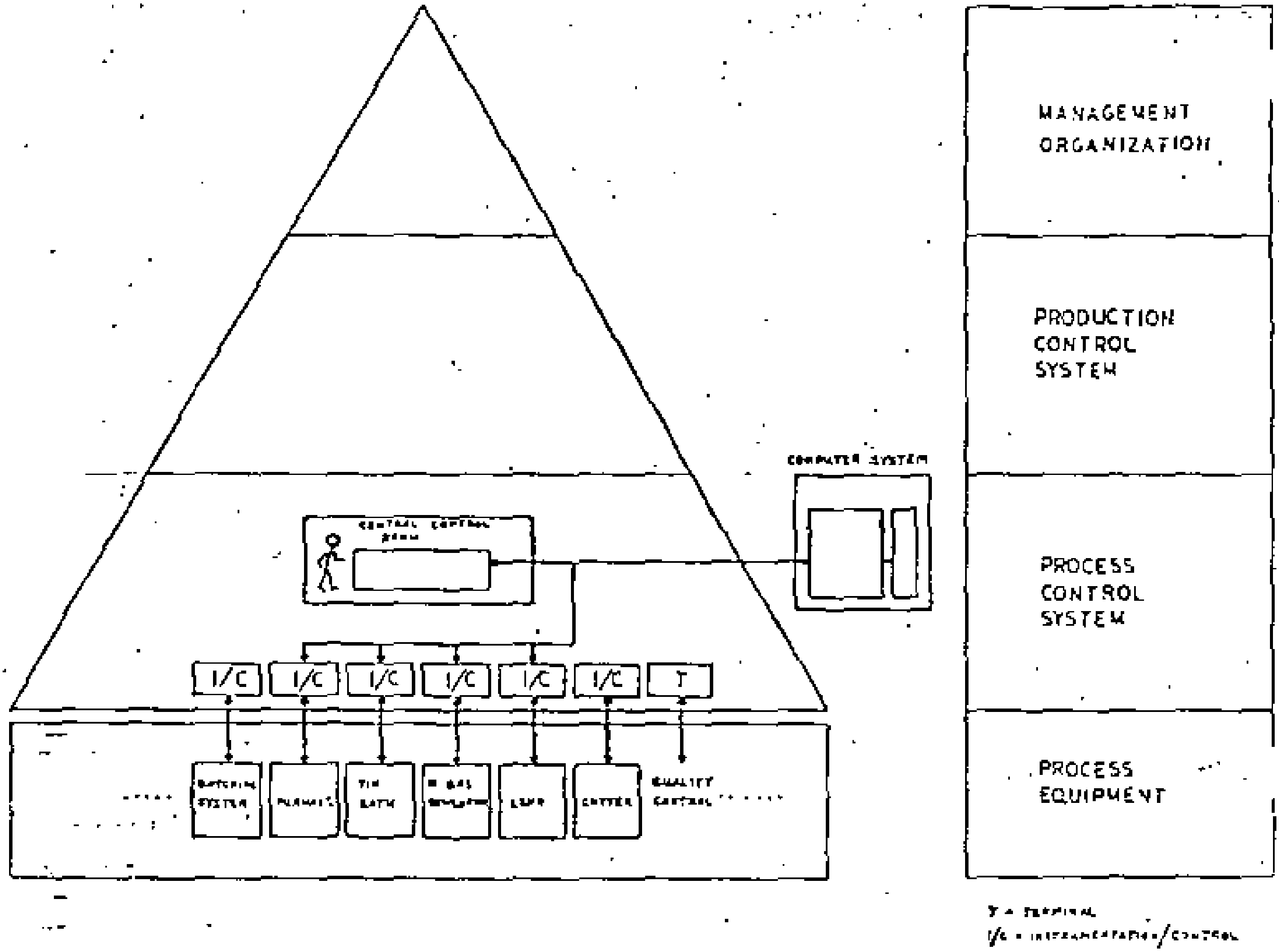


Fig. 18. Plant process control computer system with central control room.

[illegible] process control [illegible] the Ford Motor Company's process control computer system installed in Dearborn, Mich. The process computer control system controls a float glass manufacturing process [24], [25]. The system as illustrated in Fig. 18 handles approximately 50 closed control loops and monitors close to 500 process variables. The [illegible] functions cover the [illegible] furnace, tin bath, annealing [illegible], and [illegible]. M[illegible] and the quality inspection is also effected. The system which results in reduced manufacturing [illegible] improved quality [illegible] of handling [illegible] generation of new programs [illegible] applications [illegible] process. The central control room is represented in Fig. 19. The operator console, on-line printer, alarm typewriter, television display and recording devices, and graphic panels can be identified. The scarcity of recording instruments is apparent.

On the basis of these examples, it would appear that the glass industry, following the trend pioneered by other

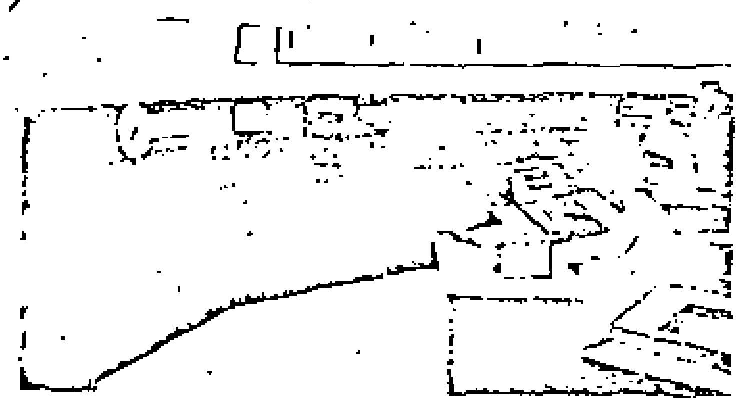


Fig. 19. Central control room—Ford Motor Company (Dearborn, Mich.).

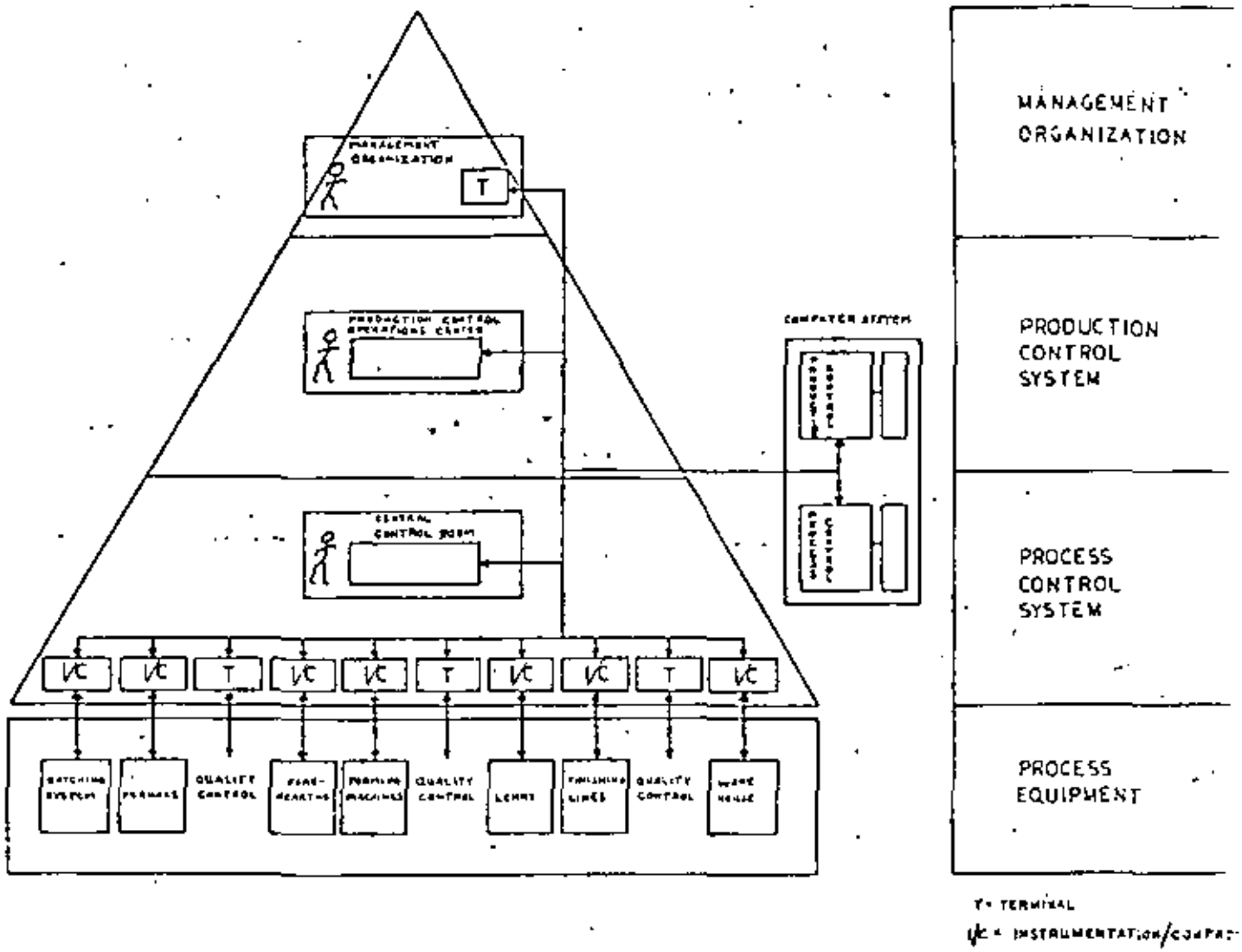


Fig. 20. Integrated plant control system.

process industries, is slowly moving, in an evolutionary fashion, toward computer-directed, central, process control systems.

It is believed that the trend toward integration will not stop at the process control level, but that production control and management control functions will progressively be included into the design of fully integrated on-line, real-time control systems. The diagram in Fig. 20 illustrates the structure of a possible integrated plant control computer system based on functional design. It is an integrated system because it performs both the process and production control functions on line and in real time. The information flow, data collection and report generation are highly automated. The current status of the entire plant is available on a minute-to-minute basis. This permits the effective implementation of advanced management techniques with decisions made on the basis of quantitative information available where and when needed. There is no evidence that such integrated control systems are in operation today although, as we mentioned previously, some of the existing control systems might already have developed to include some production control functions.

The series of diagrams, the last one in particular, al suggests a clear trend toward making computing pow available as a utility throughout the system in much th same way as electric power is available today.

The integrated control systems approach should nat rally be expected to affect our basic concepts of plant sign and operation. In particular, it should be expected have a very significant impact on the management a organizational structure of the plant. This is the sub, that will be discussed in the following section.

IV. Human Factors [26]–[29]

The emphasis of this survey has been so far on th economic and technological aspects of systems develop ment in the glass industry. We have discussed problem relating to the development of the automatic control lo represented by the diagram in Fig. 21, symbolizing t physical process controlled by an on-line computer. M manufacturing systems are man-machine systems, orga zations whose components are men and machines, tied by communications network, working together to achieve common goal. Even in highly automatic computer con systems, the place of the human remains vital as Fig

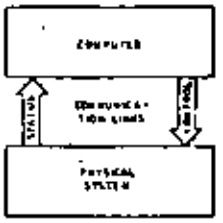


Fig. 21. The automatic control loop.

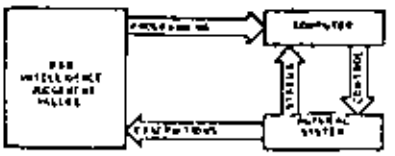


Fig. 22. The man loop.

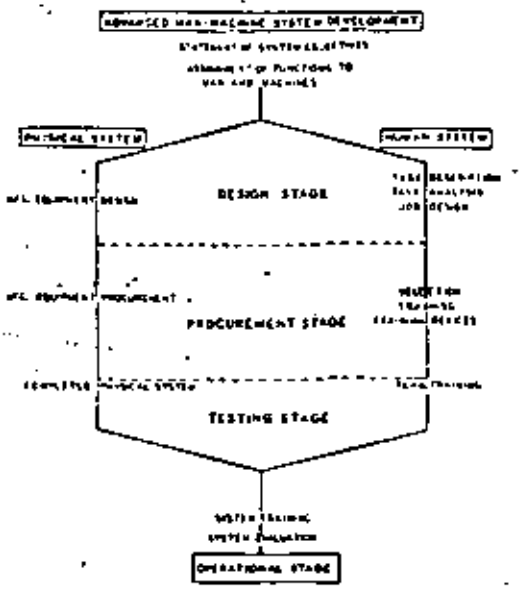


Fig. 23. Man–machine system development.

suggests. Man communicates with the system through programming, manual data entry, and instrumentation. He further observes the process to evaluate through the use of his intelligence, judgment, and values, the performance of the automatic control loop in relation to his criteria of adequate or optimal system performance.

Of particular concern today to those involved in development of integrated control systems is the need for an awareness of man as a component in man–machine systems whose developmental needs resemble those of the machine or hardware components. Planning for the design and development of human components of systems has not been as systematically pursued in the past as it might have been. Characteristically, systems were designed and developed first, and assumptions were made that the human components required either existed or could be found or could be trained to operate this system. Only in relatively recent years, especially with the advent of extremely complex military and aerospace systems, has an increasing awareness developed of the need for systematic design and development of human system components. In the United States this is probably best reflected in the types of documents required of potential contractors for the development of complex man–machine systems. Qualitative and quantitative personnel requirements information (QQPRI) documents which specify the design and development of the personnel subsystem necessary for implementation, maintenance, and operation of these complex systems are required. This can no longer be an evolutionary development process. It must be planned and designed as the physical subsystem(s) is.

Fig. 23 schematically represents the man–machine system development cycle. Advanced system development involves the initial statement of system objectives and culminates in decisions leading to the assignments of operational functions to men and machines. From this point on, the human and physical systems proceed on parallel courses of development to the point at which the completed components are assembled for testing and training in preparation for operation. Although not specified in Fig. 23, the parallel development of the two major subsystems does not in any way imply independent development. One of the primary values to be gained from such a man–machine systems development approach lies in the repeated and ongoing interaction of the developers of the two subsystems at each point in the development cycle. In addition to assuming that all required components are available at a specified end point, the continuing interactions contribute immensely to preventing the need for costly and time-consuming retrofittings of components and major system modifications. To accomplish this, however, implies the development of an ability to communicate effectively and interrelate on the part of representatives of diverse disciplines. Compromises and trade-offs will be required. Ultimate optimization of each subsystem will undoubtedly not be possible, but total system optimization and effectiveness will be more closely approximated.

In light of what has been said about integrated process control possibilities in the future, what are some of the implications for human system components? The implications are numerous. Just a sampling would be the following: 1) Traditional organizational structures may be inappropriate for the management of integrated control complexes either because they are too entrenched or because their traditional control concepts are outmoded. 2) Routine, nonmotivating jobs may be eliminated entirely, resulting not only in a smaller but in a more involved, committed, and motivated workforce. 3) General technical and educational backgrounds of higher levels will be required, and programs and methods to prepare individuals for performance of the man functions in the system will have to be developed. The relative status of various jobs (e.g., machine operators and maintenance employees) may be modified, with attendant needs for modification of long-standing attitudes and opinions. 5) The traditional protection and security functions of large organizations may no longer be required, leading to either a drastic reduction or elimination of the need for such functions entirely.

Whatever the end product of an integrated plant or company control system turns out to be, it is almost certain to require different approaches to the organization, management, development, and maintenance of the human components. What is implied in this paper is that planning for and awareness of the need for such an integrated approach to the human component development, along with the physical system developments, must begin now if we are to achieve the higher levels of integrated control in this reasonably near future.

V. Conclusions

In this survey we have discussed some of the economic, technological, and human aspects of systems engineering. We see systems engineering as the technique through which the electric technology, exemplified by the digital computer, is being applied to our industry.

Several major trends that characterize the evolution of systems engineering technology in our industry have been identified:

1) There is a marked trend toward the increased integration of process control, production control, and management control functions.

2) Modeling techniques are playing an increasingly important role and should lead to the design of optimum systems through the integration of the design of the process and of its control system.

3) The importance of human factors cannot be overemphasized. Our understanding of these factors is one of the major elements, possibly the most important one, controlling the rate of implementation of modern technology in industry.

As engineers, we find ourselves increasingly moving in a position to influence directly social and human patterns. The nature of our work must change as our essential responsibility becomes one of education of the public in modern technology.

References

[1] M. McLuhan, *Understanding Media: The Extension of Man*. New York: McGraw-Hill, 1964.
[2] S. E. Elmaghraby, *The Design of Production Systems*. New York: Reinhold, 1966.
[3] M. Athans and P. L. Falb, *Optimal Control*. New York: McGraw-Hill, 1966.
[4] K. Chen, "Models for integrated control," presented at the 1967 Systems Engineering Conference, Chicago, Ill.
[5] I. Lefkowitz, "Multilevel approach to the design of complex control systems," presented at the 1967 Systems Engineering Conference, Chicago, Ill.
[6] J. W. Forrester, *Industrial Dynamics*. New York: Wiley, 1961.
[7] V. A. Trapeznikov, "Control, automata, technical progress," presented at the 1966 IFAC Cong., London, England.
[8] H. Chestnut, *Systems Engineering Tools*. New York: Wiley, 1965.
[9] I. M. Sheinkop and L. S. Belousova, "Models of liquid for investigating glass movement in gas-operated furnaces," *Steklo i Keramika*, vol. 23, pp. 24–25, February 1966.
[10] J. C. Hamilton, R. R. Hough, and W. B. Stevenson, "Improved techniques for studying the design and operation of glass melting furnaces by means of models," *Advances in Glass Technology, Proc. 1962 Internatl. Conf. on Glass*, (Washington, D.C.), pp. 190–195.
[11] J. D. McClelland, "Plastic flow model of hot pressing," *J. Am. Ceramic Soc.*, vol. 44, p. 329, 1961.
[12] P. Eykhoff, P. M. E. M. van der Grinten, H. Kwakernaak, and B. P. Valtman, presented at the 1967 IFAC Conf., London, England.
[13] R. J. Mouly and L. A. Zangari, "The development of an automatic diameter control system for glass drawing processes," *ISA Trans.*, vol. 3, pp. 158–161, April 1964.
[14] A. T. Bublitz, R. J. Mouly, and R. L. Thomas, "Statistical feedback squeezes product variations," *ISA J.*, pp. 55–60, November 1966.
[15] B. J. Hoelink, "Process dynamics of a glass furnace following a step change of one of the batch components," presented at the 1968 Internatl. Cong. on Glass, London, England.
[16] J. Duffin and K. Johnson, "Glass container process: forehearth simulation," IBM Corp., Systems Development Div., San Jose, Calif., Rept. 02-472-1, June 1965.
[17] W. Oppelt, "Regulating processes in furnaces and their representation with the help of block diagrams," *Glastechnische Berichte*, vol. 26, no. 5, pp. 149–156, 1953.
[18] D. Sting, "Grantar batch process models," Case Institute of Technology, Cleveland, Ohio, Progress Rept. 19, October 1965.
[19] P. D. Griem, Jr., "Digital computers for glass process control," presented at the 1967 Am. Ceramic Soc. Meeting, New York.
[20] ——, "Direct digital control of a glass furnace," presented at the 20th Ann. ISA Conf., Los Angeles, Calif., 1965.
[21] R. R. Hudgins, "Fiberglas process control using DDC," presented at the 1967 IFAC/IFIP Internatl. Conf., Menton, France.
[22] "Computer operation in Owens-Illinois glass container plant," *Natl. Glass Budget*, September 23, 1967.
[23] "First computerized glass container plant dedicated at Dayville by Knox," *Natl. Glass Budget*, October 15, 1966.
[24] "Ford's computer controlled plant to operate nonstop for three years," *Natl. Glass Budget*, July 22, 1967.
[25] "Ford's flat glass plant takes capacity, quality," *Automotive News*, May 1, 1967.
[26] R. M. Gagne, Ed., *Psychological Principles in System Development*. New York: Holt, Rinehart, and Winston, 1962.
[27] J. C. Kennedy, "Psychology and system development," in *Psychological Principles in System Development*, R. M. Gagne, Ed. New York: Holt, Rinehart, and Winston, 1962.
[28] I. L. Auerback, "The information revolution and its impact on automatic control," presented at the 1967 IFAC Cong., Basel, Switzerland.
[29] K. Davis, *Human Relations at Work: The Dynamics of Organizational Behavior*, 3rd ed. New York: McGraw-Hill, 1967.

The place of digital backup in the direct digital control system

by J. M. LOMBARDO
The Foxboro Company
Foxboro, Massachusetts

INTRODUCTION

The key to the success of direct digital control on large industrial processes lies in its flexibility in implementing everyday process control problems as well as advanced control at lower overall system cost. Control concepts for continuous processes use the computing, monitoring, information storage and analytical ability of the direct digital control computer. In the batch or discontinuous process the computer's logic capability is emphasized. To perform batching operations, a comprehensive logic system is necessary. Implementation of such a system using digital techniques provides many advantages over implementation using analog equipment with auxiliary digital logic circuits.

To fully appreciate these advantages, the reader must have a basic understanding of continuous control systems as well as the batch type systems. The following will describe single loop control, several advanced control concepts and control of semicontinuous processes, as an introduction to digital computer application and backup.

Single loop control

Simple single loop feedback control is the most common control found in the process industries. It is used for controlling flow, level, temperature, pressure and many other variables. Both pneumatic and electronic devices are available which provide this type of control.

Basically, these controllers compare the measurement of a variable with its desired value or set point. If the two values are not equal, the controller adjusts a control value to minimize the difference (Figure 1).

In action, the controller is an analog computer which calculates a one, two or three term expression,

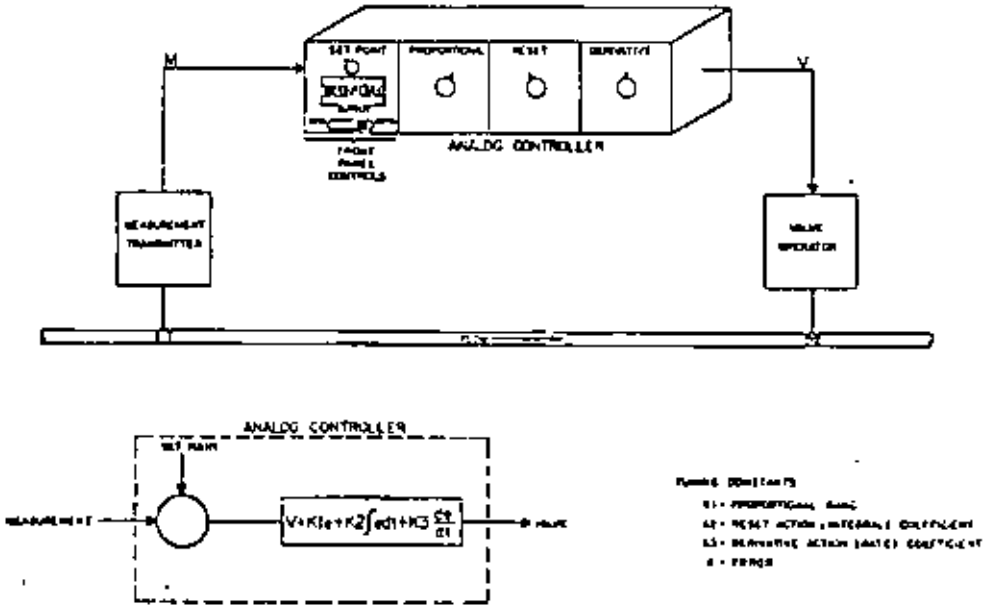


Figure 1 – Typical single variable feedback control loop

Reprinted with permission from *AFIPS Conf. Proc., Vol. 30, 1967 Spring Joint Comput. Conf.*, pp. 771-778.

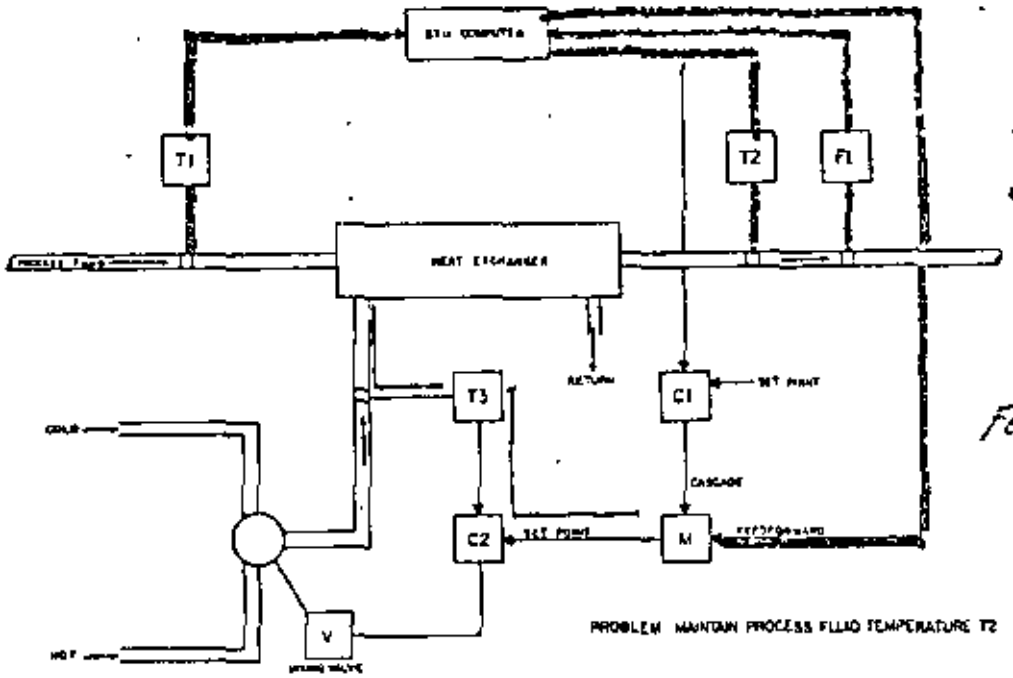


Figure 2—Advance control techniques applied to a heat exchanger

depending on the type of control action required by the process. The three terms define proportional, reset and derivative control action. During process start-up, coefficients of the three terms are manually set on the controller to provide the best response under normal operating conditions. If operating conditions change, as the process operator changes the set-point radically, the coefficients are no longer at optimum values.

Advanced control concepts

As the control problem becomes more complicated, single-loop feedback control is no longer sufficient. Figure 2 illustrates three types of advanced control: inferential, feedforward and cascade.

In the inferential control, a relationship is calculated between two or more measurements which is used to control the desired but unmeasurable variable. In Figure 2, the Btu computer performs a calculation based on the difference between the outlet and inlet temperatures to the heat exchanger (T2—T1) and the flow rate of process fluid through the heat exchanger. This calculation is a measure of the heat transferred to the process fluid and determines the demand of hot or cold fluid needed to maintain process fluid output temperature T2.

Analog computing devices perform the necessary calculations and control can be executed with conventional analog control devices. Additional calculations may be necessary before some variables are combined. For example, the differential pressure signal provided by the commonly used orifice plate is proportional to the square of the flow. A computing element is therefore necessary to extract the square root of the differential pressure signal.

Figure 2 also illustrates feedforward control. The calculation of heat transfer (Btu) rate is "fed forward" to adjust the flow of heating or cooling fluid and change temperature T3. This feedforward calculation anticipates disturbances in both inlet temperature T1 and process flow F1. To provide more stable control of T2, the feedforward signal anticipates the change in heat input required. The magnitude of the feedforward action is usually determined by experimentation and may have to be adjusted periodically, since the heat transfer characteristics of the heat exchanger change with age.

A third control technique illustrated by Figure 2 is cascade control, a technique where one controller adjusts the set point of another controller. The output of temperature controller C1 is fed (cascaded) to the set point of temperature controller C2 through a multiplying device M. Hence a change in process fluid output temperature T2 affects the set point of controller C2 to ultimately maintain output temperature.

The control loops discussed have been applied to continuous processes which operate at near steady conditions with only nominal process or set point disturbances. Therefore adjustment of the proportional, reset and derivative coefficients is rarely necessary unless process changes are nominal. In batch

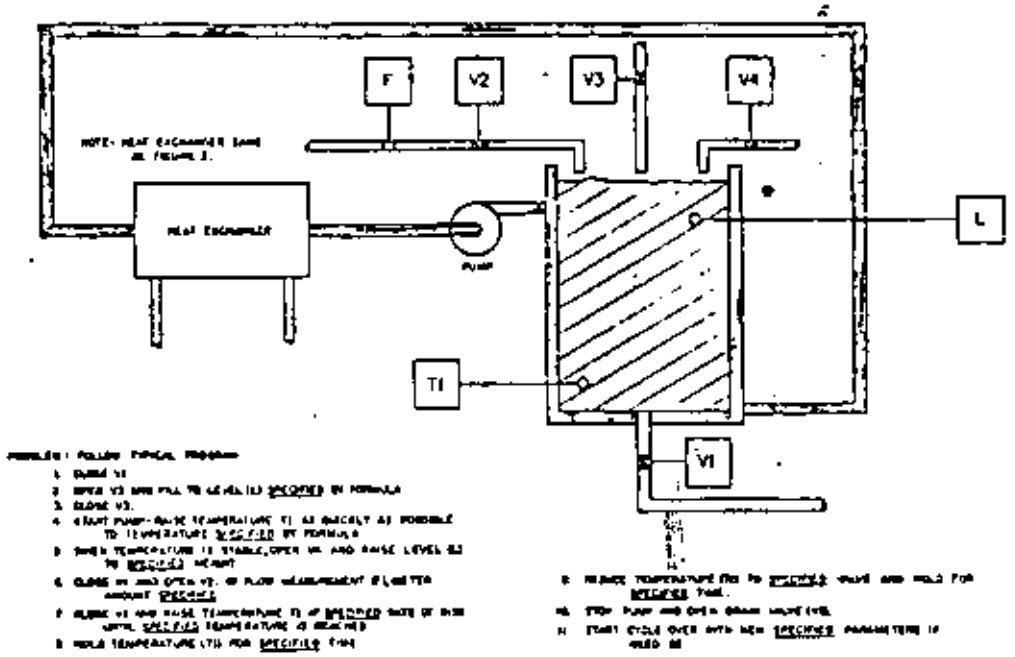


Figure 3—Simple batch control sequence

tinuous well-behaved process, use of these adjustments would be very limited. Many high production petrochemical processes are in the continuous process category.

Control of semicontinuous processes

Figure 3 presents a process control problem where steady operating conditions are not maintained. This type of process requires a control system which changes operating conditions according to a preplanned event/time schedule. Batch or semicontinuous processes require controlled sequencing because: various equipment must be started and stopped frequently, product requirements change frequently and operating parameters change. It should be noted that most batch or continuous processes still use feedback control, but with programmed changes of control set point.

Figure 3 illustrates a simple chemical reactor. Ingredients are added sequentially and temperature is maintained according to various preset programs to provide the chemical reactions necessary for various products. The reaction within the vessel can vary from endothermic to exothermic during the production cycle. Hence in order to hold a set temperature the control system may be required to switch from heating the reactor to cooling it when the reaction starts to generate its own heat.

In the typical chemical reactor or mixing vessel, different control sequences may be necessary for each new product. For instance, there may be changes in specified ingredient mix and heating and cooling temperatures and temperature rates of change. Process control problems of this nature require more complex control than the feedback, feedforward and multivariable controls previously described. This control requires programmed sequencing of events, including equipment starting and stopping.

In Figure 3, the control of reaction temperature TI is basically a feedback control problem. However, the problem is complicated, since TI must change at the proper times, sometimes in step-wise fashion and other times at a controlled rate. Also, the sequence of events must be readily changed, depending on the intended product.

Combinations of special purpose digital and analog control equipment have been built which satisfy the demands of the discontinuous process. However, the programming of this equipment is relatively inflexible and the control cannot be well-tuned because of the cyclic nature of batch processes. Many of these systems are not used at full operating speed, since the control constants are a compromise.

Applying the digital computer

Digital computers are of significant interest to the industrial process control field due to their ability to store programs which calculate simple and complex control relationships, compute variables which are not directly measurable, monitor the process and take action according to a preplanned schedule. The digital computer easily performs tasks that the analog system finds difficult or cannot be easily programmed to adapt the overall control system to changes in process-

dynamics, materials, equipment and production demands. Because of this versatility digital computers are being designed and installed in continuous process plants as well as in batch process plants. Many of the installations use direct digital control techniques on all or some of the control problems.

Table I compares two systems each using direct digital control exclusively. As shown, a continuous process application in an oil refinery has 530 analog measurements of which 275 are associated with control calculations, the other inputs are for performance monitoring and system operation analysis. Of the 275 control inputs, 180 are used for direct control of simple loops; the remaining 95 are used in advanced control. Therefore, approximately one-third of the 275 inputs associated with control are used to implement multivariable and advanced control techniques.

Table I – Comparison of computer system input/output between continuous and batch process control

	CONTINUOUS PROCESS	BATCH PROCESS
TOTAL ANALOG INPUTS	530	430
ANALOG INPUTS IN CONTROL LOOPS	275	210
CONTROL LOOPS		
SINGLE	180	111
CASCADE	80	70
FEED FORWARD	15	9
DIGITAL INPUT CONTACTS	270	1015
DIGITAL OUTPUTS (ON-OFF)	355	1200

Table I also shows the input/output distribution for a large batch control installation currently being implemented by a digital computer system. A comparison of the batch with the continuous process reveals a significant increase in contact sensing elements and on-off control outputs. In order to sequence events the batch system must sense the status of process equipment and conditions. Also, many devices must be turned on and off. With the batch system, man-machine communication needs also increase. Increased number of push buttons, signal lights and the increased size of digital displays require more digital inputs and outputs.

It is also significant that the number of control outputs (295) can exceed the number of analog inputs (240) in the batch system. This situation occurs in batch processes because the same measurement can be used in control of different control elements and with different control algorithms, depending on the sequence of events and the starting and stopping of equipment.

The philosophy of DDC

With the introduction of the digital computer to the process control field, it became evident that relatively little was known about most processes. Most processes could not be adequately represented by mathematical models which would permit improved process control.

Early attempts at applying the digital computer emphasized supervisory control in which the computer adjusted the set point of an analog controller. In these systems the analog controller retained the last computer control setting if the computer failed. On continuous processes, this control was quite satisfactory; in fact, once the system was operating satisfactorily, it made little difference whether the computer was there or not. The operator could still adjust control actions, as he did before the installation of the supervisory computer. This made the process operators happy, but in many instances the process engineers and plant supervisors were not. There was no guarantee that the operators would achieve the optimum control settings for the plant.

What additional advantages did the computer provide? If so desired, the computer could make feedforward, cascade and inferential calculations which would optimize control set points for economic or production considerations. Economic constraints relating to material balance, throughput, inventory, etc. could be developed. In a sense, an economic mathematical model was possible, whereas a process model was still difficult to achieve, due to lack of process knowledge. In addition, the on-line process computer performed other useful work to aid operators, plant supervisors and process engineers: see Table II.

Table II – Some non-critical functions of an on-line process computer

- LOG OPERATING DATA IN ENGINEERING UNITS
- CALCULATE AND DISPLAY OPERATOR GUIDES
- INTEGRATION OF MATERIAL FLOW
- REPORT ON PROCESS STATISTICS - MATERIAL USED FUEL USAGE, THROUGHPUT, ETC.
- CALCULATE AND DISPLAY OR RECORD UNMEASUREABLE VARIABLES SUCH AS BTU RATE, MASS FLOW
- MONITOR AND ALARM PROCESS LIMITS
- RECORD PROCESS EVENTS DURING UNUSUAL DISTURBANCES
- MONITOR AND RECORD CHANGES IN SET POINTS, ALARM LIMITS, ETC. MADE BY THE OPERATOR
- PROVIDE ON DEMAND OPERATOR INFORMATION SUCH AS TREND RECORDING, ALARM STATUS REPORT, LOOP SET POINT AND PARAMETER DATA

Direct digital control was under consideration at the same time that the general purpose digital computer was performing process analysis, monitoring and supervisory control.[1] It was reasoned that DDC would reduce the cost of a process control computer by eliminating the cost of the individual feedback controllers. Since the controller merely performs a calculation, why couldn't the computer perform the calculation? Several experimental ventures showed that the DDC concept was physically possible.[2,3] The feedback control law was calculated within a general purpose computer and the resulting signal outputted directly to the control valve.

At first, it appeared that the trade-off between individual loop controllers and a direct digital control (DDC) computer was in the area of 200 loops. There was a hooker, however. This trade-off did not include any provisions in case the computer system failed. For most installations this meant using analog controllers to back up the DDC computer on each loop considered critical.

The DDC equipment was designed so that, if the computer failed, each valve would remain in its last directed position unless backed up by analog control. Critical loops were backed by an analog controller which would maintain loop control on computer failure. Control valves of the other DDC loops were "locked in" at their last output, but the operator could manually position each valve from a console on which he could read valve position and process measurement.

Figure 4 shows two loops from a large system. The measurement M_p feeds the manual control panel, enables the operator to manually operate valve V_1 in case of computer failure. The loop containing measurement M_p and valve V_1 has an analog controller for back up since measurement M_p is fast acting and cannot be controlled manually by the operator.

With the evolutionary history of digital process computer equipment, it is impossible to more than estimate mean-time-between-failures (MTBF). For the smaller digital computers, including input/output equipment, that have been applied to the process control problems, calculated MTBF has ranged from 1000 to 2000 hours. Advances in circuit design indicate that reliability will increase, but reliability statistics on integrated circuits are not yet available. However, regardless of the projections and the calculated claims, the time-shared single computer system will never be perfect and will sometime fail. Therefore, control security must always be considered on any process installation contemplating a digital computer.

For continuous processes, involving less than 150 loops, it appears that the single computer with set point, analog control or DDC with analog backup and some pure DDC on noncritical loops makes the most sense. However, the user must be fully aware that he will give up economic and process control optimization, as well as the functions listed in Table II, if the computer fails. Perhaps most important,

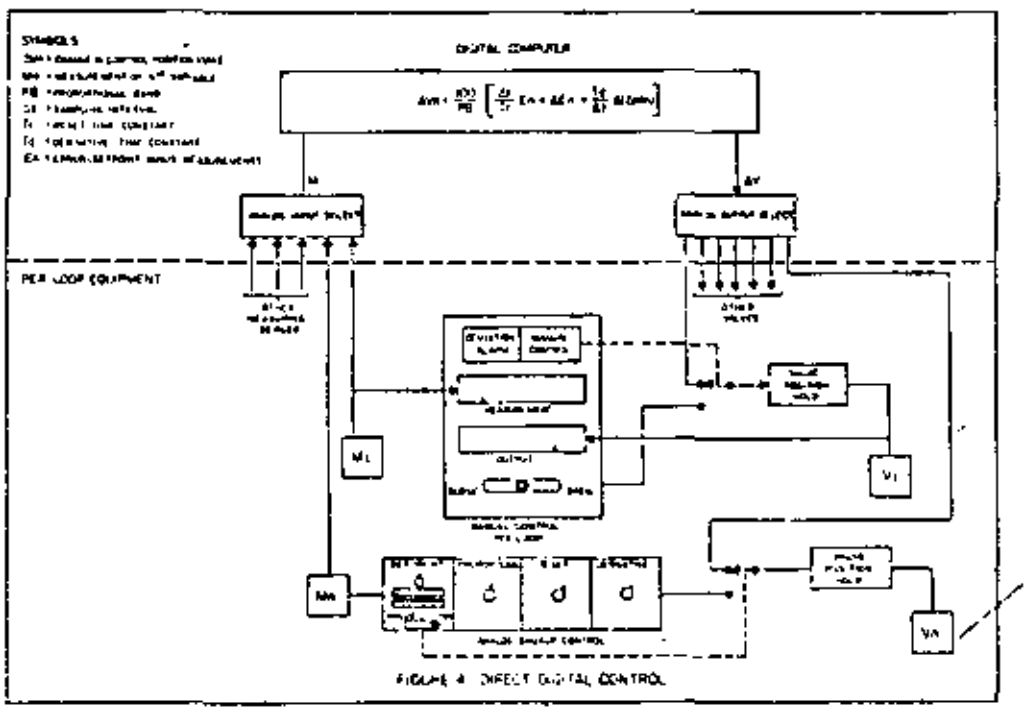


Figure 4 — Direct digital control

any advanced control that was dependent upon the computer, such as feedforward, decoupling and multivariable, will be lost during computer shutdown.

For the installation where control is not continuous, but where control sequencing is imperative, the use of computer set analog controllers is not sufficient. The computer provides sequencing and logic analysis which must have backup, if process operation is to be assured. The process control problem is not solved by keeping all control settings stationary upon computer system failure. In a chemical reaction for instance, the contents can solidify or the reaction can "run away," if the process set point is not changed at the proper time.

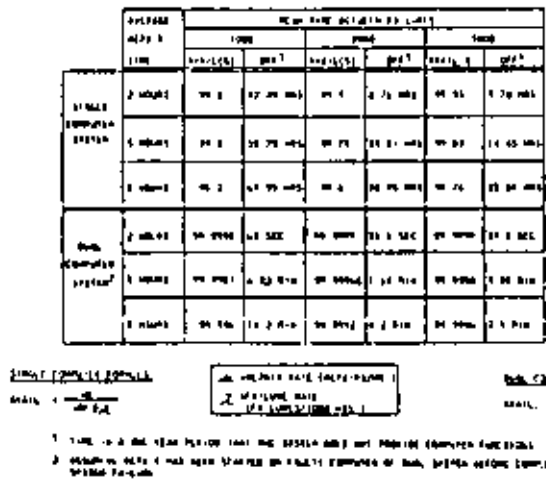
A parallel DDC computer system

Figure 5 illustrates a parallel DDC computer system which not only provides computer backup but "backs up" the time-shared analog and digital input/output equipment which connects the computer to the various measurement and control elements. It also backs up all interloop controls, as well as all sequence control action.

In addition, this system can continue to perform the noncontrol functions such as those listed in Table II. It therefore permits control to continue even if one computer and/or its time-shared I/O equipment should fail. Note that if any of the time-shared equipment fails, process control is transferred to the backup subsystem.

Table III shows some interesting statistical data[4] which compare the availability of a single computer system with a parallel computer system. The table assumes that the MTBF of a single computer system is the same for each computer subsystem of the parallel computer system. Experience has shown that repair time for various failures, with on-site maintenance personnel, averages between 5 and 8 hours, depending upon the skill of the maintenance personnel, the availability of spare equipment, etc. With the parallel system, it appears that the average repair time can be maintained under 5 hours, since the system incorporates elaborate programs for self-diagnosis to ensure proper transfer to the backup

Table III—Availability—single computer vs. dual computer system



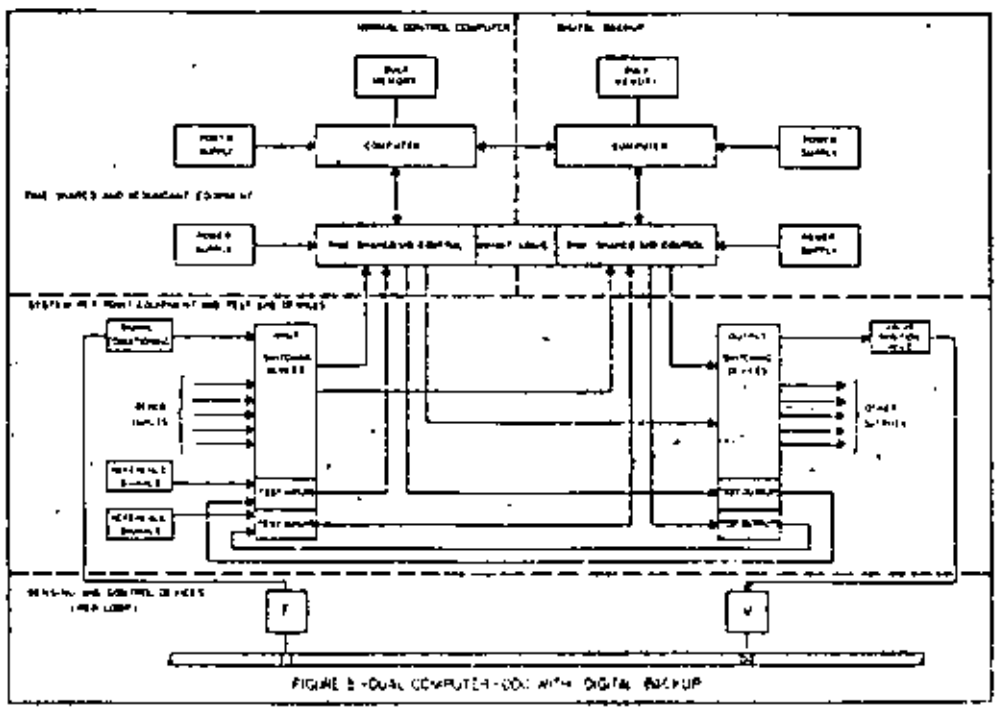


Figure 5—Dual computer—DDC with digital backup

system. The failed computer subsystem is available for self-checking while the backup subsystem maintains process control.

Systems of this type can be economically attractive since they provide not only the essential control, but the system security essential to batch or start-stop operations. A parallel control processor using direct digital control techniques takes full advantage of the digital computer's process control capability without reservation and compromise. It can include advanced control techniques, such as self-tuning or adaptive control which cannot be obtained with set point control. The parallel computer processing system may provide these features and, in addition, may offer cost advantages over a conventional analog control system for the large continuous process.

For the continuous process in Table I, the computer contains the equivalent of 272 analog controllers. Implementation of a system of this size with DDC and analog backup could exceed the cost of implementation with the parallel or redundant computer scheme.

Input/output equipment

Figures 4 and 5 show that in DDC, as in all control systems, measuring elements and final control devices are still essential. Each measurement is individually conditioned before being fed to the multiplexer of the computer input/output system. Failure of any input or output therefore is similar to failure of a single controller and will not disable other loops. The system should be designed so that failure of any circuit element will not cause the loss of any common power supplies. Also, in case of a power failure, there must be battery backup or a redundant power supply.

Other cautions must be observed in the design of the parallel system interface equipment:

The system must be able to identify and diagnose the fault of any time-shared input/output equipment without disrupting control. The normal control computer and the backup system should both contain several inputs and outputs which can be used for automatic on-line testing of I/O operation regardless of which subsystem is controlling the process. Some of these test inputs are connected to reference signals, others are connected to output test signals, closing the test loops through each subsystem.

All failed devices must be easily removed for replacement. Any disruption of normal functions during repair should be limited to the few inputs or outputs which share the same printed circuit as the failed element.

There also should be a diagnostic program which verifies correct operation after the failed component has been replaced.

Output devices, for valve positioning or on-off control, which require power to maintain their status and/or output signal, should have at least a 30-minute battery backup system, in case of system AC power loss.

The system must detect the failure of any time-shared element in the input/output system control logic and automatically switch to digital backup. While operation is in the backup mode, the failed control logic must be electrically isolated and inhibited from operating input and output control devices. Repair can then proceed with no fear of accidental interference with process control.

In normal operation, with the control computer in command, the backup system must continually check its input/output operations to ensure that backup is available.

The inhibit logic must be fail-safe so that its failure will not disturb the system in control. It must be tested automatically to ensure that transfer to backup can take place if a transfer is commanded by a failure detection. If inhibit logic will not transfer the other computer automatically, the system should annunciate that fact and provide an independent manual override which forces transfer of the control of the input/output equipment to the other computer.

Other system design requirements

The system must have a computer-to-computer communication link which continually updates the backup program data and status on a periodic fixed time basis. The backup computer thus receives dynamic operating conditions within a short time period (in the order of seconds for a batch process).

Any program changes made on-line while the control computer is operating the process must be transferred to the backup control computer at the same time. This updating must include operator changes to control settings as well as any on-line program changes.

A bulk memory must be used on both computer systems to retain the many formulas and programs that may be required. Bulk memory can also contain interpretive programs to simplify construction of a batch program, diagnostic programs for fault detection and programs to aid maintenance. Sophisticated man-machine communication programs, which involve lengthy message storage, can also be included.

Diagnostic programs for the computer-to-computer communications link should test for link failure, annunciate the failure and command the changeover to the backup system. A program system permits updating and on-line diagnostics while time-sharing the real-time programs in bulk memory.

There should be a system procedure and a system diagnostic program consist in rapid repair of failed subsystem. Another procedure and program is required to transfer all operating programs from the backup subsystem back to the repaired computer, without interfering with process control.

When the backup system is not on control, it is available for program compiling, debugging and problem simulation using the test inputs and outputs. It must also perform diagnostics to ensure operation is correct for takeover if necessary. When backup computer takes over process control, these programs are discontinued.

CONCLUSION

By using DDC with complete input/output control and computer backup, the parallel computer processing system permits unrestricted application of computer control techniques. It takes full advantage of the logic and computational ability of the digital computer, whereas a computer system which depends on analog set point control or analog backup cannot.

The parallel control computer system program storage ability, together with backup of logic control, program sequence and formulation, makes it ideally suited for complex batch or start-up and shutdown applications.

Complex continuous control systems would also benefit with this control system. Built with state-of-the-art electronics, the system should challenge the economics of computer set point control and single computer direct digital control with analog backup.

REFERENCES

1 J W BERNARD J F CASHEN
Direct digital control
Instruments and Control Systems Sept 1965
2 E VANDER SHRAFF W I STRAUSS
Direct digital control - an emerging technology
Oil and Gas Journal November 16 1964
3 J W BERNARD J S WUJKOWSKI
DDC experience in a chemical process
ISA Journal December 1965
4 R H MYERS K L WONG H M GORDY
Reliability engineering for electronic systems
John Wiley and Sons New York 1964

Recent Developments in Automation of Cement Plants

E. H. GAUTHIER, JR., MYRON R. HURLBUT, MEMBER, IEEE, AND EDWARD A. E. RICH, SENIOR MEMBER, IEEE

Abstract—Over a decade has passed since process computers were first applied to control parts of the cement manufacturing process. The path from there to the present has successes—and failures—along its way. Over 30 cement plants have installed such computers as part of their efforts to keep profit margins from shrinking. Progress in using these process control techniques has been largely evolutionary. Certain factors can now be identified more certainly as essential ingredients for success. Among these are the following. 1) "People factors" of the cement manufacturer stand as first in importance. These include management support, process know-how, training and supervision of operators, and an inner confidence and determination that "we can make it work." 2) Well-done interface jobs of adapting control room design and operators to each other, automation components with the process and its machinery, and plant design to fit automatic control fundamentals. 3) Designing the process to really be controllable. 4) Control hardware and software which fit the nature of this industry. Each of the foregoing factors is expanded with emphasis on how recent developments of better understanding, control functions, hardware, software, and of process and plant design are merging to help shape the future of automation in the cement industry.

Paper 71 TP 9-IGA, approved by the Cement Industry Committee of the IEEE IGA Group for presentation at the 1971 Thirteenth Annual IEEE Cement Industry Technical Conference, Seattle, Wash., May 11–13. Manuscript received June 10, 1971.

E. H. Gauthier and M. R. Hurlbut are with the Manufacturing and Process Automation Business Division, General Electric Company, West Lynn, Mass. 01910.

E. A. E. Rich is with the Industry Sales and Engineering Operation, General Electric Company, Schenectady, N. Y. 12345.

Introduction

OVER 10 years have elapsed since digital process control computers were first introduced in the cement manufacturing industry (cf references). Over 30 have been installed or soon will be. Some have achieved success. Some have not. Some are moderately successful.

During this period several distinct trends have emerged. Among these is the realization that the essential ingredients for successful process control systems, as shown in Table I, also apply to the cement industry. These ingredients were derived from a study of diverse process industries which had used process control computer systems. A further trend is increasing evidence that economic benefits of the more successful systems in cement plants tend to be at least equal to those shown in Table II.

Some additional trends are the following.

1) Increased awareness—and adjusting to the implications—of the essential importance of adequate "people factors" to operate and support such systems.

2) The spreading use of direct digital control (DDC) as part of the automation system as contrasted to computers using supervisory methods of loop control only.

3) The development of adequate interface concepts and hardware to adapt the automation system to the process and to the people using the control system.

TABLE I
ESSENTIAL INGREDIENTS FOR SUCCESSFUL PROCESS CONTROL SYSTEMS USING DIGITAL COMPUTERS

Ingredient	Accountability	
	Owner	Supplier
Management Support	X	
Process Know-How	X	C
Control Strategy	C	X
Equipment Specifications	X	C
Interface Engineering	X (Part)	X (Part)
Standard Software		X
Programming	C	X
Installation	C	X
Maintenance	X	C
Training	X	C
Process Hardware, Operation	X	
Control Hardware	X (Part)	X (Part)

X = responsible, C = consult.

TABLE II
OPERATING BENEFITS

Item	Typical Percent Values
Reduced fuel consumption per unit weight of product produced	3-12
Increased annual production with same basic process facilities	7-15
Reduced maintenance of kiln linings, cooler grates, etc.	10-40
Reduced kiln chains wear.	10-40
Grinding power savings, plus less wear on liners and grinding media.	2-4
Extending quarry reserves, minimizing costs of purchased additives, better waste dust utilization.	*
More even uniformity, as well as long term average uniformity, of quality.	*
Continuously produced production data, and production totals (for measuring trends of unit performance, management reports, etc.)	*
Labor	None - to increase, Some - 1 to 2 men, one shift per 5 day week.
Savings only where plant design is modified to better adapt to automation:	
Homogenizing (blending) facilities.	Less
Central Control Room, Central Operator's Panel and Associated Equipment	Less

* Figures not available.

4) The increasing awareness by many engaged in plant design that automatic process control principles provide a basis for making significant improvements in many aspects of plant design. In some cases the total plant investment required is favorably affected.

5) The large increase of total automatic control functions being performed where major consideration of such automatic control is taken in design and operation of new cement plants. This has been especially so for some plants designed by Europeans.

6) The increase of availability of automatic process control systems when the major components of the system are supplied with needed power by an isolated "clean" ride-through power supply.

7) The recent development of small or minicomputer with adequate supporting interface hardware and software makes possible economic automatic control of smaller segments of the process than was practical hitherto. In effect, a process segment becomes controlled by its own "dedicated" computer. The small computer provides the possibility of economically adding automatic control to selected parts of many existing as well as new plants. This is especially true where plant design and operating realities favor a stretched out step-by-step approach with a minimum of interaction between each new step and those already taken.

People Factors

"People factors" are the major key in achieving successful profitable automatic process control. Even some slower first-generation process control computers are still earning their way when adequate people factors have been created and maintained in place over the years. The faster and more powerful third- and fourth-generation computers do not bring success where adequate supporting people factors are not designed and maintained in place. What are some of these people factors which can be considered vital to success?

A Favorable Environment for Central Control Operators: This favorable environment which is created mostly by plant management includes the following. 1) There should be no shame on, or threat to the security of, the operators if the automatic system controls the process better overall than the operators do. 2) There should be a desire on the part of each control operator and their supervisors in that they *want* the control system to succeed, they *believe* they can help make it succeed, and they *take* the necessary steps to make it succeed. Finally, this results in the realization that making the system work well is really a contribution to his company's profitability, hence to better job security. 3) There should be written and readily available operating rules. To be effective these must be simple, closely fit the local situation, and then be enforced fairly. Yet means must be retained for accepting and placing in effect valid suggestions for improvements coming from operating personnel.

Training Supplemented by Regular Refresher Courses: Ignorance and misconceptions about automation are a major source of apprehension about automation on the part of operators and their supervisors. This ignorance is often well disguised. Well-designed training and refresher courses, especially tailored to the needs and capabilities of these personnel, provide a tactful yet effective way to dispel sufficient ignorance about automation so that good progress is achieved.

The best training courses for operators generally result when prepared and administered by those having responsible charge and administration of control of the cement making process. Short courses of training in control sys-

tem concepts and applications are also desirable for higher levels of cement plant management and for those supervisory personnel who, while not responsible for process control, significantly affect its results by the quality of support and understanding they give in discharging their duties. Training and practice in maintenance of automation components is also vital. Usually the highest availability on control has been found to exist where the owner does most of his own maintenance work on automation system components. Training in programming for 2 or 3 cement plant personnel is very useful. The nature of the cement making process is such that conditions often change. These changes may arise from wear, chemical or physical properties, and other sources. A reasonable proficiency in modifying the control programs to closely accommodate these changes, when they affect process control, does much to maintain good control efficiency. Equally important, a high confidence level in the process control system itself is thereby maintained.

Adjusting Job Descriptions of Plant Supervisory and Process Control Personnel: The goal of this is to make the descriptions more closely fit the realities of automatic process control. For example, the four major continuous parts of most cement plants (raw grinding, homogenizing or blending, burning, and finish grinding) highly interact with each other, especially in the downstream direction. (Plants which use hot kiln gas for drying have even more complex interacting control problems.) How often does one see ball mill operators taking actions which influence chemistry, or vice versa, but without coordinating with the chemists to assure overall minimum perturbations to the process? Conceptually, the best arrangements, taking automatic control of the process into account, follow.

1) A single manager of process control who at least manages operating control of the grinding, blending, and burning operations. He is responsible for training and discipline of the central operators. He is also accountable for operation of the continuous parts of the process.

2) The chemical and other operating personnel act more as advisors to this Process Control Manager but with no direct authority over central operating personnel.

3) This single Process Control Manager usually will make minor program adjustments necessary to keep abreast of process changes and to make desired improvements. This can be delegated in whole or in part to others, but it is his responsibility to judge, install, verify, and finally determine the usefulness of such changes.

Some plants have modernized their supervisory and operating structures to achieve successful automatic process control with no overall increase of personnel. Some other plants have retained traditional job descriptions. The highest plant supervision has tended to stay busily, sincerely, and relatively aloof from addressing themselves to adjusting to the implications of automation. Frustratingly enough, most of the failures and moderate successes are found in this class.

People Factors of Automation Suppliers: Supplier's people are a key ingredient in assisting a user of automation to achieve successful control of the process, especially where application software for process control is purchased. Their know-how, combined with know-how of the owner's representatives, largely provides the basis for the later success or failure of the new control system. The trend is to better organize the planning, training, installation, and operation of automation systems to make success more certain.

Direct Digital Control

DDC time shares the digital computer to directly control the final element, such as a valve, damper, etc. Usually some form of backup hardware—computer manual station, as an example—exists on the central operator's panel for each final output device being controlled. This backup device also provides a means of manual adjustment of the final output device when the computer is out of service. It may even be in the form of a full analog controller.

Many of the earlier cement automation systems utilize conventional analog instrument controllers to manipulate those process variables which are within the capabilities of such analog controllers. Supervisory logic, often called "Level 2," calculates the output signals to cause manipulation of the set points of such analog controllers. Supervisory logic is used to handle those control situations where combinations of interactions with other control loops, nonlinearities, very long process delays, and highly involved calculations make usual analog controllers relatively useless. This system is also known as digital analog control (DAC); or digitally directed analog control (DDAC). As hardware, and especially as good supporting software, became available in the last half of the 1960's, DDC spread so that now it is first choice in many new installations. Among the advantages DDC provides, as compared with more conventional analog instrumentation, are as follows.

1) The DDC computer readily checks limits, provides digital filtering over long periods of time, makes mathematical calculations, and does decision making—many of which are difficult or impractical with analog instrumentation equipment.

2) In many instances more precise control results because the drift problem within the regulator itself is absent.

3) The use of DDC forces operators to be systematic in documenting all constants associated with each regulating loop. This is rarely done with analog regulating systems, although such systems would work better if such documentation was done and kept up to date, and used to maintain best adjustments.

4) DDC is comparatively easily arranged for bumpless transfer for different modes of operation, prevention of reset windup, and automatic failure detection.

5) With DDC addition and deletion of loops and changes in the control equations to be used are easily done. This

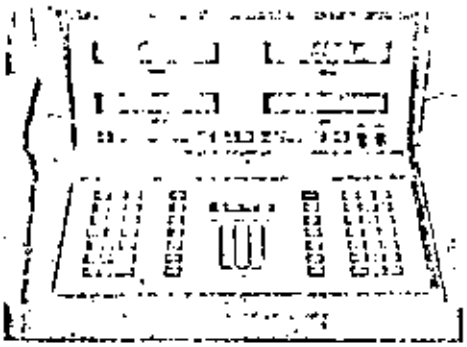


Fig. 1. DDC operator's console.

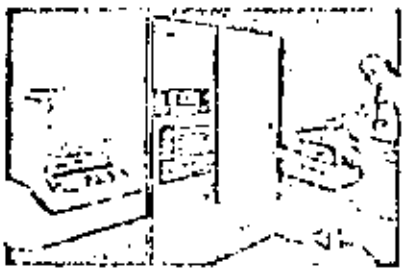


Fig. 2. Minicomputer with CRT operator's console for cement process control.

is especially useful during automatic start-up and shutdown of major parts of the total process during which transient manipulation of control loops is often required.

6) In a DDC computer the overall higher level of process control (called Level 2 or supervisory control) tends to be more easily done since the computer must only communicate with itself to change set points, switch loops in and out, modify control equations, and so forth.

7) In analyzing several existing installations it is evident that a well-done carefully thought-through and well-operated digital computer system using analog regulators on loops for which they are suitable can provide essentially as good a control of the process as can DDC for many parts of the continuous process. However, this is not true during automatic start-up. As the complexity of the regulating loops increases (such as some cases where complex gas flow patterns exist between raw grinding and the kiln-cooler department), DDC provides significant advantages by readily permitting easy switching of regulating loops and modifying their forms to follow the variable gas flows which such plants have.

Fig. 1 shows one form of a DDC operator's console used for the man computer interface in a DDC system. Fig. 2 shows a cathode-ray tube (CRT) input output console more recently available for cement plant control. The use of CRT seems likely to spread.

Interface Concepts and Hardware

Trials and experiences clearly show that essential ingredients for successful cement automation also include adequately interfacing 1) the central control room design and the central operators to each other; 2) the automation components with the process and its machinery; 3) the plant design with the automation system; 4) the plant power distribution system layout with the automation system; 5) many drives and their control with the logic in the automation system where automatic start-up-shutdown is included in the automation system for selected parts of the plant processes.

Other factors exist. The foregoing are the most important. Discussion of each follows.

Interfacing Central Control Room with Central Operators

Simplification of the layouts of the central room and of the central operator's panel (COP) is worthwhile. Such simplification tends to lower initial costs for central control room equipment and wiring. Operating, troubleshooting, and maintenance procedures are greatly simplified if good concepts are used in such layouts. Among the factors which permit good simplification without sacrificing operability or reliability are as follows.

1) It would be wise to simplify the control room operator's job. A typical central operator is hard-pressed to effectively monitor and properly respond to more than a few hundred displayed items of information. Yet one sometimes finds a COP in a cement plant having 1000–2000 different indicating lights, 50–200 ammeters and indicators, 20–40 recorders, 200–600 push-button stations and selector switches, 30–60 controllers and set point stations, etc. Why so many?

2) In the design stage rigidly question whether it is necessary for each device to be in the central room. If it is an ammeter primarily intended for maintenance uses, it probably belongs on the motor control center for the motor in question. If it is an indicating light showing status of an individual drive, what can the operator do about that light? Often such status lights are better on their departmental motor control center or relay panel. Maintenance may be their primary purpose. If so, it is better done by having such lights at the motor control center or relay panel. If it is a recorder, would not the purpose of the central operator be better served if he were limited to charts of the critical variables only? Other analog variables can be recorded by switching to one or more shared recorders when special tests and observations are to be made.

3) Group starting of a complete grinding mill with its auxiliaries or of a subdepartment permits large reductions on the COP of push buttons and indicating lights. Group starting helps highlight the distinction in the design stage between those devices really needed at the COP and those devices needed for maintenance. Devices for maintenance are generally more useful—and less expensive overall—if located on their motor control center or the associated relay panel. The location of individual drive status indicating devices on the corresponding departmental motor

control center or relay panel permits quick fault finding by maintenance personnel when the central operator notifies them that a sequence of starting cannot be completed.

4) Where fully automatic computer-directed start-up and shutdown is being planned, including transient manipulations of regulating loops, arrange that the procedures for manual and computer start-ups and shutdowns be as similar as possible. This helps teach the operator correct procedures by having him observe computer start-ups. It also helps the operator sense and diagnose difficulties when something is amiss in the computer-controlled procedure.

Interfacing Automation with the Process and with Its Machinery

To control a process first requires reasonable knowledge by the process controller of process conditions. Since automatic process control digital computers are electronic devices, their process status knowledge comes from frequent monitoring of status of selected contacts and analog signals—all derived from process conditions.

Switches and their transducers help detect process limits, process flows, levels, starvation, and so forth, and provide the computer and the control operator vital status information. Some of these switches also provide part of the traditional process-flow sequence interlocking.

Process variables such as selected temperatures, flow rates, pressures, speeds, and so forth, provide process knowledge to the computer by using suitable transmitters to convert to suitable analog input signals to the computer. The clear trend for new construction is to at least make such feedback signals compatible with future process control computers.

The second major factor in remotely and automatically controlling the process is that all variables required to achieve process holdpoints must be physically available and remotely controllable. The trends discerned from applying automation show that this may mean: 1) substituting adjustable speed fan drives for damper controlled gas flow circuits in some cases; 2) assuring that all feeder drives and their feeding devices have adequate physical range of feed rates to meet actual process control needs; 3) providing sufficient number of independent raw feeders so that the chemical hold points desired can, in fact, be achieved without excessive dependence on downstream blending facilities to hopefully make up for deficiencies in this area; 4) selection of kiln, cooler grates, and other drives so that they are low drift, have preferably zero dead-band in control, have comparatively flat speed-torque curves, and can have vernier speed changes made of as low as 0.1 percent when required; and 5) arranging all such "commanded" variables with necessary components so that they are compatible with commands from the computer and its associated devices without requiring intervention by people; neither should excessive wear of the final mechanism result when subjected to large numbers of small control changes.

Interfacing Plant Design with the Automation System

The trend is to modify new plant designs (and operating procedures) and the automation system to better fit each other. For existing plants, some are so designed as not to be very compatible with automatic process control. Yet many existing plants are compatible with automation in certain parts of their process. For these, the trend toward using minicomputers tends to be a "good fit." Specific details of plant design interfacing with automation are elaborated in a following section of this paper.

Interfacing the Plant Power Distribution System Layout with Automation

A good trend based on sound engineering, but emphasized by automatic process control considerations, is to strictly departmentalize all power circuits. This means: let main power feeders serve the raw department from the raw-material feeders under raw silos to the inputs to raw homogenizing silos and nothing else. Let the cement grinding electrical power feeders serve that department and nothing else, and so on, throughout the plant. Automatic control helps highlight the importance of a well laid out power distribution system, especially when automatic start-up and shutdown are planned.

Part of the interfacing of the automatic process control systems with the power distribution system is to carefully consider in advance the effects, prevention, and cure of surges appearing in the power distribution system; of high-speed reclosing by remote utility circuit breakers; transient voltage dips and losses of whatever duration and origin; and just where power for the process control should actually be taken from the main power distribution system.

An X-ray analyzer in the laboratory that is responsive to welding somewhere else in the plant, or to spotting of a ball mill motor, tends to be relatively useless at those times. In fact, it may even give out erroneous data. A power supply for the process control taken from circuits which are subject to frequent outages or have severe switching transients, such as from cranes on them, tends to also be a poor choice. Transient overvoltages and severe short circuits in input-output wiring have each been known to "wipe out" large sections of automatic process control equipment in cement and in other plants. Good interface engineering of the power distribution system and of the automatic process control is a distinct trend and is worthwhile to do correctly. A specific solution to many of these problems is given in greater detail in a following section of of this paper.

Interfacing Drives and Their Control with Automatic Process Control When Automatic Start-Up and Shutdown are Included in the Automation System

Very few cement plants in the United States have included automatic start-ups and shutdown of selected por-

tions of the process in their process computer control system. Many are doing group starting and stopping of drives only by other means, with the operator manipulating the loops for the transient conditions during such start-up-shutdown. However, a few European-designed large throughput new cement plants have included such features. The work of doing this shows that rigorous attention must be paid to the process mechanical equipment and to its reliability when part of an overall system, as well as to the design of the automatic start-up and shutdown logic itself, if real success in executing this function is to result.

The implementing of this automatic start-up and shutdown clearly shows that first class interface engineering between machinery builders, plant designers, and automatic process control designers is best accomplished before the plant is physically built. Not only must complete possible sequences of start-up and shutdown and their variations be foreseen and accurately described ahead of time, but the performance and behavior of the various process flows and transient conditions must also be foreseen and described as accurately as possible ahead of time. Where such extra rigorous thinking is done completely during the design stage, automatic start-up and shutdown, including the transient manipulation of regulating loops, becomes more easily accomplished. When such rigorous thinking is not done ahead of time, then the actual implementing of the start-up and shutdown tends to be more protracted, unnecessarily expensive, and probably the function should not then be in the computer. An effect of applying this function already has been to contribute to modifications in process and machine design.

Effects of Automation on Plant Design

Here is where a truly exciting aspect of automation begins to be evident. Good automatic process control finally provides a means of making raw materials into finished cement relatively quickly and accurately once the raw materials are committed into the raw grinding system. Certain other industries have noted and taken advantage of the ability of automatic process control to reduce the storage between successive following parts of the process. The cement industry is beginning to use these techniques more and more [1]. Selected aspects follow.

Raw Department and Control

Designers of one relatively new United States' cement plant understood and implemented the idea of minimizing time delays between raw mill feeders and quickly obtaining and acting upon chemical information about the stream going into their downstream homogenizer. By combining on-line X-ray chemical gauging techniques of ground raw composition with short times for transport, sample analysis, and corrective actions, they were able to utilize a single 8 15 h homogenizing vessel between their raw department and their kiln. Overall, they felt that this approach saved them an investment of approximately 1 million dollars.

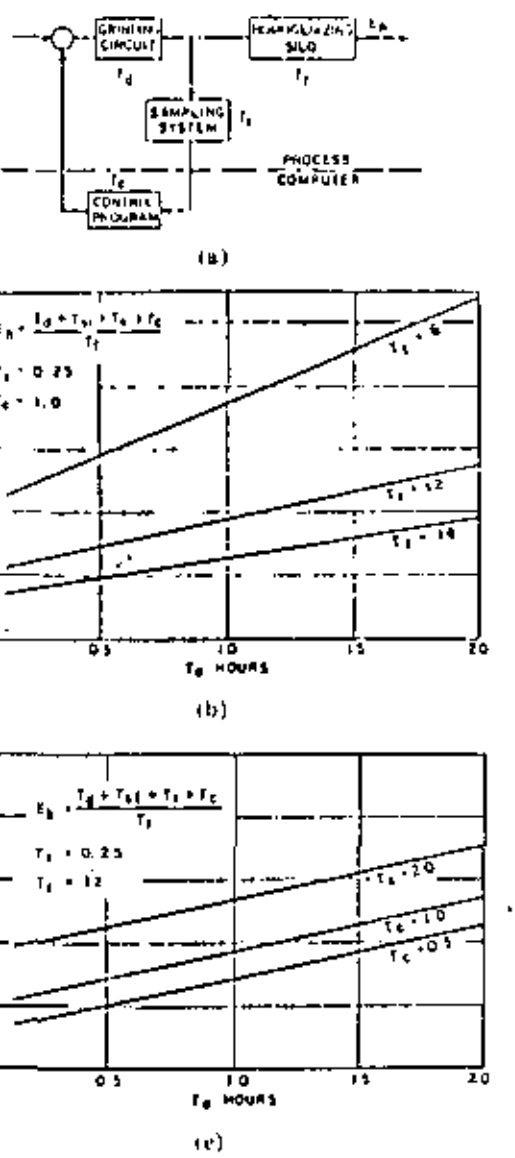


Fig. 3. Time delays and raw mix composition control accuracies. (a) Idealized raw mix control system for analyzing time delays versus homogenizer sizing. (b) Effects of time delays and homogenizer size on raw mix control. (c) Effects of control interval and system time delays on raw mix control.

Another plant in Western Europe combined good on-line gauging of raw mix chemical composition at the discharge point of the raw mill grinding circuit with short-time delays in making corrections and with computer control of their prehomogenizing pile building and computer guidance in quarry operation. By learning the techniques and performing them consistently well, they were able to completely eliminate downstream homogenizing equipment with the corresponding high operating expense of such vessels.

Another U.S. plant originally had planned to use large mill feed bins between the raw mill feeders and each raw mill. Analysis of the effects of the time delays such mill feed bins would have on decreasing possible chemical accuracies led them to eliminate such large feed bins and reduce the delays in that portion of the material transport

circuits to about 10 min instead of the 2 h planned originally.

An analytical approach to assist in understanding the effects on reduction of process delay transport, sampling, analysis, and correction times in improving the accuracies of process control, is given in Fig. 3(a)-(c).

Fig. 3(a) is a block diagram of a simplified raw mix control system that can be used to calculate approximate worst case errors in the homogenizer analysis due to system transport times. The grinding circuit, the sampling system, the sampling interval, and the control interval are treated as causing simple time delays T_d, T_s, T_{si}, and T_c respectively. The homogenizing silo has a filling time T_f corresponding to the actual "fullness" at which the silo is in fact operating. When an error occurs in the feed composition, the computer control program can do nothing until it detects the error at the output of the sampling system. It may take up to the sum of all these delays for the control program to detect the error and correct the feeders. During this time then, a total of $T_d + T_s + T_{si} + T_c$ h of bad material has gone into the system. The maximum error in the homogenizer composition will occur if the feed error occurs when the homogenizer is near full, and there is no time left to correct the error. Thus both the batch and continuous homogenizers may be considered the same, and the maximum fraction of the input error that will be present in the homogenizer output is then given by

$$E_h = \frac{T_d + T_s + T_{si} + T_c}{T_f}$$

where

- E_h Unit homogenizer error, corresponding to a unit raw material feeder chemical composition error.
- T_d Transport delays for time consumed by material traveling from raw material feeders to the sampling station, h.
- T_{si} Sample interval, h (zero for on-line gauging in the example but is more for laboratory X-ray and manual chemical analyses).
- T_s Sample preparation and analysis time, h (0.025 h used in example).
- T_c Control interval, h (typically 3 min to 1 h).

Fig. 3(b) uses this equation to show the effect of grinding circuit transport times T_d on the homogenizer error E_h for homogenizers with 6-, 12-, and 18-h filling times and assuming a control program interval of 1 h ($T_c = 1.0$) and a 15-min sampling time ($T_s = 0.25$ h). To show the implications of these curves, consider a system with a delay of 1.5 h and a filling time of 18 h. If the delay were reduced to 0.65 h, the same results could be achieved with a silo of only 12-h capacity.

The results shown in Fig. 3(b) were obtained with corrections made at 1 h intervals. Fig. 3(c) shows the effect of increasing and decreasing the interval between corrections (T_c). It can be seen that decreasing the interval to 0.5 h shows small gain with a 12-h filling time, but that increasing the interval to 2 h causes a considerable loss in accuracy.

Fig. 4. On-line X-ray chemical gauge in cement plant.

Fig. 5. Laboratory X-ray chemical analyzer in cement plant.

These results show the necessity of shortening the forward path and feedback path time delays in the raw mix system.

The possible implications of reducing investment in the "front end" of the plant by using principles shown in Fig. 3 represent a distinct and relatively new trend which will likely be used more in the future. Figs. 4 and 5 show views of X-ray chemical analyzers used in cement plants.

Improving Kiln-Cooler Design Concepts from a Control Viewpoint

Analyses of the trends show the following.

1) Increasing emphasis is being placed on keeping the arrangement of process flow and auxiliary devices in the kiln-cooler circuit as simple as possible.

2) The comments about characteristics and arrangements of kiln, cooler, grate, and selected fan drives previously given in this paper are applicable.

3) The larger grate type of coolers are more controllable if individual drives are provided to control air flow to each major compartment and for the cooler exhaust. The older practice of using very few cooler fans arranged with separate dampers to control air flow to each major compartment makes for a tough controllability problem. The trend is to provide separate fans for each function so as to permit the cooler to achieve its best performance as a heat recuperator *and* as a cooler of clinker [5].

4) When coal is used as a fuel, the trend is to try to reduce the variations in composition, especially the ash content of the coal. Such variations inevitably produce wide variations in chemical composition of the clinker where a

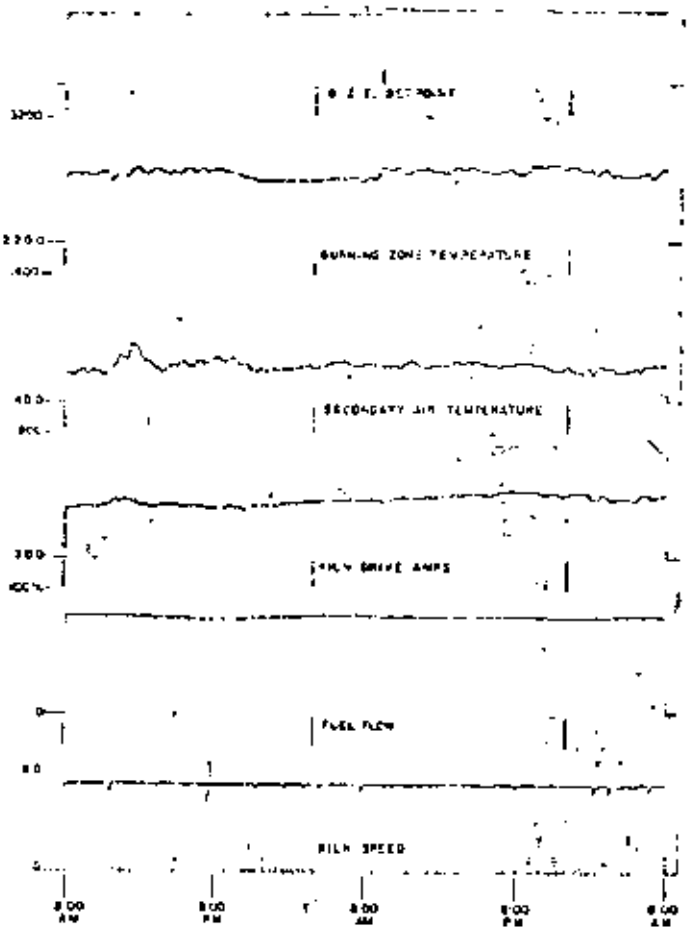


Fig. 6. Two days of computer control of cement kiln.

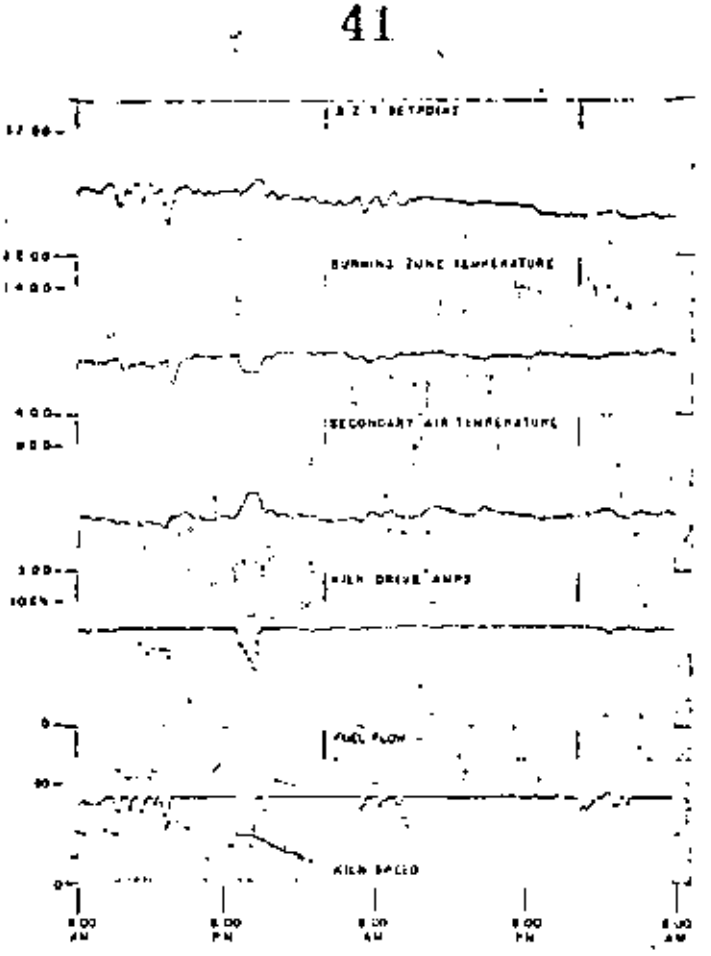


Fig. 7. Two days of manual control of same cement kiln as in Fig. 6.

major component is the widely varying ash content of the coal as it enters the burning process. Blending of such coal may help. Purchasing higher grade coal may help. Some have even shifted to other types of fuel as the problems and costs of using coal have become more evident.

5) Another trend is to return dust to the kiln in a more uniform manner to improve controllability. Avalanching of dust in hoppers under precipitators or dust collectors and starving of dust feed at other times are factors tending to require violent control actions to respond to such kiln feed perturbations. The trend is to treat return dust as another separate kiln feed and install a return dust surge bin and return dust metering equipment arranged to gradually modulate the average return dust feed rate to fit the general level of return dust being generated.

6) Sizing all process components sufficiently large so as to provide "room" for control at top production rates is another trend. It is difficult to attain top quality control if the induced draft fan, fuel feeder, kiln drive, and so forth are operating "wide open" at their top limit, i.e., out of range.

7) There is more emphasis being placed on the recuperation aspect of the cooler by obtaining good secondary air temperature measurements and then using adequate logic to emphasize heat recuperation. Some have pioneered and persevered in making good measurements of secondary air temperature.

Figs. 6 and 7 show good comparative but typical results with and without computer control on a kiln-cooler in a cement plant.

Grinding Mill Circuit Design and Automatic Process Control

The basic objective of grinding mill circuit control is to maintain fineness within a narrow band, usually at some maximum production level consistent with the existing process and machine constraints. Usually indirect measurements are required since not many fineness sensors are yet operating. Yet a trend does exist to apply and use more fineness sensors, particularly in cement grinding mill circuits [6]. Trends in grinding mill circuit design which are emphasized even further by considerations of automatic process control include the following.

1) Obtain good measurements of mill feed rates, either by weighing feeders or by a combination of total mill feed rate and selected weighing feeders for additives.

2) In closed-circuit grinding, sensing of input watts to elevators, separators, and ball mills is always preferable to attempting to obtain equivalent measurements using drive input amperes. Power system voltage affects drive input

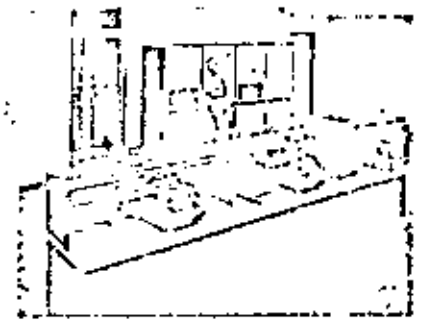


Fig. 8. Process control computer installation in cement plant.

Fig. 9. Process control computer installation in cement plant.

amperes but only slightly affects drive input watts. Thus control based on ampere readings may, at times, be based no false information.

3) The trend is to place more emphasis on adequate sizing of all components of the grinding mill circuit. This especially applies to those handling recirculating load. Good control may be impractical if some of the recirculating load auxiliaries, or of other components, prevent adequate handling of the flow rate which may be inherent due to variations in the process materials actually used.

4) Some cement plants designed by Europeans are now being built and include features for closed-loop control of fineness using continuous fineness sensors.

Figs. 8 and 9 show a process computer installation in a cement plant.

Trends in Automatic Control Functions Being Performed

The majority of United States' cement plants using digital process computer control techniques have applied them to the control of raw mix chemical composition, kiln-cooler control, sensor validity checking and alarm logging, production and trend data logging, and daily operating reports.

Grinding mill load control and DDC are also operating in a number of U.S. cement plants with DDC being more widely used in the last few years. Most new cement plants outside of the United States and Canada are designed by Europeans. In some instances selected European designed cement plants have been significantly altered in their design concepts to better interface with automatic process control principles and equipment. The purpose has also been to keep overall investment to a minimum. Thus a trend for such plants is to not only perform the control and other functions just listed, but to often include, as appropriate, additional functions of

1) quarry scheduling guidance;
2) prehomogenizing pile building control;
3) prehomogenizing pile building calculations;
4) grinding mill load control with maximizing as well as steady-state versions being utilized and with fineness loops, in some cases, being used based on continuous fineness sensors;
5) cement mix composition control;
6) cement silo monitoring and validity checking;
7) monitoring of drives for unscheduled shutdowns;
8) automatic start-up and shutdown of continuous process departments by programmed logic, including transient manipulation of regulating loops as well as on/off control of drives themselves;
9) control of overall load coupling for departments between which relatively low surge capacity for materials exists.

A more detailed description of many of these functions is given in [5].

Increasing Availability of Automatic Process Control Systems by Adequate Power Supplies

As briefly mentioned previously, a distinct need is to more thoroughly analyze the interrelation of the power supply for the automatic process control system and its major components and the power distribution system characteristics. The distinctive solution and trend found useful in many such automatic process control systems is to isolate the power supply for the process control computer and certain critical sensors (such as X-ray gauge and analyzer, oxygen analyzer, and selected instrumentation). This isolated power supply is often in the form of a separate induction or dc motor-driven alternator equipped with a flywheel and necessary control to ride through most power system transient disturbances. Such an isolated power supply, when properly designed, provides clean power to these control components.

The result of using such an isolated clean ride-through power supply is that the process control equipment is not harmed, or taken out of service, during momentary dips or voltage losses in the main power supply. In addition, the surges which sometimes get into the main power distribution system and its major components are kept out of the process control equipment.

As plants are designed which integrate automation and plant design together more carefully and include automatic start-up and shutdown of selected portions of the process, the isolated ride-through power supply concept is extended to also include power to the relays controlling the motor control centers themselves. This lays a basis for rapid restart of critical portions of the process following temporary shutdowns due to a short-term loss of voltage in the main plant power system.

In some instances the ride-through power supply becomes battery supported for, say, periods of from 5 to 30 min in order to permit standby auxiliary Diesel engine

generator sets to be activated and take over the function of supplying critical loads for slow turning of the kiln, for operating pumps, fans, and cooler grates during loss of power from the normal main power source to the plant.

The ability to quickly restart the continuous process parts of the plant after a shutdown due to temporary loss of power system supply voltage is becoming increasingly important in another way. Most interconnected public utility power systems must use high-speed reclosing on their main transmission lines to keep separate generating stations in synchronism during short circuits occurring on lines which interconnect such generating stations. Unless the short circuit is removed promptly and the interconnection between generators restored promptly, the separate generating stations tend to swing apart sufficiently so that they cannot be safely reclosed together without elaborate time-consuming resynchronizing provisions.

The effects of this high-speed short-circuit interruption and subsequent high-speed reclosing, as seen at the cement plant bus, is that power is lost for typically 1/3-1/2 s, after which power comes back from the utility. During that short time, most drives shut down due to their using instantaneous undervoltage protection.

Minicomputers for Cement Plant Control

During most of the time in which digital process computers have been applied to cement plants, adequate software with interface hardware and necessary peripheral equipment have been available only with medium-sized process control computers. These medium process control computers have the capability of doing any or all of the functions listed in the preceding section of this paper, either individually or simultaneously. Because such medium process computers have substantial total capabilities, they tended to be uneconomic when being considered for a single particular control function such as, for example, control of raw mix chemical composition.

Small or minicomputers have been available for many years. Yet by themselves they are relatively useless on a real-time process control job unless an adequate library of standard software and good application software especially tailored for real-time process control are available. Moreover, the lack of adequate supporting interface hardware particularly suited to the industries to be served had reduced their usefulness in such industries.

Surveys of many existing cement plants in the United States concerning the possibility of applying automatic process control have disclosed that there is a need and a potential usefulness for minicomputers if they are adequately equipped with supporting standard and application software, necessary interface hardware, and supporting training and other installation services.

Such a recent development is shown in Figs. 10-12. These figures show block diagrams of a line of minicomputers backed up by adequate standard software, hardware, and services all aimed at serving this segment of the cement manufacturing industry.

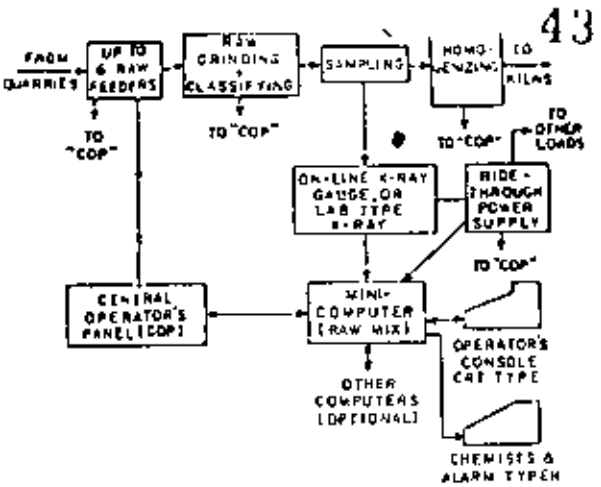


Fig. 10. Raw mix control with minicomputer package using X-ray sensing of chemical composition.

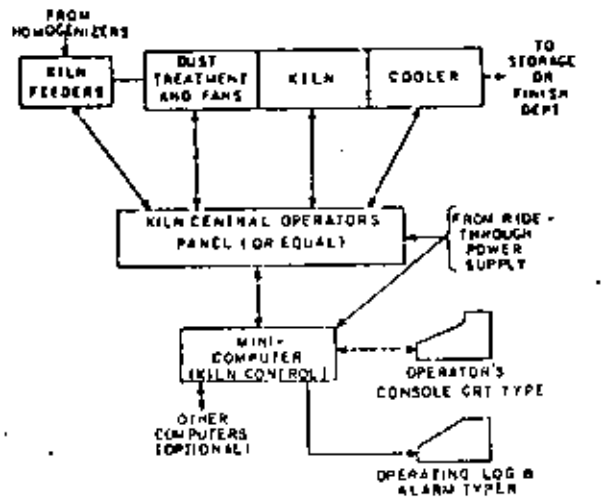


Fig. 11. Kiln-cooler control with minicomputer package.

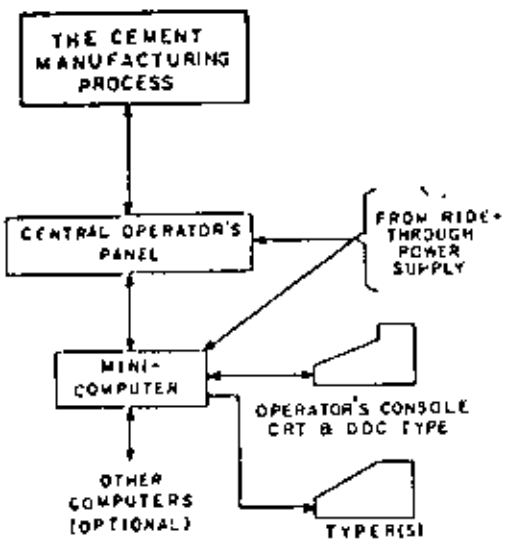


Fig. 12. Process data handling and DDC with minicomputer package.

The three minicomputer-based automation systems shown in Figs. 10–12 are for raw mix chemical composition control, for kiln control, and for extended digital capability (including generic DDC software). Each system has the following control equipment and operational features: they are all built around a minicomputer system of hardware and standard library programs designed to meet process control requirements; controls process through outputs to control devices typically on the COP; each utilizes modern reliable devices for interface with operator video-type of operator's console plus typer for combination operator's log and alarm log; each is a complete and independent system; each is powered from a ride-through power supply; and each can communicate with other minicomputer-based systems belonging to its own family as appropriate.

The raw mix system shown in Fig. 10 includes an X-ray analyzer for elemental analysis of the raw mix stream samples. The functional control provided by the pre-engineered software will control up to six raw material feeders, correcting feeder rates as necessary to maintain the sample analysis at the proper chemical composition. The system will also control up to two homogenizers to be batch or continuous filled for a specific chemical composition by adjusting the chemical composition hold point for the raw mix stream. A chemist's log with paper tape punch and reader for dumping and loading programs are included, as is the capacity for various custom options.

The kiln-control system shown in Fig. 11 provides the basic functions of maintaining stable kiln operation through proper setting and adjustment of the primary kiln variables such as feed rates, speed, fuel flow, and air-to-fuel ratio. Cooler grate and gas flow are also controlled as appropriate to the type of clinker cooler used. The system has the capability of being able to control the various types of kiln-cooler combinations, dry and wet feed, with and without preheater. Kiln performance log is included, as is the capacity for various custom options.

The system for extended digital capability provides the capacity for extensive data acquisition, for monitoring and alarming, and for daily production summary, and other logging functions; and combined with the extensive data acquisition capacity is the availability of DDC software for digital control of any or all process loops. The set points for the DDC process loops are normally set at the operator's video-type console, although supervisory controls of those set points from another minicomputer system family member is also possible (Fig. 12).

In addition to the previously described equipment and operational features, the successful minicomputer-based automation system must continue to include the full complement of all organizational backing by both users and supplier, including awareness, training, and commitment by the user, and including adequate installation start-up assistance, and follow-on service availability by the supplier. The minicomputer-based system then is simply an extension of the latest automation technology to meet the evolving needs of the cement industry. All of the fundamental requirements for success still exist and must continue to be met for successful automation to result.

Conclusions

We may conclude the following.

1) The cement plant owner and his representatives have available even wider choices than before as to the sizes and capabilities of automatic process control systems which they can economically use.

2) This broadening of the base for process automatic control computers to also include the minicomputers, adequately supported by standard software and interface hardware, means that many existing plants can have automation applied to at least portions of their process which may not have been very economical hitherto.

3) Adequate people factors including appropriate job assignments of operating personnel and their supervision, combined with good initial and continued training remain vital for success in automatic control.

4) The possibilities of modifying basic plant design for new plants to better adapt to the possibilities of automatic process control are exciting. They lay a basis for significantly changing for the better total plant investment and operating profitability.

5) Success in achieving automatic process control is not an accident. Success is best designed-in from the beginning. To be attained, it principally includes owner involvement from the beginning and thereafter plus heavy supplier involvement from the beginning but tapering off as operation on control proceeds.

6) Principal essential ingredients for success with the typical accountability for each have now been identified for automatic process control systems in cement manufacturing plants. Typical economic benefits derived from successful automatic process control by digital computer installation have also been identified.

References

[1] Computer Users Symposium, "Standard of process control computers for the cement industry," presented at the 1969 IEEE Cement Industry Technical Conference, Toronto, Ont., Canada, May 13–15.

[2] R. E. Evans and J. H. Betz, "Seven years of process computer control at California Portland Cement Company," *IEEE Trans. Ind. Gen. Appl.*, vol. IGA-6, Sept.–Oct. 1970, pp. 472–475.

[3] S. R. B. Opie, "Direct digital control—a total system approach," *1967 IEEE Conv. Rec.*

[4] R. Egger, "Design and conception of integrated automated cement plants," *IEEE Trans. Ind. Gen. Appl.*, vol. IGA-5, Nov.–Dec. 1969, pp. 752–758.

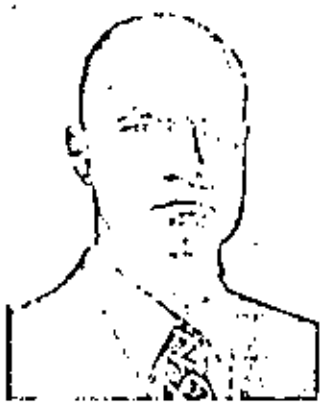
[5] M. R. Durlhui, D. L. Lippitt, and E. A. E. Rich, "Applications of digital computer control to the cement manufacturing process," presented at the 1968 Int. Seminar on Automatic Control in Lime, Cement, and Connected Industries, Brussels, Belgium, Sept. 9–13.

[6] J. Warshawsky and E. S. Potter, "Automatic sampling and measurement of surface area of pulverized material," *IEEE Trans. Ind. Gen. Appl.*, vol. IGA-5, Nov./Dec. 1969, pp. 773–776.

[7] F. Lebel, A. Guy, and D. E. Hamilton, "Computer direction of quarry operations," *Rock Products*, Mar. 1967.

[8] S. Levine, "Sophisticated sampling systems optimize computer operations at Allentown Portland Cement," *Rock Products*, Apr. 1967.

[9] D. D. Bedworth and J. R. Faillace, "Instrumenting cement plant for digital computer control," *ISA J.*, Nov. 1963.
[10] J. R. Ronnig, W. R. Morton, and R. A. Phillips, "Making cement with a computer control system," presented at the 1961 IEEE Cement Industry Technical Conference, Pasadena, Calif., Apr. 14–17.
[11] J. R. Ronnig and W. R. Morton, "Application of a digital computer to the cement-making process," *IEEE Trans. Ind. Electron. Contr. Instrum.*, vol. IECI-13, Apr. 1966, pp. 2–9.
[12] E. A. E. Rich, "Cement automation—1965," presented at the 1965 IEEE Cement Industry Technical Conference, Allentown, Pa., May 12–14.
[13] J. Seringeour, "Instrumentation and control for industrial minerals—current and future," *Can. Mining Met. Bull.*, July 1967.
[14] G. J. Dick and R. G. Schlauch, "Large size clinker cooler operations," presented at the 3rd Annu. Cement Industry Operations Seminar, Chicago, Ill., Nov. 26–28, 1967.



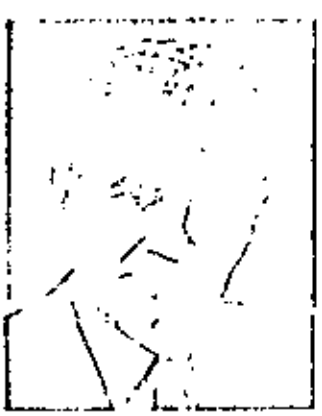
E. H. Gautier, Jr., was born in Mobile, Ala., on April 26, 1925. He received the B.S. degree in electrical engineering from Duke University, Durham, N.C., and has engaged in postgraduate work at the University of Pittsburgh, Pittsburgh, Pa., and the Virginia Polytechnic Institute, Blacksburg.

He has been with the Westinghouse Electric Corporation, the Elliott Company, and the Pittsburgh Plate Glass Company. He joined the General Electric Company in 1962 and is presently Senior Sales and Application Engineer at their Manufacturing and Process Automation Division, West Lynn, Mass. He is experienced in the automation practices of industrial drives and processes in the cement, steel, and glass industries.

Myron R. Hurlbut (S'55–M'58) was born in Klamath Falls, Oreg., on September 2, 1936. He received the B.S. degree in electrical engineering from Oregon State University, Corvallis, in 1958.

He joined the General Electric Company upon graduation and completed their Advanced Engineering Program in 1961. After a year as Supervisor of the Advanced Engineering Program, he spent four years working in computerized patient monitoring as part of the medical electronics work at the X-Ray Department in Milwaukee, Wis. Since 1966 he has worked in cement plant automation, first with the Process Computer Department, Phoenix, Ariz., and presently with the Manufacturing and Process Automation Marketing Department, West Lynn, Mass. He has been awarded three patents for cement kiln control systems.

Mr. Hurlbut is a member of Phi Kappa Phi, Tau Beta Pi, Eta Kappa Nu, and Pi Mu Epsilon, and is a Registered Professional Engineer in the State of Ohio.



Edward A. E. Rich (M'38–SM'45) was born in Salt Lake City, Utah, in 1916. He received the B.S. degree in electrical engineering from the University of Utah, Salt Lake City, in 1937.

He has been with the General Electric Company since 1937. He spent two years in various test department assignments and eight years as a designer of synchronous machines. In 1947 he joined the Industry Sales and Engineering Operation Schenectady, N. Y., and is now principally involved in General Electric's engineering and sales relations with the glass and cement industries. He is the author of numerous IEEE and other technical papers presented in the United States and in other countries.

Mr. Rich served as Secretary of the Cement Industry Committee of the IEEE IGA Group for eleven years and is presently Chairman of the Control Working Group of the Glass Industry Committee. Since 1966 he has served as Vice Chairman, Papers, of the Technical Operation Department of the IGA Group. Has also served as Member-at-Large of the Group Ad Com in 1970. He received the 1969 Distinguished Service Award of the Cement Industry Committee.

The operation of a paper machine is critically affected by the complex interaction of subsystems that traditionally are subjected to independent control action. This article describes the design of a control system for a paper machine that takes into account such interaction, and shows how the design technique may be applied to a basis-weight/moisture-control system.

Interactive Control of Paper Machines

E. B. DAHLIN, Measurex, Inc.

Good paper-machine control must include coordination of such subsystems as refiner, headbox, and dryer, and speed and stock feed. Without such coordination, control actions that are taken in one part of the overall process may be major sources of upset in another part. Specifically, refiner adjustments may upset moisture control; speed changes may influence paper formation unless compensatory headbox actions are taken; setpoint changes in basis weight may cause both an upset in moisture content and a variation in sheet strength.

Effective handling of interactions among paper-machine subsystems may be initiated by constructing a control system that takes advantage of existing analog controls while employing the full power of digital computing techniques. The resulting control system is not too complex, yet greatly improves the output product, and at the same time keeps open as much digital computer capacity as possible for more sophisticated algorithm implementation.

For a general approach to mathematical modeling for the paper industry, Ref. 1 is suggested. Previous work by the author on certain algorithms appears in Refs. 2, 3, 4, and 5. The experimental data used in this article was gathered as described in Refs.3and 5.

Paper machine influences

The table on the next page establishes qualitative relationships among essential independent variables that may be either random disturbances or manipulation inputs, and an array of dependent variables. Variables preceded by an asterisk are normally manipulated and variables that can be observed are boxed.

Control objectives may be defined from a study of this table. The basic need for automatic regulation stems from the existence of the disturbance variables. Measured and manipulated variables afford possibilities for forming feedback control loops. The table also indicates simultaneous effects of manipulated variables upon variables related to specifications for the product quality, such as basis weight, moisture, and formation.

It is often useful to stabilize fiber flow by cascaded control around the stock valve. In the table, fiber flow is identified as dry stock flow (DSF), and is defined as the product of consistency and stock flow. The feedback loop, closed from calculated DSF to stock valve position, will be affected by consistency reading noise and temperature impact on the consistency meter calibration.

However, these steady-state calibration errors are not too critical to good control because the long-term behavior of fiber flow is determined by feedback of basis weight. The usefuless of the calculated DSF loop is rather to prevent short-term variation in consistency from upsetting the basis weight and the moisture content of the reel.

The feedback loop from steam pressure to steam valve position is normally implemented with analog pneumatic controllers. Within the dynamic range of this control loop, the effects of steam line pressure and flash tank pressure are prevented from propagating through the system to influence dryer heat flow rate and reel moisture.

Interactive computer control system

The block diagram of Figure 1 shows how a digital computer may be applied to a paper machine for the purpose of improving product quality through more sophisticated control of interactions among operating subsystems. The computer receives measurements of quality-defining variables and performs highly complex analysis of, for example, nonlinear calibration characteristics and calibration parameters for different paper grades. Production rates, fiber consumption per produced reel, and means and variances of quality-defining variables can be prepared regularly for management. The computer can calculate on-

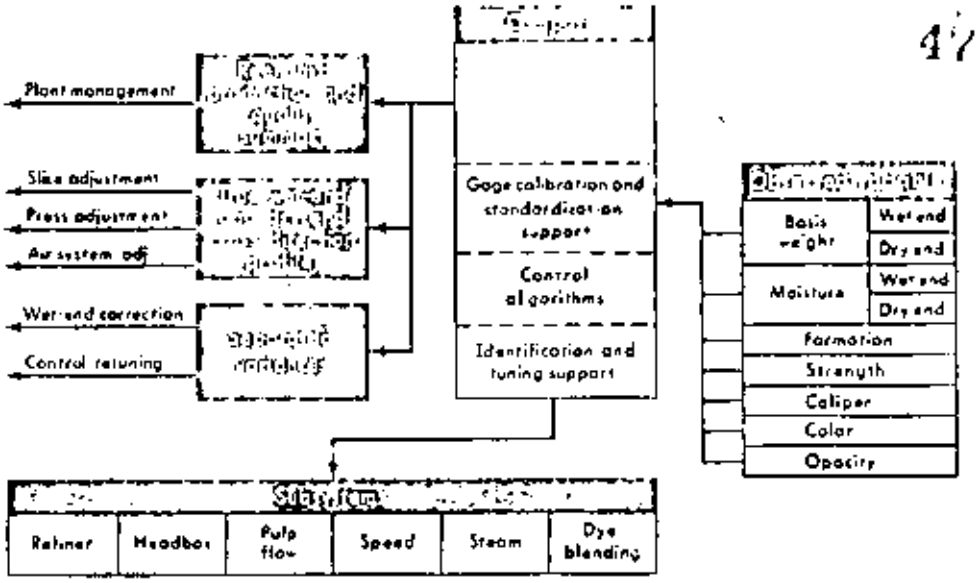


FIG. 1. Block diagram showing how a digital computer may be applied to a paper machine to improve quality of the product through more sophisticated control of interactions among the several subsystems of the wet and dry ends.

tine the true cross-direction profiles of, say, basis weight, bone-dry weight, and moisture.

These and other complex functions are routinely performed as shown in Figure 1, where the computer implements algorithms to supplement the analog control of local operations. This system incorporates features that are not generally included in paper-machine control.

To illustrate the methods used, a relatively simple system for controlling basis weight and moisture will now be discussed (Figure 2).

The subsystem for pulp-flow control consists of a flowmeter feeding a signal to an analog flow controller, which then manipulates the stock valve position. A consistency meter transmits a signal to the digital computer. A program provides digital filtering for eliminating high-frequency noise associated with the consistency measurement. After noise removal, the program calculates flow set-point corresponding to fiber flow (dry stock flow or DSF). Implementation of this loop with a mixture of analog and digital hardware provides a profitable balancing between dynamic performance and cost.

The dryer-control system is a conventional pneumatic control loop regulating the steam pressure in a dryer section, usually the one directly ahead of the basis weight and moisture scanner.

The supervisory controller utilizes measurements of basis weight and moisture obtained (preferably) at scanning speeds between 500 and 1,000 in. per min. These speeds enable the computer to have better process information to work with in, for example, calculating cross-direction profiles.

Process identification

The objective of process identification is to determine process dynamics parameters—a must step in constructing the process mathematical model if an adequate control system is to result. The parameters are

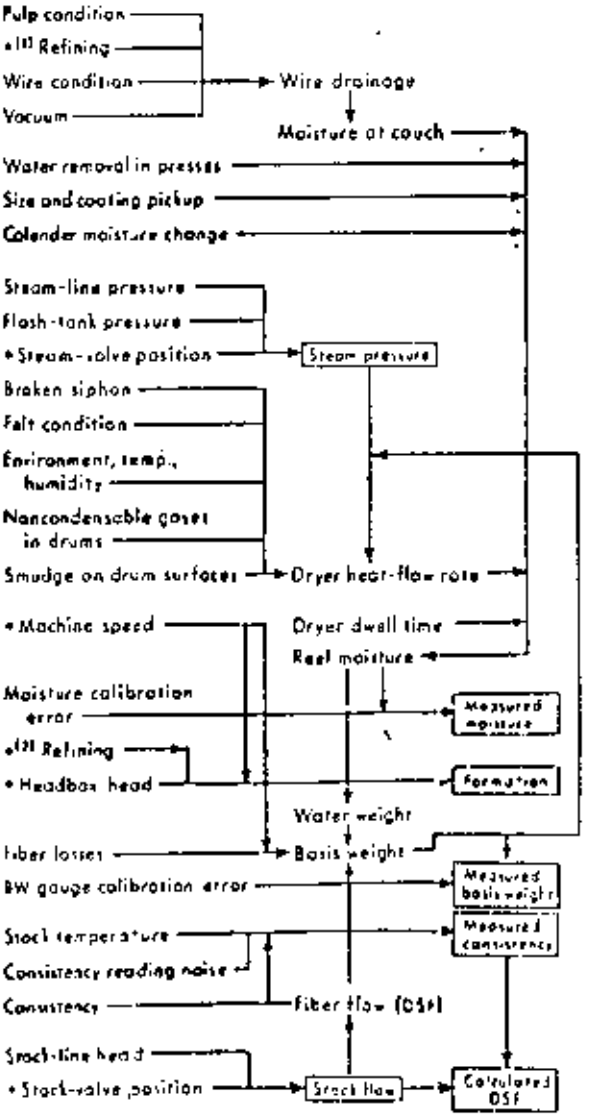


Table - Paper Machine Influences

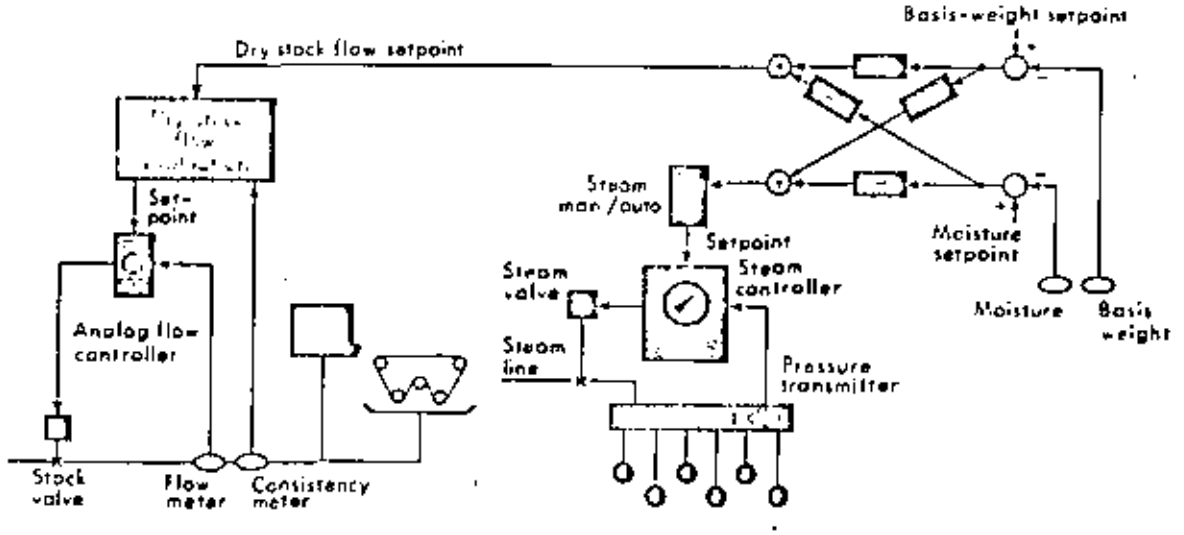


FIG. 2. An illustrative example of a combined basis-weight and moisture-control system, the subject of detailed discussion in this article.

required for both the design and the tuning phases.

The procedure is to introduce small upsets in the manipulated variables and collect the response data in computer memory. After a small perturbation of the stock valve, for instance, a variation in stock flow, consistency, basis weight, and moisture will be observed. Again, a small variation in steam pressure setpoint will cause changes in steam pressure, moisture, and basis weight that are transmitted to the computer. The required parameters are obtained by analysis of such response data.

The criterion for a good model is that it provide the basis for good control-system tuning. The degree of needed accuracy of parameter estimation also depends on the loop sensitivity to discrepancies between assumed model structure with parameter values and actual process dynamics.

In the application being discussed, an adequate model structure can be derived from the wet-end model equations of Beecher, Ref. 1. Ignoring the head box time constant, the simplified linear model developed by Beecher is

$$\frac{\Delta BW}{\Delta DSF} = K_0 e^{-\tau s} \frac{1 + \tau_2 s}{1 + \frac{\tau_2 s}{r_f}} \tag{1}$$

where ΔBW = basis weight change at the reel

ΔDSF = change of concentrated pulp flow to paper machine (gallons dry pulp per min.)

K_0 = gain constant

τ = transport delay from stock valve to reel

τ_2 = mixing time constant in the wire pit

r_f = the fraction of fiber flow that does not circulate through the wire pit (retention)

s = Laplacian operator

This model being acceptable for the wet end of the machine, the dryer is next considered. The nature of a response to a small steam-pressure change is obtained from a heat-flow analysis of the drum, Figure 3. If the felt is the same temperature as the pocket air, the heat flow per unit area may be modeled as in Figure 4. The driving signal is the temperature on the inner surface of the condensate. The sheet is considered as having a wet and a dry section, the latter resisting water removal. The rate of water removal is

$$F_w = CU_{sw}(T_s - T_w) \tag{2}$$

where C = reciprocal enthalpy of water evaporation

U_{sw} = heat conductivity per unit area between the wet sheet node

T_s = temperature of the fibers in the sheet

T_w = temperature of the water in the sheet

With temperature as the analog of voltage, heat capacity as the analog of electric capacity, and heat-transfer numbers as analogs of electric conductivities, the transfer function for Figure 4 may be written

$$\frac{F_w(s)}{T_1(s)} = \frac{C\eta}{R_t} \frac{AB}{(s + A)(s + B)} \tag{3}$$

where

$$\eta = \frac{U_{sw}}{U_{sw} + \dfrac{1}{R_{ss} + R_{ww}}} = \text{incremental dryer efficiency} \tag{4}$$

$$R_t = R_{11} + R_{22} + R_p \tag{5}$$

$$R_p = R_{ss} + \frac{1}{U_{sw} + \dfrac{1}{R_{sw} + R_{ww}}} \tag{6}$$

$$R_{ij} = 1/U_{ij}$$

A, B = characteristic radian frequencies of the network

When the sheet is transported through the dryer section—all of whose drums are assumed to have the

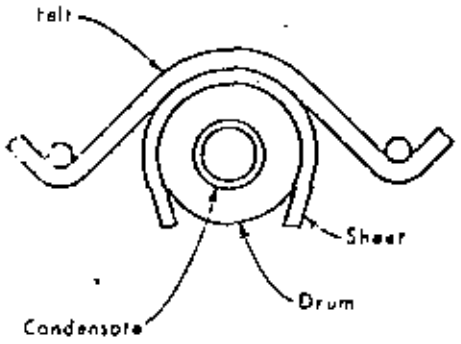


FIG. 3. Cross-section of dryer drum used in develop model, Figure 4.

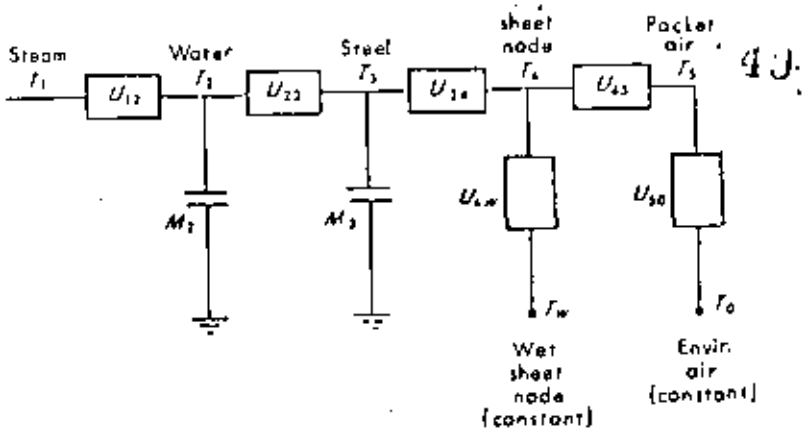


FIG. 4. Electrical analog network for dryer drum, explained in text.

same characteristics- the total water content of the sheet element is

$$w(t_1; t_0) = w(t_a; t_0) - \int_{t_a}^{t_1} \gamma F_w \, dt \qquad (7)$$

where $w(t_1; t_0)$ = weight of water per unit area at time of the element that entered the dryer section at time t_0.

γ = the fraction of the total dwelling time in the section during which the sheet element is in contact with the steam drums

From the concept of Equation 7, the water content seen by a fixed observer located at the end of the dryer section and watching the sheets go by can be expressed by the differential equation

$$\frac{dw(t)}{dt} = \frac{dw_0}{dt}(t - \tau_d) - \gamma \left[F(t) - F(t - \tau_d) \right] \qquad (8)$$

where $w(t)$ = water weight per unit area at dryer end

$w_0(t)$ = water weight pua at dryer entry

τ_d = dwelling time in the dryer section

Taking the Laplacian transform of Equation 8 and combining the result with Equation 3 yields the dryer-section transfer function:

$$\frac{w(s)}{T_1(s)} = -\frac{\gamma C \mu}{R_t} \frac{A}{s + A} \frac{B}{s + B} \frac{1 - e^{-\tau_d s}}{s} \qquad (9)$$

The constants A and B depend upon the heat-transfer coefficients and the heat capacities of the system, and are difficult to estimate. In a typical dryer, A varies widely while B is relatively independent of the heat transfer to the drum, Figure 5. The A reflects the condition of the internal heat transfer of the drum. Analysis of heat transfer between drum and sheet, wet and dry sheet nodes, and sheet to air pocket shows that A and B are both fairly independent of these parameters.

Equations 1 and 9 define a reasonably adequate model structure for purposes of process identifica-

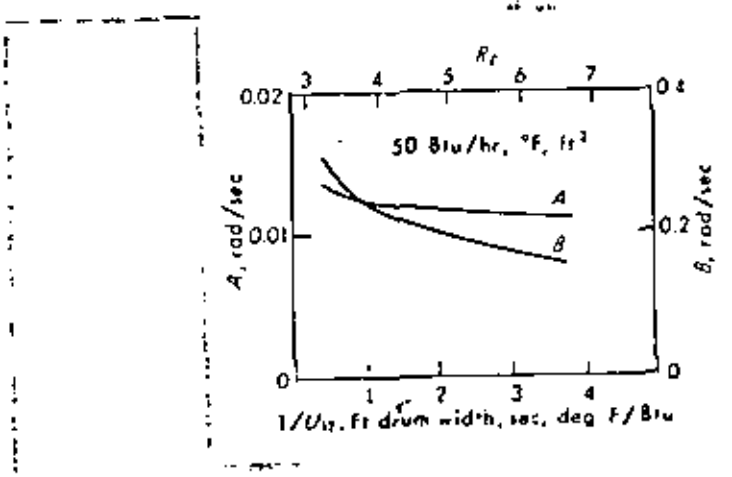


FIG. 5. Comparison plot of characteristic radian frequencies A and B of the dryer-drum analog of Figure 4.

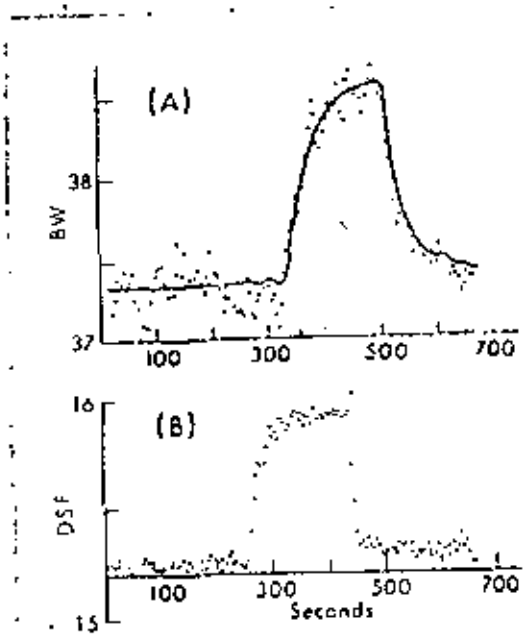


FIG. 6. Response to perturbations during an identification experiment of A (basis weight) and B (dry stock flow). The solid curve of Figure 6A is the response to the data in Figure 6B when used at a driver of the basis-weight model. Correspondence of this curve with the plotted data shows that a good model has been constructed

tion. It is, in practice feasible, Ref. 3, to combine these two structures into a single form:

$$Ke^{-\tau s}\,\frac{C}{s+C} \qquad (10)$$

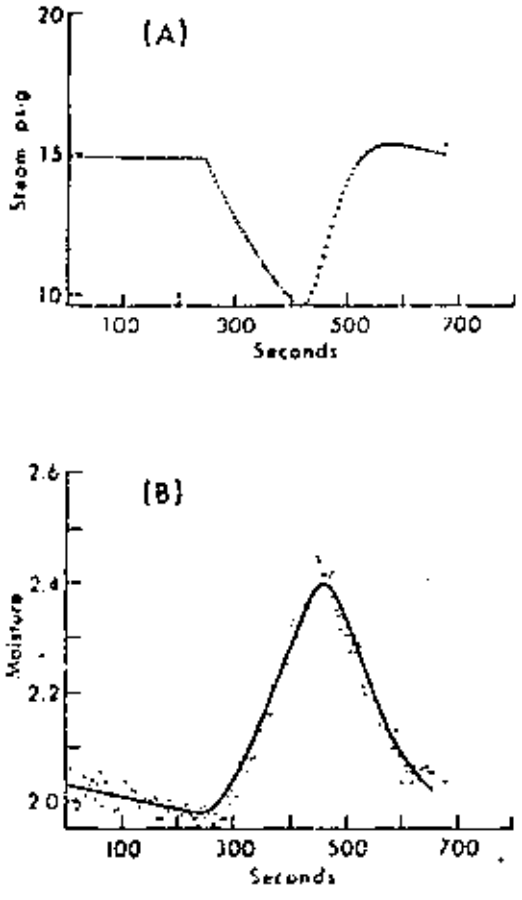
where K, τ, and C are gain, transport delay, and pole, respectively. Each of these parameters is individually determined for the two transfer functions of Equations 1 and 9. If this simplified form is used, there must be very short intervals between perturbations on the identification process, Ref. 3. A method for determining K, τ, and C by analysis of the data from an identification experiment is given in Refs. 2 and 6.

Figure 6 shows the response to perturbations during an identification experiment of A, basis weight, and B, dry stock flow. When the input time series data of Figure 6B is used to drive the basis-weight model, the solid curve of Figure 6A results, indicating by agreement with plotted data that a good process model has been established in the computer.

Exposing the steam-pressure setpoint to perturbations yields the nonlinear behavior shown in Figure 7A, revealing the water-removal limitations of the steam drums. Figure 7B shows correspondence between observed response of moisture content and response obtained from driving the model with the data of Figure 7A.

The method illustrated here has been tested over many variables on a variety of paper machines. The conclusion is that effective tuning of larger systems can be accomplished by this method of process identification.

FIG. 7.A—Plot of steam pressure variation obtained by exposing its setpoint to step perturbations. B.—Correspondence between observed response of moisture content and response obtained by driving the model with the data of Figure 7A.



The control algorithms

For the system of Figure 2, algorithms are designed to obtain a specified response to setpoint changes in either moisture or basis weight. This response is overshoot-free and has exponential settling characteristics. Observations of closed-loop operation have been made on many installations, and the effectiveness of loop decoupling and transport delay has been verified, Refs. 4 and 5.

The process model of Figure 8 is used for controller design. The hold blocks maintain a continuous output signal updated periodically by the computer, which acts as a sampling device. The hold functions are incorporated in the steam manual-auto station and the analog flow controller shown in Figure 2.

The closed-loop dynamics of the steam-pressure loop are represented by a single pole, E. Cross-coupling characteristics are indicated by parameters α_1 and α_2. An illustrative example of crosscoupling networks is given in Figure 9 and discussed later in some detail.

The dynamic effects of scanning, and of the resulting control by the cross-machine averages for moisture and basis weight (alternately bone dry and conditioned), are included in the model of Figure 8. The averages are calculated from sums formed over the samples taken during a single scan. In the model, continuous integration serves as an approximation for this calculation. Such approximation significantly reduces the complexity of the final control algorithm without producing any effect on control-system tuning. The symbol z^{-1} indicates a time shift equal to the scanning time increment T.

In Figure 10, speed of settling is shown as being dependent on a parameter (λ) that is chosen as high as possible consistent with permissible frequency and amplitude for steam pressure and stock-flow changes demanded by the controller. Better control requires greater activity in manipulating variables. Often, however, the maximum value of λ is determined by overshoot characteristics generated by nonlinear phenomena not accounted for in the model.

As to decoupling, let it be assumed that a moisture setpoint change is made, Figure 9. The algorithm $C11$ will then see a positive moisture error d and call for a steam pressure decrease e. When steam pressure

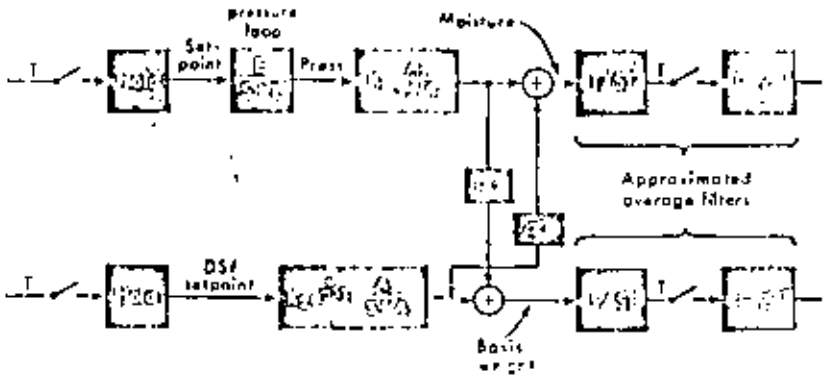


FIG. 8. Model for design of a cross-coupled controller described in text. Cross-coupling not only reduces variable interaction but produces a faster, tighter control response.

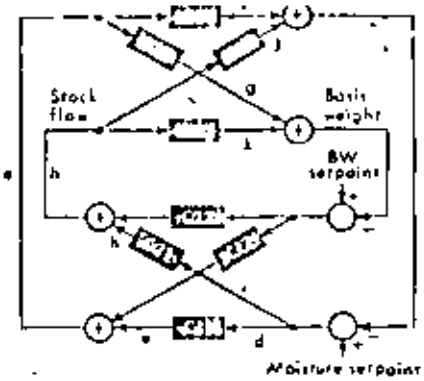


FIG. 9. Illustrative cross-coupling network for system of Figure 8, showing how disturbance of either setpoint will result in an offset of the expected disturbance in the compensation control channel.

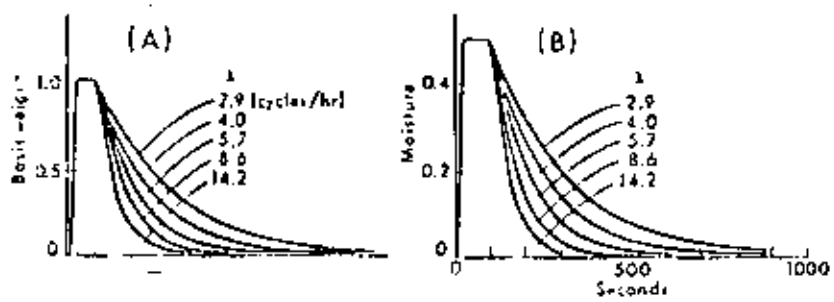


FIG. 10. Settling speed curves for basis weight (A) and moisture (B) shown dependent on the value of parameter λ explained in text.

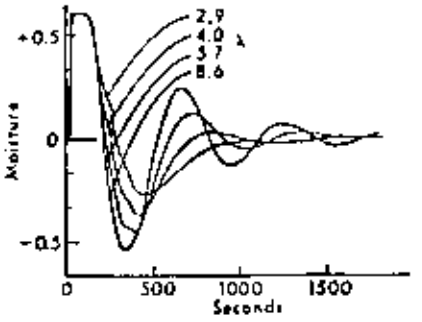


FIG. 11. Response curves of the moisture side of controller with all cross-coupling removed. Note oscillatory results for various values of the parameter λ.

decreases, basis weight and moisture will be affected according to response curves indicated by *f* and *g*. Due to process interaction, not only is moisture changed but basis weight is also upset. Eventually the basis weight algorithm *C22* would correct the basis weight error, but this needless upset is avoided by operation of the decoupling algorithms *C12* and *C21*.

Algorithm C21 will have already seen the moisture error *d* that occurred with the setpoint change, and called for a stock-flow decrease *h*. This has resulted in a nullifying effect on moisture and basis weight as indicated by signals *j* and *k*. Proper selection of algorithm *C12* can similarly offset the effect of a basis-weight setpoint change on moisture.

In addition to interaction-free setpoint change, the decoupled controller achieves much faster control action than do independent basis weight and moisture controllers, Figure 11. Using normalized units to indicate deviation from setpoint, the graph shows moisture response to an upset when the two controllers are applied without decoupling algorithms. The graph may be compared with moisture response in Figure 10B, where much tighter control is obtained with good stability.

The decoupled controller prevents unnecessary control actions. When a consistency variation causes simultaneous upset in basis weight and moisture, for example, the combined effect of the algorithms in Figure 9 will cause a stock-valve correction, leaving steam pressure unaffected.

It is hoped that this discussion of a combined basis weight and moisture-controller design has indicated how the basic principles may be applied to much more complex control systems.

REFERENCES

1. "Dynamic Modeling Technique in the Paper Industry," A. E. Beecher, *TAPPI*, Vol. 48, No. 2, pp. 117-120, February 1963.
2. "On-Line Identification of Process Dynamics," E. B. Dahlin, *IBM Journal of Research & Development*, Vol. 11, No. 4, pp. 408-426, July 1967.
3. "Process Identification and Control on a Paper Machine," E. B. Dahlin and L. B. Sanborn, IFAC/IFIP Symposium, "Digital Control of Large Industrial Systems," Toronto, Canada, June 17-19, 1968.
4. "Designing and Tuning Digital Controllers," Part I, E. B. Dahlin, *Instruments and Control Systems*, June 1968.
5. "Designing and Tuning Digital Controllers," Part II, E. B. Dahlin, R. L. Zumwalt, M. G. Harner, and W. A. Wickstrom, *Instruments and Control Systems*, July 1968.
6. "Process Identification for Control System Design and Tuning," E.B. Dahlin, Measurex Corp., and D.B. Brewster, Westvaco Corp., *Control Engineering*, April 1969, p. 81.

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

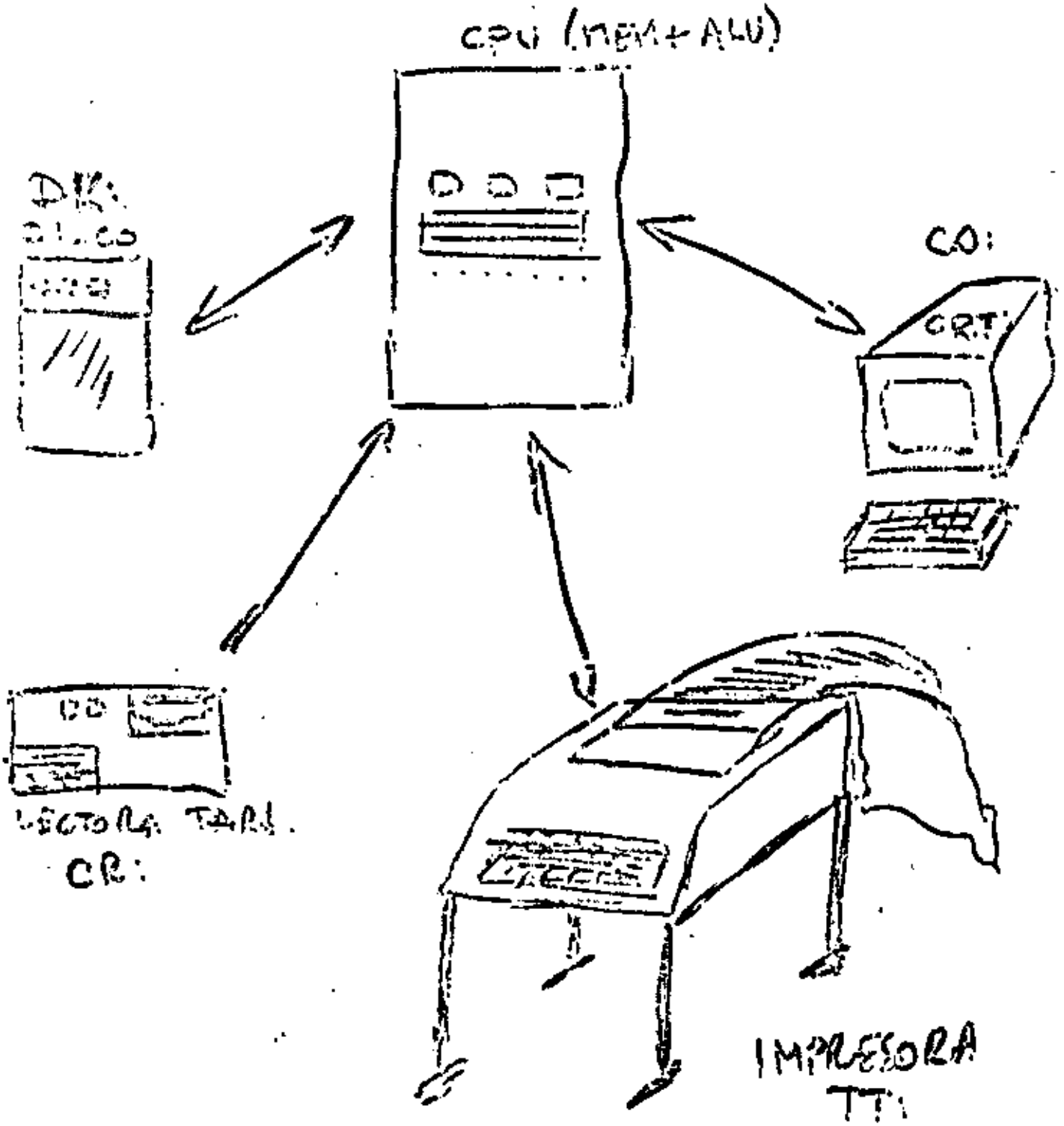
INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

MCR, EDI, FOR, MAC, TKB, PIP

M. EN C. JOSE ARMANDO TORRES FENTANES

JULIO, 1980

SISTEMA PDP/11-XX



OBJETIVO

- APRENDER A DESARROLLAR PROGRAMAS SENCILLOS EN LA PDP11 EMPLEANDO RSX11

ELEMENTOS A CONSIDERAR

- TERMINAL
  - TECLADO
  - ACCESO
  - COMANDOS SIST. OPERATIVO
- DESARROLLO DE PROG.
  - CREAR y EDITAR
  - COMPILAR
  - LIGAR
  - EJECUTAR
- ARCHIVOS
  - ESPECIFICACION
  - MANEJO
- PERIFERICOS
  - SELECCION
  - ASIGNACION

PROGRAMAS DEL SISTEMA A EMPLEAR

* MCR (MONITOR CONSOLE ROUTINE)
  Comandos para controlar la operación del sistema desde la terminal e instalar tareas

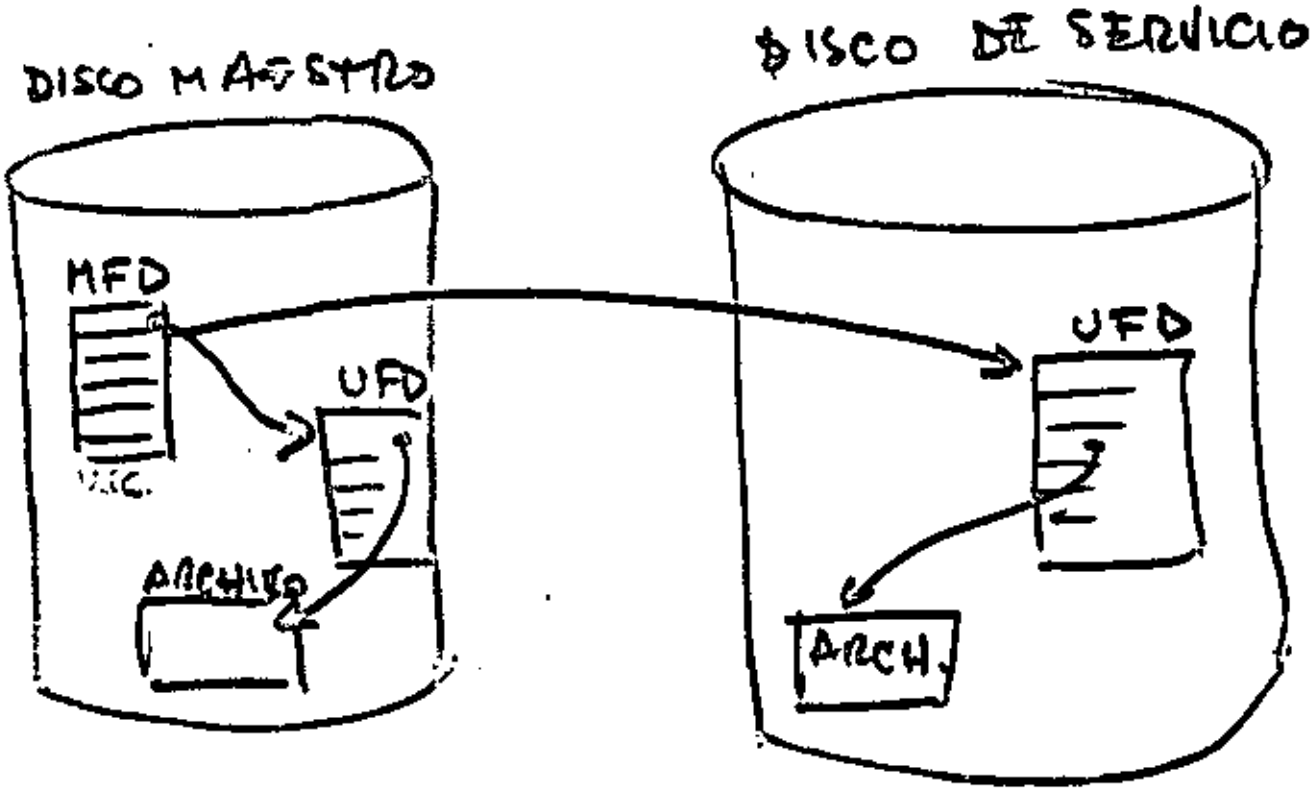
- EDI (TEXT EDITOR)
  crear archivos de
  - datos
  - prog. fuentes
  - comandos

- FOR y MAC
  Compiladores para crear prog. objeto

- TKB (Task Builder)
  Crear tareas ejecutables

- PIP (Peripheral Interchange Program)
  Manejo de archivos periféricos y

ORGANIZACION y NOMBRE DE ARCHIVOS



FORMATO PARA NOMBRE:

dispositivo: [UIC] nombre . tipo ; versión

MTA : [6,1] RUTA . MAC ; 4

disp. = 2 letras + 1 ó 2 dígitos + : { real / ficticio

UIC = g, m ; g, m ∈ [1, 377₈]

nombre = 1-9 caracteres

tipo = 3 letras

versión = número ∈ [0, 77777₈]

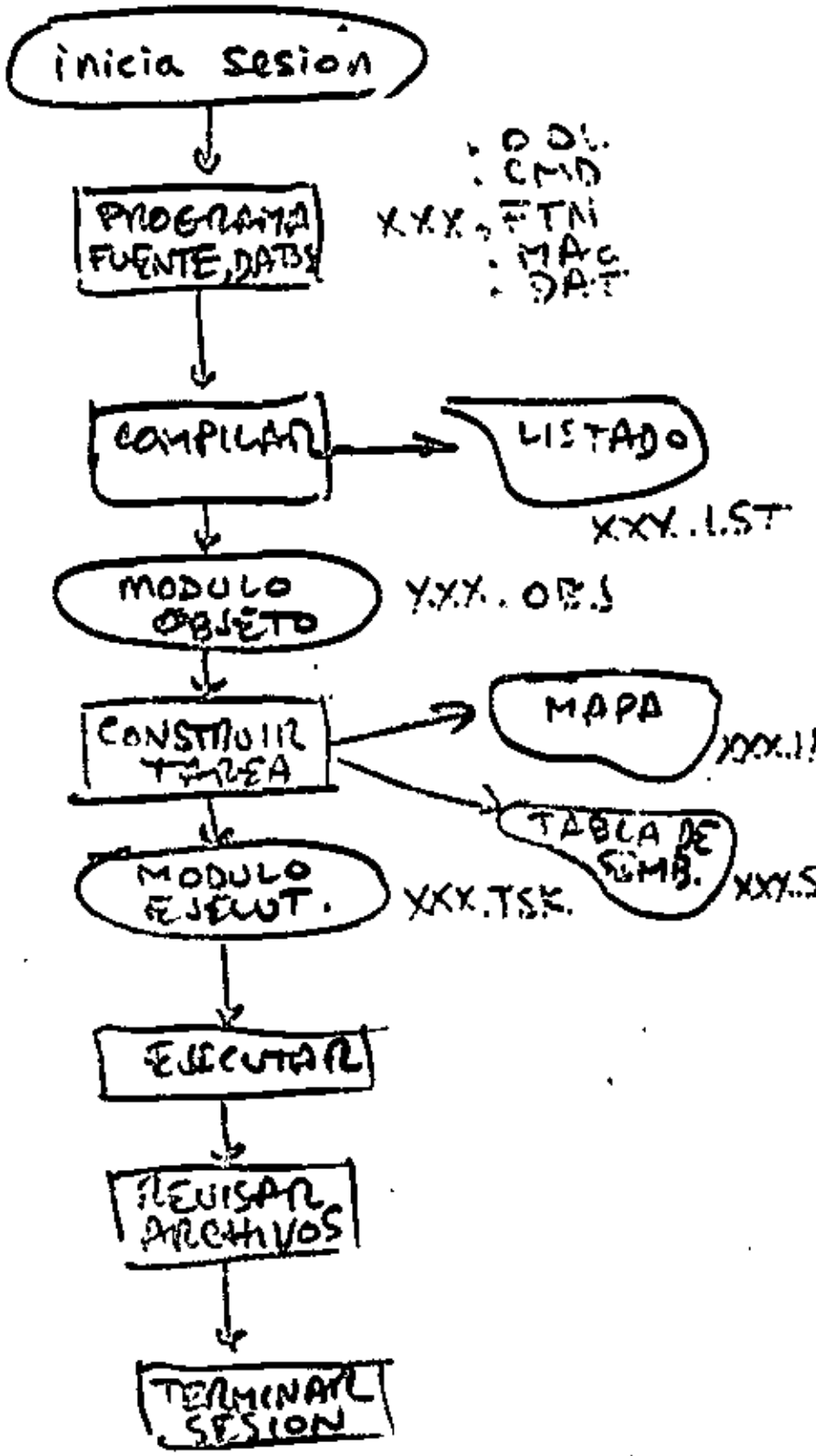
TIPOS DE ARCHIVOS

BAS
CBL
FTN
MAC
OBJ
TSK
MAP
LST
DAT
STB
ODL
CMD
SYS

DESARROLLO DE UN PROGRAMA y ARCHIVOS QUE SE GENERAN

MCR

>HEL



Cuando archivo sal = archivo entrada

TERMINAL

TIPOS { CRT
DECWRITER (LA30)

ELEMENTOS { TECLADO { caract. alfanum.
caract. de control
MEDIO DE DESPLIEGUE { Pantalla
Papel

TECLAS DE CONTROL { C.R.
CTRL.
RUBOUT or DEL
TAB

CARACTERES DE CONTROL
CTRL + letra

↑C	→ MCR
↑O	no listar
↑S	detener list.
↑Q	reanudar ↓
↑R	(CR) + imprime línea corrig.
↑U	(CR) + borra
↑Z	termina tarea y regresa a MCR
↑L	FF

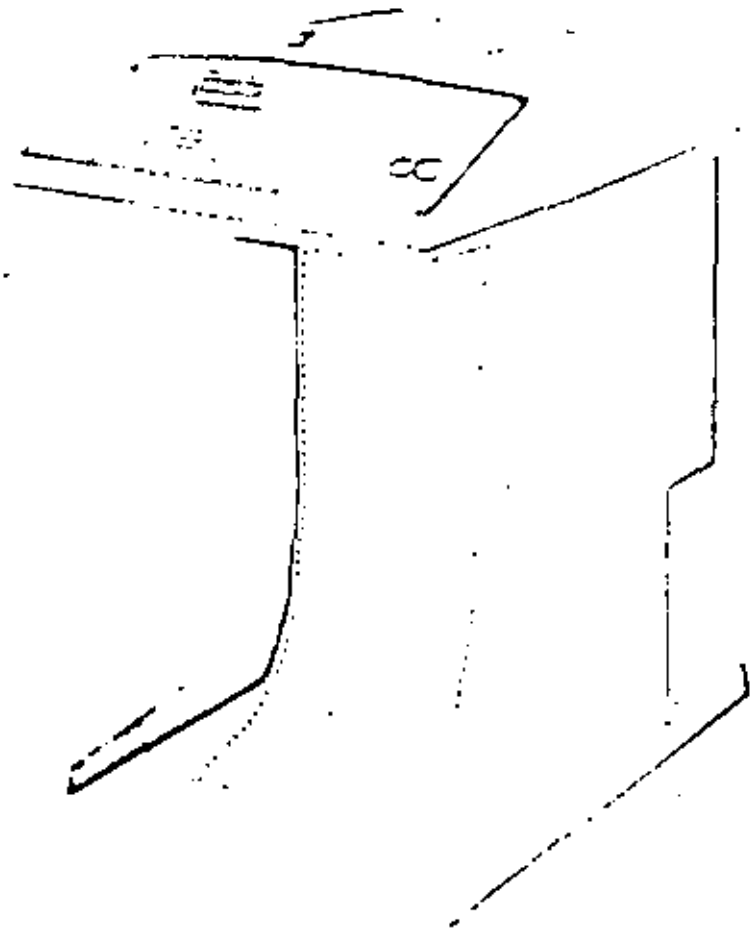


Figure 1-1 An LA30 Hard-Copy Terminal

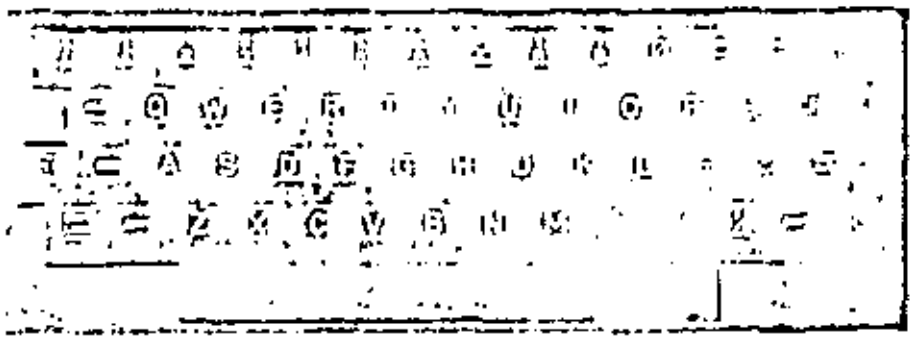


Figure 1-3 Keyboard of an LA30 Terminal

MCR

FORMATO COMANDOS

comando ^ parámetro(s) / switch[=valor] /...

- comando: 3 caract.
- parámetro(s): Tarea del sistema dispositivo

REQUISITO PARA DARLOS

- Terminal encendida
- Sesión iniciada
- Teclear (CR) para obtener >
- ó tambien ^C
- Ser usuario privilegiado para algunos comandos

TIPOS DE COMANDOS

- Protección multiusuario
  - ALL
  - BYE
  - DEA
  - HELLO
  - HELP
- Enicialigación
  - BAD
  - ASN
  - ACS
  - BOOT
  - DMO
  - INI
  - INS
  - SET
  - TIM
  - UFD

- Información
  - DEV
  - LOG
  - FAIL
  - TIME
- Control Tareas
  - ADD
  - ALT
  - CAN
  - CLO
  - FIX
  - REA
  - RED
  - REI
  - RES
  - RUN
  - UNF
- Mantenimiento Comunicación
  - ACT
  - ATL
  - BRK
  - BRO
  - OPE
  - SAV
  - TRA

EJEMPLOS

- inicio de sesión
  - > HEL (CR)
    ACCOUNT or NAME : [UIC]
    PASSWORD : password (1-6 caract.)
  - > HEL [UIC] (CR)
    PASSWORD : password
  - > HEL [UIC]/password.

- fin de sesión
  - > BYE

- instalar tarea
  - > INS A.B /TASK=CVD /PRI=65. /PAR=PAR14K

    switches CKP {Y / N
    INC = nn. inc. din mem.
    PAR = ____
    PMD {Y / N
    PRI = nn.
    SLV = {Y / N
    TASK = nombre
    UIC = [g, m]

    + rápido para ejecutar que RUN.

password	is a 1- to 6-character alphanumeric string. The account information maintained by the system includes the correct passwords for each UIC and last name. You cannot gain access to the system unless you type the password that corresponds to the UIC or name you have entered. So that your password remains private, the system does not print the characters you type in response to PASSWORD:.

(13)

If you do not know your UIC or password, contact the system manager, or whoever controls the use of the system at your installation.

Example:

```
>HEL <CR>
ACCOUNT OR NAME: CHARLES <CR>
PASSWORD: GREY <CR>

          RSX-11M BL10 MULTI-USER SYSTEM

GOOD MORNING
22-MAY-77 11:07 LOGGED ON TERMINAL TT4:

22-MAY-77

      SYSTEM WILL BE DOWN TODAY FROM 13:00-15:00 FOR
      CORRECTIVE MAINTENANCE
```



```
>HEL <CR>
ACCOUNT OR NAME: CHARLES  <CR>
PASSWORD:          <CR>

          RSX-11M BL10 MULTI-USER SYSTEM

GOOD MORNING
22-MAY-77 12:14 LOGGED ON TERMINAL TT4:

22-MAY-77

      SYSTEM WILL BE DOWN<CTRL/D>
```

Suppressing ACCOUNT OR NAME: Prompt

Another way to shorten the time it takes to log on is to enter the Hello command and UIC (or last name) on the same line:

```
 HEL 201/312 <CR>
PASSWORD:
```

Entering the command in this way eliminates the need for the prompt ACCOUNT OR NAME:.

- Correr programa en archivo

  >RUN archivo/switches { instala ejecuta remueve

- Correr tarea instalada

  >RUN tarea/switches

  └→ en STD

  switch RSI=nT

  # ┘ └ unidad / H M S T

- cancelar ciclo ejecución

  > CAN Tarea

- cambiar asignación unidades lóg

  >LUN /SY:L/TTØ:5/

- abortar tarea

  > ABO tarea /PMD

- alterar prioridad

  > ALTER Tarea /PRI:nn.

- montar dispositivo

  > MOU DDnn:[etiqueta]/FPRO/OVR

FPRO = [SY, OW, GR, WO]
↑ sist. ↑ dueño

D	E	W	R
1	1	1	1

1 ⇒ No permitido
0 ⇒ Permitida

- crear UFD

> UFD DDun: [g,m]

> ALC DK1:
> MOU DK1: /OVR
> UFD DK1: [12,12]
> DMO DK1:
> DEA DK1:

- reasignar unidades lógicas para tarea

>REA tarea unidad dispositivo:

- redireccionar E/S

>RED N_{nn} : O_{nn} O⇒N

- establecer condiciones de trabajo

> SET / opcion=...[/...]

BUF = TT0: 132.
UIC = [6,6]

MONITOR CONSOLE ROUTINE (MCR) COMMANDS

In this section, (P) indicates that a command format or keyword is privileged

Nonprivileged MCR Commands

The commands described below can be issued by any user.

ABO[RT] taskname [/keyword]

Keywords: PMD

Terminates execution of the specified task. All users can generate a Postmortem Dump with the PMD keyword.

ACT[IVE] [/keyword]

Keywords: ALL

Displays on the terminal all tasks issued from that terminal or all tasks active in the system.

ALL[OCATE] ddn: [keyword]

Keywords: = LLnn
TERM = TTnn

Establishes the specified device as the user's private device on multiuser protection systems. Privileged users can allocate a device to any terminal, using the TERM keyword, but nonprivileged users can only allocate devices to their own terminals. The LLnn: keyword allows the user to equate a physical device with a logical device. Specifying the command with only a 2-character abbreviation allocates the first free device of the type specified.

Assign ASN ppn = LLnn: [/keyword]

Keywords: GBL
LOGIN
TERM = TTnn

Defines, displays, or deletes logical device assignments as follows.

Local assign operations

ASN ppn: = LLnn
ASN ppn: = LLnn:/TERM = TTnn: (P)

Login assign operations

ASN ppn: = LLnn:/LOGIN (P)
ASN ppn: = LLnn:/LOGIN/TERM = TTnn: (P)

Global assign operations

ASN ppn: = LLnn:/GBL (P)

Local display operations

ASN

Login display operations

ASN /TERM = TTnn: (P)

Global display operations

ASN /GBL (P)

Local delete operations

ASN =
ASN = LLnn:

Login delete operations

ASN =/LOGIN (P)
ASN =/LOGIN/TERM = TTnn: (P)
ASN =/TERM = TTnn: (P)
ASN =/LLnn:/TERM = TTnn: (P)

Global delete operations

ASN =/GBL
ASN = LLnn:/GBL (P)

Active Task List ATL [taskname]

Displays on the entering terminal the name and status of all active tasks in the system or the status of the particular task specified.

BRO[ADCAST] TTn:message
BRO[ADCAST] -g:filespec
BRO[ADCAST] ALL:message(P)
BRO[ADCAST] LOG:message(P)

Displays the specified message at one terminal for a nonprivileged command or at a number of terminals for a privileged command.

BYE

Logs the user off a multiuser protection system.

CAN[CEL] taskname

Cancels time-based initiation of a task. Privileged users can cancel any task but nonprivileged users can only cancel tasks that they initiated.

DEA[LLOCATE] [ddn:]

Releases a private (allocated) device when ddn: is the device name and unit number. Privileged users can deallocate any device but nonprivileged users can only deallocate devices that they have allocated. If no device is specified, the command deallocates all of the user's allocated devices.

DEV[ICES] [/keyword]

Keywords: dd
/LOG

Displays symbolic names of all devices or of all devices of a particular type known to the system. The LOG keyword displays logged-in terminals.

Dismount DMO ddn[:][label][/keywords]

Keywords: DEV(P)
USER
TERM = TTnn(P)

Tells the file system to mark the volume for dismount and to release the control blocks. Privileged users can dismount any volume, but nonprivileged users can only dismount devices that they have mounted.

Group Global Event Flags FLAGS[/cpg][/keyword]

Keywords: CRE
ELIM

For privileged users, creates or eliminates global event flags for any group. For nonprivileged users, creates or eliminates group global flags only for their own group.

HEL[LO] UIC/PASSWORD

Logs a user on a terminal to access a multiuser system.

HELP [qualifier][qualifier 2][... qualifier 9]
HELP %[qualifier][qualifier 2][... qualifier 9]

Displays the contents of [1,2]HELP.TXT or the contents of a user help file on the issuing terminal.

INI[TVOLUME] ddn volumelabel [/keywords]

Keywords: BAD = [option]
CHA = [characteristics]
DENS = density selection
EXT = block-count
FPRO = [system,owner,group,world]
INDX = index-file-position
INF = initial-index-file size
LRU = directory-preaccess-count
MAP = map count
PRO = [system,owner,group,world]
UIC = [group,member]
WIN = retrieval pointer-count
VI = keyword list

Produces a Files-11 volume on disk, magnetic tape, or DECtape. On multiuser protection systems, users can only initialize volumes on devices that they allocated.

LOGICAL UNIT NUMBERS LUN[S] [taskname]

Displays at the entering terminal the static LUN assignments for a specified task.

MOUNT

Files-11 disk or DECtape format

MOU[NT] ddn:[volumelabel][/keyword(s)]

Keywords:
- ACP = taskname
- EXT = block-count
- FOR = acpname
- FPRO = [system,owner,group,world]
- LRU = FCB-count
- OVR (override)
- PARM = user parameters
- UIC = [uic]
- UNL
- VI (volume information)
- WIN = retrieval-pointer-count

Creates the Volume Control Block (VCB) and declares the volume logically online for access by a file system.

Files-11 (ANSI) magnetic tape format

MOU[NT] device[s]:volumeset:[/keyword(s)]

Keywords:
- ACP = taskname
- BYPASS
- DENS = tape-density
- FPRO = [system,owner,group,world]
- NOLAB
- OVR (override)
- OVRSID
- OVREAP
- UIC = [uic]/VI
- VI (volume information)

Allocates the Volume Set Control Block (VSCB) and mounted units for mounted volume sets. Devices are specified as device[n], and volumes are specified as volumeset.

PARTITION DEFINITIONS PAR[TITIONS]

Displays on the entering terminal's description of each memory partition in the system.

RES[UME] taskname

Allows nonprivileged users to continue execution of a suspended task that was initiated from the entering terminal. Privileged users can direct the Resume command to any suspended task.

RUN taskname [/RSI = magu][/UIC = [uic]]
RUN taskname dtime [/RSI = magu][/UIC = [uic]]
RUN taskname sync [dtime][/RSI = magu][/UIC = [uic]]
RUN taskname atime [/RSI = magu][/UIC[uic]]
RUN [ddn:][fs] filespec [/keyword(s)]

Keywords:
- CKP = option
- INC = size
- PAR = pname
- PRI = number
- SLV = option
- TASK = taskname
- UIC = [g,m]

Initiates execution of a task, either immediately or at one of several time-dependent intervals.

SET /keyword = value

Keywords:
- BUF = dev:[size]
- CRT[= TTnn:]
- EBC[= TTnn:]
- ECHO[= TTnn:]
- ESCSEQ[= TTnn:]
- FDX[= TTnn:]
- FORMFEED[= TTnn]
- HFILL = TTnn:[value]
- HMT[= TTnn:]
- HOLD[= TTnn:]
- LA30S[= TTnn:]
- LINES = TTnn:[value]
- LOGON (P)
- LOWER = [TTnn:]
- MAIN (P)
- MAXEAT (P)
- MAXPKT (P)
- POOL
- PRIV[= TTnn:]
- PUB[= dev:]
- REMOTE[= TTnn:]
- RPA[= TTnn]
- SLAVE[= TTnn:]
- SPEED = TTnn:[recv,xmt]
- SYSUIC[= uic] (P)
- TERM = TTnn:[value]
- TYPEAHEAD[= TTnn:]
- UIC[= uic [TTnn]]
- UIC[= TTnn:]
- VFILL[= TTnn:]
- VT05B
- WCHK[= dev]
- WRAP[= TTnn]

Establishes device characteristics for the device specified. Privileged users can alter device characteristics for all of the devices on the system, but nonprivileged users can only alter and observe characteristics for devices allocated to them.

22

Stop STP *Keyword*

Keywords: taskname
TERM = TTnn

Declares that the task specified is no longer eligible to execute or compete for memory resources.

Task List TKL [*taskname*]

Displays the names and status of all tasks installed on the system or of tasks of a particular task name.

Task Set TKS

Describes each task installed on the system.

Time TIM [*dd-mon-yy*][*hh:mm:ss*]
[*mon/dd/yr*]

For privileged users, sets and displays the date and time for the system. For nonprivileged users, only displays them.

User File Directory UFD *ddn*[*:rvla:bof*][*g,m*][*/keyword(s)*]

Keywords: ALLOC = *num-entries*
PRO = [*s,o,g,w*]

Creates a User File Directory (UFD) on a Files-11 volume and enters its name onto the Master File Directory (MFD). Privileged users can create UFDs on any volume, but nonprivileged users can create UFDs only on a volume mounted on a device that they have allocated.

UNS[TOP] [*/keyword*]

Keywords: *taskname*
TERM = TTnn

Continues execution of a previously stopped task.

Privileged MCR Commands

Allocate Checkpoint Space ACS *ddn*/BLKS = *n*

Allocates or deallocates a checkpoint file on disk for systems that support the dynamic allocation of checkpoint space.

23

ALT[ER] [*taskname*][*/keyword*]

Keywords: PRI = running and static priority
RPRI = running priority only

Changes the static or running priority of an installed task.

BOO[T] [*filespec*]

Bootstraps a [illegible] that exists on a task image file on a Files-11 volume.

Breakpoint to XDT BRK

Passes control to the Extended Debugging Tool (XDT).

CLO[QUEUE]

Displays on the entering terminal information about tasks currently in the clock queue, or about tasks activated by either a RUN command or a .RUNS directive that specified a time-based option.

Fix-in-Memory FIX *taskname*

Loads and locks a task into its partition.

INS[TALL] [*S*][*filespec*][*/keyword(s)*]

Keywords: CKP = option
EST = option
INC = size
PAR = pname
PUD = option
PRI = number
SLV = option
TASK = *taskname*
UIC = [*g,m*]

Makes a specified task known to the system.

LOA[D] *dd*[*/keywords*]

Keywords: PAR = *pname*
SIZE = *partsize*
HIGH

Reads a nonresident device driver into memory and constructs the linkages required to allow access to the device.

24

Open Register OPE[N] [*mem-addr*] + / − *n*][*/keyword*][*mem-addr*]
[*constants-addr*][*=value*] < *line-terminator* >

Keywords: TASK = *taskname*
PAR = *partitionname*
=NL
DRV = *dd*

Allows examination and optional modification of a word of memory.

REA[SSIGN] *taskname lun ddn:*

Reassigns a task's static logical unit numbers from one physical device to another.

RED[IRECT] *nddn: = oddn:*

Redirects all I/O requests from one physical device unit to another (from a to a).

REM[OVE] *taskname*

Deletes an entry (task name) from the System Task Directory (STD) and thereby removes the task from the system.

SAV[E] [*keywords*]

Keywords: WB
MOU = "*string*"
SFILE = "*string*"

Copies the current system image into the system image file from which the current system was booted.

System Service Message SSM *message-text*

Inserts text into the error log reports.

UNF[IX] *taskname* [*keyword*]

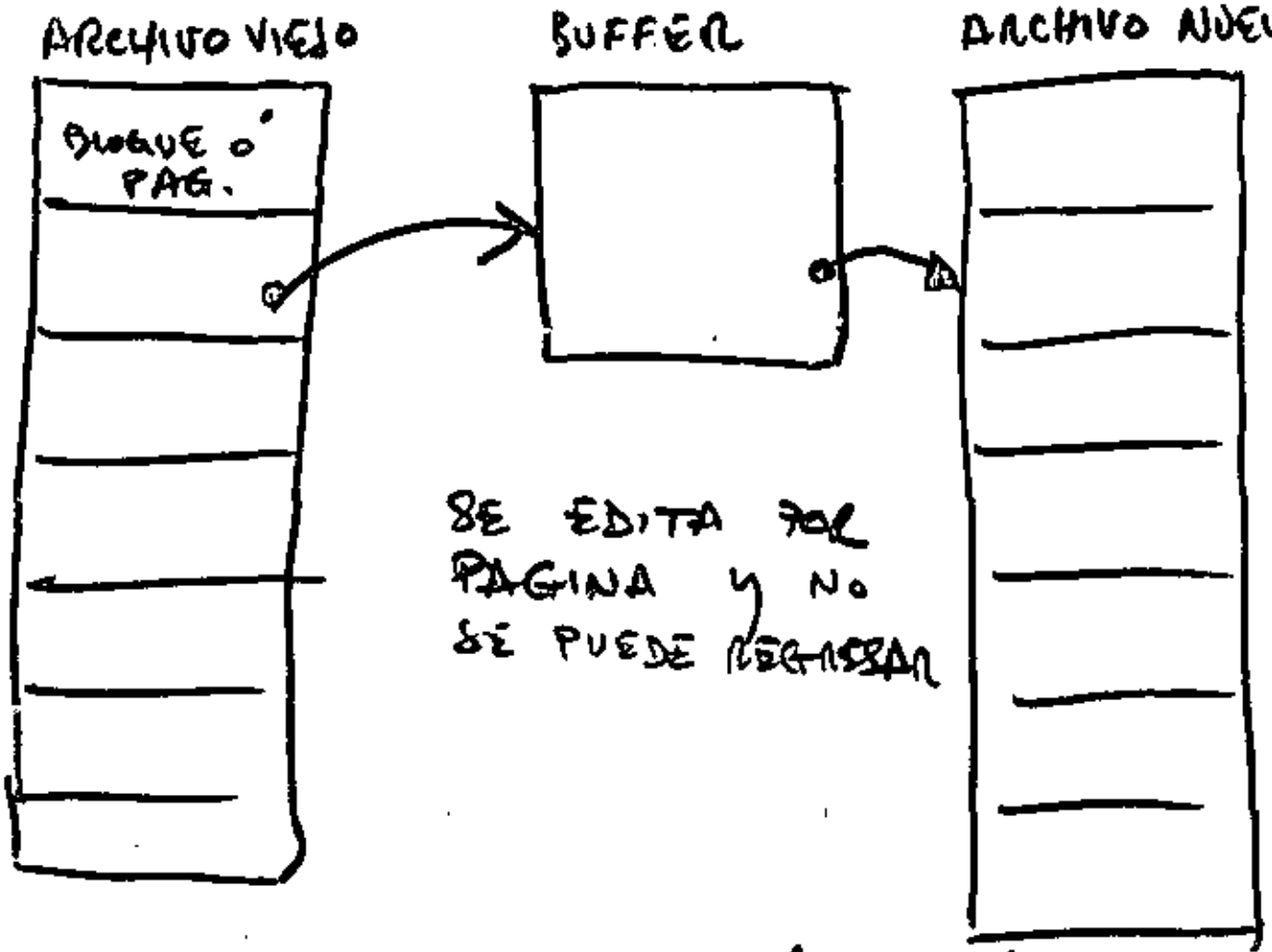
Keywords: RG
RI

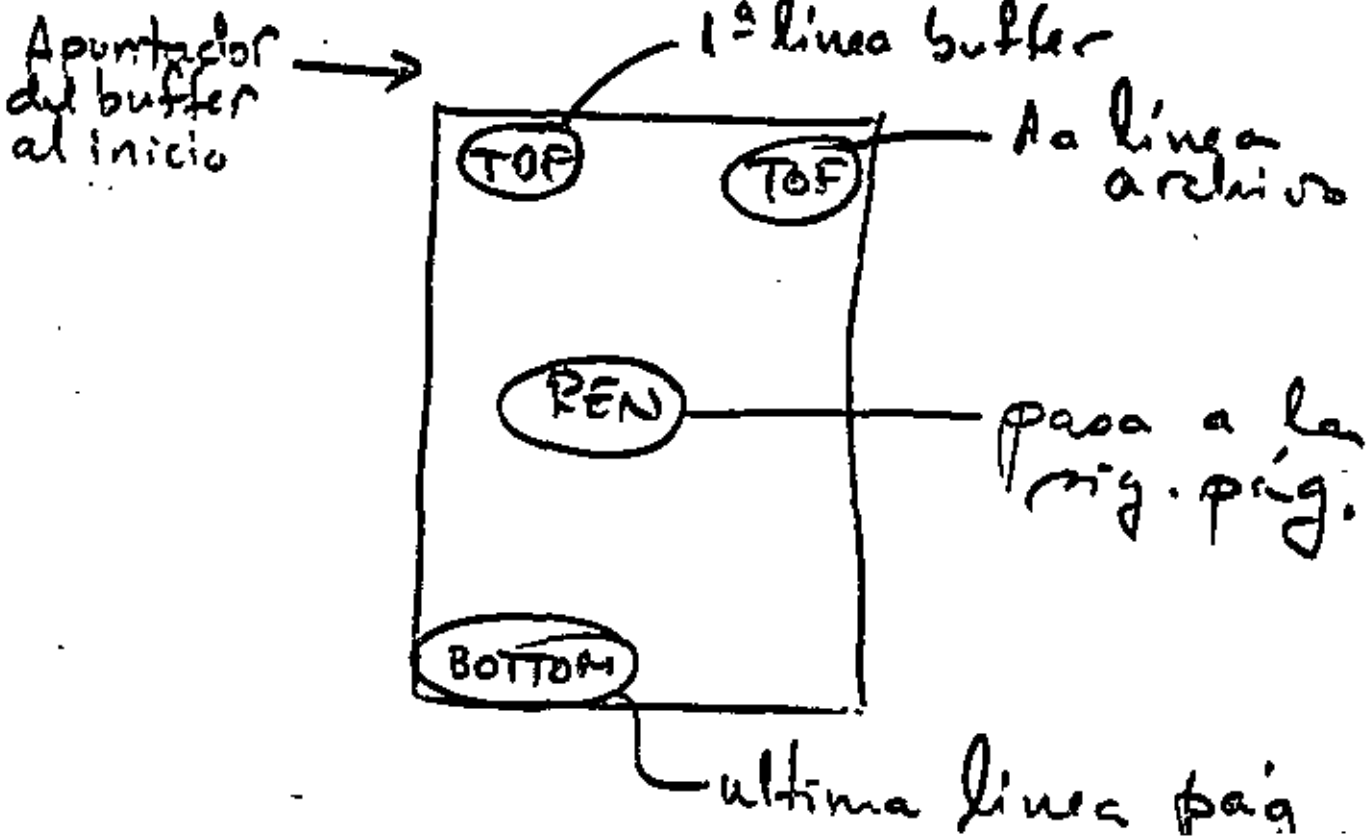
Frees a fixed task from memory.

UNL[OAD] *dd:*

Removes a loadable device driver from memory.

EDI


ARCHIVO VIEJO
BUFFER
ARCHIVO NUEVO
BLOQUE o PAG.
SE EDITA POR PAGINA Y NO SE PUEDE REGRESAR


Apuntador del buffer al inicio
1ª línea buffer
TOF
TOF
1a línea archivo
REN
pasa a la sig. pág.
BOTTOM
ultima linea pág

- Creación y/o modif. de archivos

```
>EDI (CR)
EDI> nombre.tipo
[CREATING NEW FILE]
INPUT
(
  instruc.
  datos
END (CR)
(CR)
*EXIT
[EXIT]
>
```

```
[..... LINES READ IN]
[PAGE 0]
*
```

- TIPOS DE COMANDOS

Localizar y cambiar Texto:

- Find
- Locate — *L ENTER
- Change /S1/S2/ — *C/E...R/TY
- Next
- Print
- Top
- (CR)
- NP
- PAGE n
- SC /S1/S2/
- PASTE /S/S1/

Insertar y Borrar
- Insert
- Add
- Delete [n]
- (ESC) apunta a e imp. línea ant.
- R type nemph.
- List listar buffer pointer no se mueve

Definición de modo
- BL ON (OFF)
- OP sec. file
- SP
- SS
- SIZE n #líneas/buff
- TA ON [OFF]

EJEMPLO Lectura de tarjetas en EDI

```
>EDI   nombre.FTN
*OPEN cRø
*SS
*SIZE 50
*READ
    | leer y editar
*WRITE
*READ
    |
*ES
*EXIT
```

```
>EDI ADD.FTN <CR>
(CREATING NEW FILE)
INPUT:
        TYPE 1 <CR>
1       FORMAT (' ENTER TWO NUMBERS - N,N') <CR>
        ACCEPT(SUPPRESSECHO) I,K,L <CTRL/R>
        ACCEPT I,K,L <CR>
2       FORMAT (2I2,I3) <CR>
        PRINT *,D
        TYPE I,K,L <CR>
3       FORMAT (' THE SUM IS ',I5) <CR>
        STOP <CR>
        END <CR>
<CR>
^EX <CR>
(EXIT)
```

(25)

(29)

LINE TEXT EDITOR (EDI) COMMANDS

In this section, the following conventions are used:

The asterisk (*) can be used in place of any number in an EDI command. It is read as 32,767.

An ellipsis (...) can be used in many search strings to identify characters between the first and last characters of the string.

EDI allows the use of abbreviations in commands. Square brackets enclose optional command text.

ADD string A string

Adds the character string indicated to the end of the current line.

ADD AND PRINT AP string

Adds the character string indicated to the end of the current line and prints the entire line on the terminal.

BEGIN B[EGIN]

Sets the current line to the line preceding the top line in the file or block buffer. This command creates a copy of the file when it is invoked in Line Mode.

BLOCK ON/OFF BL[OCK][ON] or
BL[OCK][OFF]

Changes to and from the EDI Block Mode of accessing text.

BOTTOM BO[TTOM]

Moves the line pointer to the bottom of the current block (in Block Mode) or to the bottom of the file (in Line Mode).

CHANGE [n]C[HANGE] /string1/string2/[]

Replaces string 1 with string 2 in the current line n times.

CLOSE CL[OSE] [filespec]

Transfers the remaining lines in the block buffer and input file to the output file and closes all files.

30

CLOSE AND DELETE CDL [filespec]

Transfers the remaining lines in the block buffer and the input files to the output file, closes the output file, and deletes the input file.

CLOSE SECONDARY CLOSES

Closes the secondary input file.

CONCATENATION CHARACTER CC [letter]

Changes the concatenation character to the character specified. (The default is &.)

CTRL/Z (CTRL/Z)

Closes files and terminates the editing session.

DELETE D[ELETE] [n] or
D[ELETE] [-n]

Deletes the current line(s) as specified above and n-1 lines if n is a positive number. Deletes n lines preceding the current line if n is a negative number. Negative numbers can only be used in Block Mode.

DELETE AND PRINT DP [n] or DP [-n]

Deletes the current line and prints the new current line.

END E[ND]

Sets the last line in a file or block buffer as the block buffer current line.

ERASE ERASE [n]

Erases the current line in Line Mode.

ESCAPE (ESC)
or
ALTMODE (ALT)

Prints the previous line and makes it the new current line.

EXIT EXIT [filespec]*

Transfers the remaining lines in the block buffer and input file to the output file. Closes files, renames the output file if specified, and terminates the editing session.

31

EXIT AND DELETE ED [filespec]

Transfers the remaining lines in the block buffer and input file to the output file, closes files, renames the output file, and deletes the input file.

FILE FIL[E] filespec

Transfers lines from the input file to both the output file and the specified file until a form feed or end-of-file is encountered. This command is only legal in Line Mode.

FIND [n]F[IND] string

Searches the current block or input file, beginning at the line following the current line, for the nth occurrence of the string specified. (If n is not specified, EDI searches for the next occurrence of the string.) Sets the line pointer to the line it finds. A string must begin in the first column of the line to be a match.

FORM FEED FF

Inserts a form feed into the block buffer.

INSERT IN[SERT] [string]

Enters the specified string immediately following the current line. If no string is specified, EDI enters Input Mode.

KILL KILL

Closes the input and output files and deletes the output file.

LINE CHANGE [nK]C/string1/string2/[1]

Changes all occurrences of string 1 in the current line (and n-1 lines) to string 2.

LIST ON TERMINAL LI[ST]

Prints on the terminal all of the lines remaining in the block buffer or input file.

LOCATE [n]L[OCATE] string

Locates the nth occurrence of the specified string. In Block Mode, the search stops at the end of the current block.

MACRO MACRO [x] definition

Defines the macro number x for the EDI commands in the definition.

MACRO CALL MC[CALL][n]

Retrieves the macro definition stored in the file MCALL.Ln

MACRO EXECUTE [n]Mx [a]

Executes macro x n times, while passing numeric argument a. The value x can be 1, 2, or 3.

MACRO IMMEDIATE [n] < definition >

Defines and executes a macro n times. Stores it as macro number 1.

NEXT N[EXT] [n] or
N[EXT] [-n]

Establishes a new current line n lines away from the current line.

NEXT AND PRINT NP[n] or
NP[-n]

Establishes and prints a new current line

OPEN SECONDARY OP[EN'S] filespec

Opens the specified secondary file.

OUTPUT ON/OFF OU[TPUT] ON or
OU[TPUT] OFF

Enables or disables a file transfer to an output file in Line Mode

OVERLAY O[VERLAY] [n]

Deletes n lines, enters Input Mode, and inserts new lines, as typed, in place of the deleted lines.

PAGE PAG[E] n or -n

Enters Block Mode. Reads page n into current block buffer. If n is less than the current page, P.[0] goes to the top of the file first. Pages are set by form feed characters

PAGE FIND [n]PF[IND] string

Searches successive blocks for the nth occurrence of the string. A string must begin in the first column of the line to be a match.

PAGE LOCATE [n]PL[OCATE] string

Searches successive blocks for the nth occurrence of the string

PASTE PA[STE] /string1/string2[/]

Searches all remaining lines in the file or block buffer that contain string 1 and replaces them with string 2

PRINT P[RINT] [n]

Prints the current line and the next n-1 lines on the user's terminal. The last line printed becomes the new current line.

READ REA[D] n

Reads the next n blocks of text into the block buffer. If the buffer already contains text, the new text is appended to it.

RENEW REN[EW] [n]

Writes the current block to an output file and reads a new block from an input file (Block Mode only).

RETURN RT.

Prints the next line on the terminal and makes it the new current line. This command also exits from Input Mode if it is typed as the first character of a line

RETYPE R[ETYPE] string

Replaces the current line with the specified string or deletes the current line if no string is specified

SAVE SA[VE] [n] [filespec]

Saves the current line and the next n-1 lines in the specified file.

SEARCH AND CHANGE SC/string1/string2[/]

Locates string 1 and replaces it with string 2.

SELECT PRIMARY SP

Reestablishes the primary file as an input file.

SELECT SECONDARY SS

Selects an open secondary file as an input file

SIZE SIZE n

Specifies the maximum number of lines that can be read into a block buffer.

TAB TA[B] ON or
TA[B] OFF

Turns automatic tabbing on or off

TOP T[OP]

Moves the line pointer to the line preceding the top line of the current block in Block Mode or to the top of the file in Line Mode. The TOP command creates a new version of the file each time it is executed in Line Mode.

TOP OF FILE TOF

Returns to the top of the input file and saves all of the previously edited pages. This command creates a new version of the file each time it is executed in Line Mode.

TYPE TY[PE] [n]

Prints the next n lines on the terminal. This command is identical to the PRINT command in Line Mode. However, in Block Mode, the line pointer remains at the current line unless EDI reached the end of a block.

UNSAVE UNS[AVE] [filespec]

Inserts all lines from the specified file following the current line. If no file name is used, EDI looks for a file called SAVE.TMP.

UPPER CASE UC ON or
UC OFF

Enables or disables conversion of lowercase letters to uppercase letters when they are entered at a terminal.

VERIFY V[ERIFY] ON or
V[ERIFY] OFF

Selects whether the operation of the LOCATE and CHANGE commands will be verified (printed on the terminal) after the line is located or changed.

WRITE W[RITE]

Writes the contents of the block buffer to an output file and erases the block buffer.

DEC STANDARD EDITOR (EDT) COMMANDS

C[HANGE] [range] [/NL]

Invokes EDT Character Editing Mode, which can be used only on a video display terminal. (The terminal must also be set to CRT to permit the use of Character Mode.)

Character Mode allows the following subcommands which, except for Exit, work the same as they do in Command Mode:

D[ELETE] [range] [/keyword]

Deletes the specified text from a buffer.

E[XIT] [/keyword]

Terminates Character Mode and returns to Command Mode.

I[NSERT] [range] [/keyword]

Inserts text into the buffer.

QUIT

Terminates EDT operation without creating or modifying any files.

R[EPLACE] [range] [/keyword]

Removes specified text and goes into Insert Mode so replacement text can be entered.

S[UBSTITUTE/string1/string2]

Replaces the first string of characters with the second.

CTRL/Z (CTRL/Z)

Terminates an insert. Returns to Character or Command Mode.

CO[PY] range-1 %TO range-2 [/keyword]

Keywords QUERY
$EQuence:(initnum:incr
UNsequenced

Transfers lines from one location to another, at the same time retaining the lines in their original location.

D[ELETE] [range] [keyword]

Keywords: QUERY

Deletes the specified lines from the buffer being edited.

EX[IT] [keyword]

Keywords: RE[NAME] filename
SEQ[UENCE],initnum,incr
UN[SEQUENCED]

Terminates EDT operation and saves the contents of the Main Text Buffer.

FIND range

Moves the line pointer to the beginning of the specified line.

INC[LUDE] [range] /FI:filename [range] [keyword]

Keywords: [SEQ[UENCE],initnum,incr

Locates a file and copies it into the specified text buffer at the specified range.

I[NSERT] [range] [keyword]

Keywords: SEQ[UENCE],initnum,incr
UN[SEQUENCED]

Places the text that is typed at the terminal into a text buffer.

M[OVE] range 1 %TO range 2 [keyword]

Keywords: Q[UERY]
SEQ[UENCE]:initnum,incr

Transfers lines from one location to another and deletes them from the original location.

P[RINT] [range] /FI:filename

Creates a file from the contents of a text buffer. The new file includes as part of the text the EDT line numbers in the original buffer.

QUIT

Terminates EDT and does not save the contents of any text buffer.

R[EPLACE] [range] [keyword]

Keywords: SEQ[UENCE] initnum,incr
UN[SEQUENCED]

Deletes lines specified and enters Insert Mode so that the lines can be replaced.

RES[EQUENCE] [range] [keyword]

Keywords: SEQ[UENCE],initnum,incr
UN[SEQUENCED]

Assigns new line numbers to the lines in the current text buffer.

REST[ORE]/FI:filename

Locates the file created by an EDT Save command and uses the file to recreate the status of all files and text buffers as they were preserved with the EDT Save command.

SA[VE]/FI:filename

Preserves the contents of text buffers and the status of all files used during an EDT editing session in a file which the user names using the form filename.SAVE.

SET
- CA[SE] [keyword]: UPPER | LOWER | NONE
- EXACT [keyword]: CASE | NONE
- TE[RMINAL] [keyword]: HCPY | VTDEB | V150 | VT52 | VT55 | VT61 | LA30 | LA36

Establishes criteria used by other EDT commands to flag upper- or lowercase characters and to set parameters for the terminal being used.

SHOW [keyword]

Keywords: BU[FFERS]
CA[SE]
EXACT
TE[RMINAL]
VE[RSION]

Displays buffer and software version information and options established by the SET command.

S[UBSTITUTE]:string1/string2/[range][keyword]

Keywords: BE[IEF]
Q[UERY]
-T[YPE]

Changes characters within lines of the text buffer.

S[UBSTITUTE] Next

Repeats the substitution carried out in the SUBSTITUTE command if follows.

T[YPE] range

Displays lines of text at the terminal.

WR[ITE] [range][keyword]

Keywords: FI:filename
SEQ[UENCE],initnum,incr
UN[SEQUENCED]

Creates a file from the contents of a text buffer.

XEQ range

Executes a sequence of EDT commands that were previously entered and stored in a buffer.

COMPILADOR { FOR. MAC

FORMATO

> compilador objeto [, listado] = fuente

> compilador , listado = fuente

EJEMPLO

> FOR XXX, XXX = XXX

> MAC DK1:XXX.OBJ, TI: = DK1:XXX.MAC

Si EL COMPILADOR NO ES RESIDENTE

> RUN $FOR (MAC)

FOR > ....

FOR > ↑Z

TKB

FORMATO SIMPLE

≥TKB ejecutable [, mapa, símbolo] = objeto [, ... objeto$_n$]

FORMATO COMPUESTO : + de 6 archivos
+ 4 LUN
+ 32 K

≥TKB (CR)
TKB> ejecutable [, mapa, símbolo] = objeto$_1$ (CR)
TKB> objeto$_2$, ..., objeto$_n$ (CR)
TKB> / (CR)
ENTER OPTIONS
TKB> opción$_1$ = valor ! opción$_2$ = valor (CR)
TKB> opción$_3$ = valor (CR)
TKB> //

SWITCHES QUE PUEDE TENER EL EJECUTABLE

/PR — priv.
/CP — checkp
/CR — Xref.
/EA
/FP
/MM
/NT

- control { ABORT=n. ignorar n líneas de comando
- Identif. { PAR=nombre
  PRI=n.
  TASK=nombre
  UIC=[g,m]
- Ubicación { ACTFIL=n. # arch. abiertos simu.
  FMT BUF=n. Formato buffer term.
  STACK=n. área trabajo
- Uso de recursos comunes { COMMON = nombre área
  LIBR = [g,m] nombre
  RESCOM = [g,m]
  RESLIB = [g,m]
- Distos E/S { ASGN dispositivo: unidad
- Alterar (perdiar) macro { ABSPAT
  GLB DEF
  GLBPAT
- Sincronización { ODTV
  TSKV } direcc. virtual

Ejemplo >TKB
TKB> A/CP,B=C,D/LB,[1,2]E
TKB>/
ENTER OPTIONS
TKB > ASGN CRØ:5
TKB > PAR=BIGPAR
TKB > PRI=100.
TKB>//

TASK BUILDER (TKB) SWITCHES AND OPTIONS

In this section, red type indicates user input.

The format for Task Builder commands is:

```
>TKB
TKB>task image file,memallocfile,symdeffile=inputfile(s)
```

For example, to task build a program called Zebra, type:

```
>TKB
TKB>ZEBRA.TSK,ZEBRA.MAP,ZEBRA.STB=ZEBRA.OBJ
TKB>/
ENTER OPTIONS:
TKB>option name=argument(s)
...
TKB>//(to end Task Builder operation)
or
TKB>/(if you have another task to build)
```

The Task Builder file specification is:

```
filespec=dev:[g,m]filename.typ;version/switch(es)
defaults=SY:[uic]filename.typ;*/switch
```

The Task Builder uses the following default file types for the files named:

Task Image File	.TSK
Memory Allocation File	.MAP
Symbol Definition File	.STB
Object Module	.OBJ
Overlay Description	.ODL
Indirect Command	.CMD
Object Module Library	.OLB

In the file specification above, * is the latest version number for an input file and the latest version plus 1 for an output file.

Switches

The following key is used in the description below to designate which input and output files can use the Task Builder switch specified.

[T]	Task Image	(.TSK)
[M]	Task Builder Map	(.MAP)
[S]	Symbol Definitions	(.STB)
[I]	Input	(.OBJ, .OLB, .ODL, .CMD)

The default value for switches is negative (-sw) unless otherwise specified.

/ACn

Specifies that the task is an Auxiliary Control Processor (ACP). n specifies the base relocation register (allowable registers are 0, 4, or 5, default register is 5). Overrides /PR if applied to the same file. [T]

/AL

Makes the task image file checkpointable and allocates checkpoint space in the checkpoint file. (Do not use with /CP in the same command line.) [T]

/CC

Specifies that the input file contains more than one object module. /-CC task builds only the first object module. The LB: (library) switch overrides /CC if it is applied to the same file. (Default is /-CC.) [T]

/CM

Specifies a compatability mode resident overlay structure. (Overlay segments are aligned on 256-word physical boundaries.) [T]

/CP

Makes the task image checkpointable and allows the task to be checkpointed to system checkpoint space. (Do not use in the same command line with /AL.) [T]

/CR

Appends a global cross-reference listing to the memory-allocation file. [M]

/DA

Includes a debugging aid in the task image (ODT) for a task-image output file or a user-supplied debugging program (for an input file). [T, I]

/DL

Specifies a default library file for global references that remain undefined after user-specified library files have been searched. (Can be applied to only one input file per task.) [I]

/EA

Specifies that the task uses the extended arithmetic elements. (/FP overrides /EA if applied to the same file.) [T]

/FP

Specifies that the task uses the floating-point processor. (Overrides /EA if applied to the same file.) [T]

/FU

Specifies a full search of all co-tree segments for a matching definition or reference when processing modules from the default object module library. [T]

/HD

Includes a header in the task image. (Default is /HD; /-HD is used with common blocks, resident libraries, loadable drivers, and system images.) [T,S]

/LB

Without arguments: TKB uses the input file as a library of relocatable object modules and searches to resolve undefined global references. Includes in task image any modules found in the library that resolve the undefined references. [I]

With arguments: [/LB:mod-1:mod-2:mod-3 . . . .] TKB inserts only the modules named in the command into the task image. [I]

/MA

Includes information from the input file in the memory allocation listing (when applied to an input file) or controls the display of information about the default library and shared regions (when applied to a memory allocation file). (Default is /MA for input file or /-MA for a memory allocation file.) [M, I]

/MM

Specifies that the system has memory management hardware. (Defaults to /MM if host system has memory management, or to /-MM if it does not.) [T]

/MP

Specifies that the input file describes the task's overlay (tree) structure. [I]

/PI

Specifies that only position-independent code or data is in the shareable global area. [T, S]

/PM

Produces a Postmortem Dump if the task is terminated with an SST abort. [T]

/PR:n

Specifies that the task has privileged access. /AC overrides /PR:n if applied to the same file. n specifies base relocation register (0, 4, or 6; default is 6). [T]

/RO

Enables recognition of the memory-resident overlay operator (!) in the overlay descriptor file (/MP). (Default is /RO.) [T]

/SE

Specifies that the task can receive messages by means of the Executive SEND directive. (Default is /SE.) [T]

/SH

Produces a short form of the memory-allocation file without the file contents section. [M]

/SL

Specifies that the task is slaved to an initiating task. Slave task runs under the UIC and TI: of the sending task. (Applies only to systems with multiuser protection.) [T]

/SP

Lists the memory-allocation file on the printer via the spooler. (Default is /SP.) [M]

/ST

46

/SQ

Builds program sections in the task image in the order in which they are named, rather than in alphabetical order. (Cannot be used with FORTRAN I/O handling modules or FCS modules from SYSLIB.) [T]

/SS

Extracts a global symbol definition from the input file if the global symbol table has a matching undefined reference. [I]

/TR

Specifies that the task can be traced. [T]

/WI

Lists the memory-allocation file in 132-column (wide) format. (Default is /WI:) [M]

/XT:n

Terminates the building of the task after n error diagnostics are detected; n can be octal or decimal (decimal must be specified with a decimal point, for example, 8.). [T]

Options

[H]

Option is of interest to high-level language programmers.

[M]

Option is of interest to MACRO-11 programmers.

[H,M]

Option is of interest to both high-level language and MACRO programmers.

Names used for option input can be 6 characters long, from the Radix-50 character set (A–Z, 0–9, and $).

ABORT = n

Terminates the current task-build operation and restarts the Task Builder for another. (The n satisfies the option syntax; it means nothing.) [H, M]

43

ABSPAT = segname:address:value1,...value8

Patches the task image from its absolute address through 8 words. [M]

ACTFIL = [lenmax (decimal integer)

Specifies the number of files that a task can have open simultaneously (the default is 4). [H]

ASG = devicename:unit:...:unit

Assigns logical unit number(s) in decimal to specified physical device(s). [H, M]

COMMON = name,access-code[:apr]

Declares that the task will access a system-owned resident common area. [H,M]

CMPRT = name

Identifies the completion routine in a supervisor-mode library. [M]

EXTSCT = psect:name,extension

If the program section has the concatenated attribute, this option extends the size of the named program section by the number of octal bytes specified in the extension. If the program section has the overlay attribute, it is extended only if the extension value exceeds the length of the section. [H, M]

EXTTSK = n

Extends the task memory allocation by the length n (in decimal words) when it is installed in a system-controlled partition. The extension is rounded to the closest 32-word boundary. The default is the extension of the total task size as specified by the PAR option length parameter. [H, M]

FMTBUF = max-format (decimal integer)

Specifies the number of characters (in decimal bytes) in the longest format specification to be compiled at run time. The default is 132. [M]

GBLDEF = symbol-name:symbol-value

Defines the named global symbol as having a value in the range of 0 through 177777 (octal). [M]

GBLPAT = segname:symname[+/-offset][:val1,...:val8

Patches the task image from the location addressed by the global symbol plus or minus the octal offset value through 8 words. All values are octal. [M]

GBLREF = symbol-name,symbol-value

Declares the named symbol as a global symbol reference originating in the root segment of the task. [H, M]

GBLXCL = symbolname:symbolname:...:symbolname

Specifies the symbols that are to be excluded from the symbol definition file of a resident supervisor-mode library. [H, M]

LIBR = name:access-code[:apr]

Declares that the task will access a system-owned resident library. [H, M]

MAXBUF = max-record

Specifies the maximum allowable record buffer size (in decimal bytes) in any file processed by the task. [M]

ODTV = symbol-name,vector-length

Declares the named global symbol to be the address of the ODT synchronous system trap vector (SST). The global symbol must be defined in the main root segment. [M]

PAR = name[:base:length]

Identifies the partition for which the task is built. For a mapped system, a size of 0 implies a system-controlled partition, and a nonzero size implies a user-controlled partition. Base and length do not have to be expressed if the partition resides on the host system. The default is PAR = GEN. [H, M]

PRI = priority

Sets the priority at which the task executes; can be overridden when the task is installed. The priority is a decimal integer between 1 and 250. [H, M]

RESCOM = [lgroup/]access-code[:apr]

Declares that the task will access a user-owned resident common. [H, M]

RESLIB = [lgroup/]access-code[:apr]

Declares that the task will access a user-owned resident library. [H, M]

RESSUP = [lgroup/][SV]:apr]

Specifies that the task will access a resident supervisor-mode library. [H, M]

ROPAR = partname

Specifies the partition in which the read-only portion of a multiuser task resides. [H, M]

STACK = stack-size

Establishes the maximum size of the stack available to the task. The default is 256 bytes. [H, M]

SUPLIB = name[:[SV]:apr]

Specifies that the task will access a system-owned resident supervisor-mode library. [H, M]

TASK = taskname

Names the task. [H, M]

TSKV = symbol-name:vector-length

Declares a global symbol to be the address of the task synchronous system trap vector (SST). [M]

UIC = [g,m]

Declares the UIC for time-based initiation of a task. The default is the UIC under which the Task Builder is running. [H, M]

UNITS = max-units

Declares the number of logical units used by the task in decimal number in the range of 0 through 250. The default is 6. [H, M]

VSECT = psect/name,base:window#[,physical-length]

Specifies the virtual base address, length of virtual memory address space (window), and length of physical memory allocated to the named program section. [H, M]

EJECUCION

> RUN nombre[.tipo ; versión]

nombre= disp : [usr] nombre.tipo;versión

ACTIV. QUE REALIZA
- localiza XXX.TSK
- carga en memoria
- ejecuta
- remueve de memoria

PIP

FORMATOS

PIP > nombre.tipo;versión /switches (arch)

PIP> /switches (direct.)

PIP> salida = entrada (copia)

VALORES DADOS POR DEFAULT

dispositivo = SY:

UIC = con la que se inició la sesión o la última dada

versión = última.

RANGO DE LOS SWITCHES

>PIP arch$_1$, arch$_2$, arch$_3$ /SW ← afecta a los 3

>PIP arch$_1$ /SS, arch$_2$ /SW

↑ afecta al 1: ↑ afecta a todos

CONVENCIONES

* ⇒ todas las versiones, tipos, archivos

nombre archivo de sal. SIEMPRE debe especificarse

cualquier switch en el archivo de salida afecta a los de entrada.

FUNCIONES

1) COPIA DE ARCHIVOS

PIP> Salida = entrada

PIP> Salida = entr$_1$, entr$_2$, entr$_3$

PIP> TT$_n$:= archivo.tipo; v (imprime en TT)

$v = \begin{cases} -1 & \text{versión + antigua} \\ 0 & \text{última versión} \\ * & \text{todas} \end{cases}$

2) CONTROL DE ARCHIVOS (listado direct.)

/FU toda info
/LI nombre fecha creac
/TB bloques usados
/BR breve

PIP> LP:=/LI
↳ asume por default todo lo del VIC.

3) BORRADO DE ARCHIVOS

>PIP [disp:] nombre .tipo; v /DE borrar

>PIP [disp:] nombre . tipo /PU purgar

4) VARIOS

/AP agregar a un arch. ya exist.
/SP Enviar a sprober
/RE Cambiar nomb...

5) PROTECCION

PIP> nombre/PR: No.

PR { SY, QW, GR, WO } [D|E|W|R] { 1 ⇒ prohib. 0 ⇒ perm.

Ejemplos

PIP >/LI
PIP > SAL.DAT = ENT.DAT /RE
PIP> [112,6] SAL.DAT= [12,12]ENT.DAT
PIP> SAL.* /PU
PIP> SAL.*;*/DE
PIP> ENT.FTN /OW = 15

After listing the directory, PIP displays its task prompt and waits for further input. To terminate PIP, type CTRL/Z in response to the prompt

```
PIP>^CTRL/Z>
>
```

The PIP directory listing includes the following information:

(A) The physical device unit on which the files are stored, and the UIC that owns the directory. The unit named is your SY:; the UIC is your current UIC.

(B) The date and time you issued the PIP request.

(C) The name, type, and version number of each file.

(D) The size of each file in blocks. A block is 512 bytes (256 words) long.

(E) The letter C if the file is contiguous, that is, not split into physically separated sectors on the disk.

(F) The date and time each file was created.

(G) The cumulative size in blocks of all the files listed and the number of blocks allocated for all the files.

Figure 3-1 assigns the letters in the above list to the appropriate parts of a directory listing.

```
                    (A)
(B) DIRECTORY DB0:[301,314]
    25-MAY-77 14:48
      (C)                 (D)     (E)     (F)
    ADD.OBJ;2              2.             25-MAY-77 14:31
    ADD.TSK;1             27.          C  20-MAY-77 14:32
    ADD.MAP;1              4.             25-MAY-77 14:32
    ADD.STB;1              3.             25-MAY-77 14:32
    ADD.FTN;1              1.             20-MAY-77 15:26
    ADD.TSK;3             29.          C  25-MAY-77 14:33
    ADD.OBJ;1              2.             20-MAY-77 15:27
    ADD.TSK;2             29.          C  20-MAY-77 15:28

(G) TOTAL OF 99./112. BLOCKS IN 8. FILES
```

```
>PIP TI:=ADD.FTN <CR>

        TYPE 1
1       FORMAT (' ENTER TWO NUMBERS - M,N')
        ACCEPT 2,M,N
2       FORMAT (2I5)
        TYPE 3,M+N
3       FORMAT (' THE SUM IS ',I5)
        STOP
        END
```

PERIPHERAL INTERCHANGE PROGRAM (PIP) COMMANDS

The default PIP operation (with no switches specified) is to copy files, using the following format:

outfile = infile(s)/subswitches

Creates a copy of a file on the same or another volume.

PIP allows the following parameters for this command:

outfile	If file name, file type, or file version is null or specified by wildcards, PIP uses input parameters unless it is overruled by the New Version (/NV) or Successor (/SU) switches. If the file name, file type, or file version is defined, no other field can be a wildcard and only one input file can be specified.
infile	If file name, file type, or file version is null, the default is *.*;*.

subswitches:

/BL:n	Specifies the number of contiguous blocks for the output file, where n is octal or decimal. If n is decimal, it is followed by a period (n.).
/CO, /-CO, or /NOCO	Specifies a contiguous or noncontiguous output file.
/FO	File ownership (output file UFD).
/NV	Forces the output version number of the copied file to be 1 more than the current highest version.
/SU	Copies the output file, superseding the existing output file.

APPEND outfile[/FO] = infile(s)/APP[/FO]

Opens an existing file and appends the input files, infile(s), to the end of it.

PIP allows the following parameters for this command:

outfile	Explicit file name and file type.
infile(s)	Explicit file parameters, wildcard by default.
/FO	File ownership is the output file UFD; without /FO, ownership is the UIC of the user running PIP.

BLOCKSIZE outfile[/BS:n] = infile[/BS:n]

Defines the block size for 7- and 9-track magnetic tape.

CREATION DATE outfile/CD = infile

Gives the output file the creation date of the input file rather than the date of the file transfer. (This switch cannot be used with the Merge switch.)

DEFAULT [ddn:][g,m]/DF

Changes the PIP default device and/or UFD.

DELETE infile(s)/DE[/LI]

Deletes files. /LI is a subswitch that causes PIP to list the files it deletes.

END-OF-FILE infile/EOF[:block:byte]

Specifies the end-of-file pointers for a file. If values for block and byte are not entered, PIP places EOF at the last byte of the last block in the file.

ENTER outfile = infile(s)/EN[/NV]

Enters a synonym for a file in a directory with an option to form the version number of the output file to 1 greater than the latest version for the file.

outfile	The file name, file type, or file version can be explicit, a wildcard, or null. A field that is a wildcard or null assumes a corresponding input field.
infile	Default for the file name, file type, and file version is *.*;*.
/NV	See COPY.

FILE ID /FI:filenumber:sequencenumber

Accesses a file by its file identification number (File ID).

FREE [ddn:] /FR

Prints on the terminal the amount of space available on a volume and the largest block of contiguous space.

IDENTIFICATION /ID

Causes the version number of PIP currently in use to be displayed on the terminal.

LIST [outfile = infile(s)/LI/subswitch]

Lists the contents of one or more User File Directories, with an option to specify formats for output directories.

outfile	Listing file specifier; defaults to TI:.
infile	Default is *.*;*.

The subswitches determine what type of report is displayed.

/LI/BR or /BR	Brief report.
/LI	Limited report.
/LI/FU:n or /FU:n	Full report in specifies the decimal characters per line; the default is device buffer size.
/LI/TB or /TB	Total blocks report.

MERGE [CONCATENATE] outfile = infiles/ME/subswitches]

Creates one file by concatenating two or more files. The fields and subswitches are the same as for the PIP COPY operation.

NO MESSAGE [infile(s)/NM/SW]

Causes certain PIP error messages not to be printed, for example, the message NO SUCH FILE(S). The switches that can be used with the No Message switch are:

/LI	Lists directory.
/DE	Deletes files.
/PU	Purges files.

Any subswitches of these switches can also be used.

PROTECTION symbolic: [infile/PR:symbolic/FO]
numeric: [infile/PR:octal value/FO]

Alters the file protection for the file specified. The file name and file type must be explicit.

Symbolic protection codes assign privilege merely by their presence, using:

System = /SY:RWED
Owner = /OW:RWED
Group = /GR:RWED
World = /WO:RWED

Numeric protection denies privilege by setting bits in a protection status word. Add octal values from the following list to deny privilege.

User Class	Privilege	Octal Code	Bit
System	R	1	0
	W	2	1
	E	4	2
	D	10	3
Owner	R	20	4
	W	40	5
	E	100	6
	D	200	7
Group	R	400	8
	W	1000	9
	E	2000	10
	D	4000	11
World	R	10000	12
	W	20000	13
	E	40000	14
	D	100000	15

PURGE [infile(s) /PU:n/LO]

Deletes a specified range of versions of a file that does not delete the latest version). Specification of a file version number is not necessary. Wildcards are valid for file name and file type.

When :n is specified, PIP deletes all but n latest consecutively numbered versions. Without :n, PIP deletes all but the latest version.

REMOVE [infile(s) /RM]

Removes an entry from a directory file (the opposite of ENTER).

RENAME outfile = infile(s)/RE/NV]

Changes the name of the file specified. Used with the New Version(/NV) switch, RENAME creates an output file with a version number 1 higher than the last version of the file.

outfile	A wildcard (*) or null field assumes the value of the corresponding field in the input file.
infile	Null file name, file type, and file version default to "*.*".
/NV	See COPY.

REWIND outfile(/RW) = infile(/RW)

outfile	Causes the tape on the specified unit to be erased.
infile	Causes the tape on the specified unit to be rewound before the input file is opened.

SELECTIVE DELETE infile(s)/SD

Prompts for user response before deleting files.

SHARED READING infile(s)/SR

Allows shared reading of a file that has already been opened for writing.

SPAN BLOCKS outdev:volume:outfile/SB = indev:volume:infile

Allows output file records to cross block boundaries when ANSI tapes are being copied to Files-11 volumes.

SPOOL infile(s)/SP[:n]

Specifies a list of files to be printed (n is the number of copies). This switch applies only if the user has the Serial Despooler or the Queue Manager. However, its use with the Queue Manager is not recommended.

SUPERSEDE outfile = infile(s)/SU

Copies an input file, superseding an existing output file of the same name and version number.

USER FILE DIRECTORY ENTRY outfile(s)/UF/FO] = infile(s)

Creates a User File Directory on a volume.

outfile	Specifies the UIC as [*,*] to transfer multiple infile UICs.
/FO	See APPEND.

UNLOCK infile(s)/UN

Unlocks a file that was locked as a result of being closed improperly. Lets the user know that the data contained in the file may have been corrupted.

UPDATE FILE outfile = infile(s)/UP[/FO]

Opens an existing file and writes it, from the beginning, into the output file.

outfile	Must be explicitly identified.
infile	Null parameters default to *.*. Input files replace the current contents of output files.

61

APPENDIX
SUMMARY OF EXAMPLE PROGRAM

The following listing summarizes the development of the FORTRAN program ADD and the manipulation of the resultant files.

```
>HEL <CR>
ACCOUNT OR NAME: CHARLES <CR>
PASSWORD: GREY <CR>

          RSX-11M BL18 MULTI-USER SYSTEM

GOOD MORNING
22-MAY-77 11:07 LOGGED ON TERMINAL TT4:

22-MAY-77

      SYSTEM WILL BE DOWN TODAY FROM 13:00-15:00 FOR
      CORRECTIVE MAINTENANCE

>EDI ADD.FTN <CR>
[CREATING NEW FILE]
INPUT
        TYPE 1 <CR>
1       FORMAT (' ENTER TWO NUMBERS - M,N') <CR>
        ACCEPT 2,K,L <CTRL/R>
        ACCEPT 2,K,L <CR>
2       FORMAT (2I2) <CR>
        PRINT *U
        TYPE 3,K+L <CR>
3       FORMAT (' THE SUM IS ',I5) <CR>
        STOP <CR>
        END <CR>
<CR>
*EX <CR>
[EXIT]
>PIP TI:=ADD.FTN <CR>

        TYPE 1
1       FORMAT (' ENTER TWO NUMBERS - M,N')
        ACCEPT 2,K,L
2       FORMAT (2I5)
        TYPE 3,K+L
3       FORMAT (' THE SUM IS ',I5)
        STOP
        END
>EDI ADD.FTN <CR>
[00009 LINES READ IN]
[PAGE 1]
*LOCATE ENTER <CR>
1       FORMAT (' ENTER TWO NUMBERS - M,N')
*CHANGE/ENTER/TYPE/ <CR>
1       FORMAT (' TYPE TWO NUMBERS - M,N')
*NEXT <CR>
*PRINT <CR>
        ACCEPT 2,K,L
```

62

```
*LOCATE SUM <CR>
3       FORMAT (' THE SUM IS ',I5)
*CHANGE/SUM/RESULT/ <CR>
3       FORMAT (' THE RESULT IS ',I5)
*LOCATE (2I5) <CR>
[*EOM*]
*TOP <CR>
*<CR>
        TYPE 1
*EXIT <CR>
[EXIT]
>EDI ADD.FTN <CR>
[00009] LINES READ IN
[PAGE 1]
*INSERT <CR>
C       THIS PROGRAM ADDS TWO NUMBERS TOGETHER <CR>
<CR>
*LOCATE NUMBERS <CR>
1       FORMAT (' TYPE TWO NUMBERS - M,N')
*ADD !INPUT PROMPT <CR>
*PRINT <CR>
1       FORMAT (' TYPE TWO NUMBERS - M,N')!INPUT PROMPT
*LOCATE RESULT <CR>
3       FORMAT (' THE RESULT IS ',I5)
*DELETE <CR>
*<ESC>
        TYPE 3,K+L
*INSERT <CR>
3       FORMAT (' THE SUM IS ',I5)!DISPLAY RESULT <CR>
<CR>
*TOP <CR>
*<CR>
C       THIS PROGRAM ADDS TWO NUMBERS TOGETHER
*RETYPE C       ADD DISPLAYS THE SUM OF TWO NUMBERS <CR>
*LIST <CR>
C       ADD DISPLAYS THE SUM OF TWO NUMBERS
        TYPE 1
1       FORMAT (' TYPE TWO NUMBERS - M,N')!INPUT PROMPT
        ACCEPT 2,K,L
2       FORMAT (2I5)
        TYPE 3,K+L
3       FORMAT (' THE SUM IS ',I5)!DISPLAY RESULT
        STOP
        END

*EXIT <CR>
[EXIT]
>FOR ADD,ADD=ADD <CR>
>TKB ADD,ADD,ADD=ADD
>RUN ADD <CR>
TYPE TWO NUMBERS - M,N
7,3 <CR>
THE SUM IS     10
TT13 -- STOP

>RUN ADD <CR>
TYPE TWO NUMBERS - M,N
522,628 <CR>
THE SUM IS 1150
TT13 -- STOP

>RUN ADD <CR>
TYPE TWO NUMBERS - M,N
9,16 <CR>
```

```
THE SUM IS    25
TT13 -- STOP
>PIP <CR>
PIP /LI <CR>


DIRECTORY DB0:[301,314]
25-MAY-77 14:48

ADD.OBJ;2               2.          25-MAY-77 14:31
ADD.TSK;1              29.       C  20-MAY-77 14:32
ADD.MAP;1               4.          25-MAY-77 14:32
ADD.STB;1               3.          25-MAY-77 14:32
ADD.FTN;1               1.          20-MAY-77 15:26
ADD.TSK;3              29.       C  25-MAY-77 14:33
ADD.OBJ;1               2.          20-MAY-77 15:27
ADD.TSK;2              29.       C  20-MAY-77 15:28

TOTAL OF 99./112. BLOCKS IN 8. FILES

PIP
PIP>^CTRL/Z>
>PIP ADD.*/PU <CR>
>PIP /LI <CR>

DIRECTORY DB0:[301,314]
25-MAY-77 14:56

ADD.OBJ;2               2.          25-MAY-77 14:31
ADD.MAP;1               4.          25-MAY-77 14:32
ADD.STB;1               3.          25-MAY-77 14:32
ADD.FTN;1               1.          20-MAY-77 15:26
ADD.TSK;3              29.       C  25-MAY-77 14:33

TOTAL OF 39./49. BLOCKS IN 5. FILES

>PIP ADDTWO.*;*=ADD.*;*/RE  <CR>
>PIP /LI <CR>

DIRECTORY DB0:[301,314]
25-MAY-77 16:43

ADDTWO.OBJ;2            2.          25-MAY-77 14:31
ADDTWO.MAP;1            4.          25-MAY-77 14:32
ADDTWO.STB;1            3.          25-MAY-77 14:32
ADDTWO.FTN;1            1.          20-MAY-77 15:26
ADDTWO.TSK;3           29.       C  25-MAY-77 14:33

TOTAL OF 39./49. BLOCKS IN 5. FILES
```

OVERLAYS

```
PIP
PIP>TI:=A.FTN\M\N
        TYPE 10
        ACCEPT 15,I
15      FORMAT(I2)
10      FORMAT(2X,'DAR DATO ENTERO(XX) PARA MAIN= ',$)
        CALL B
        CALL C
        TYPE 20,I
20      FORMAT(2X,'DATO LEIDO EN MAIN= ',I2)
        END
PIP>TI:=B.FTN
        SUBROUTINE B
10      FORMAT(2X,'DAR DATO ENTERO(XX) PARA SUB.B= ',$)
        TYPE 10
        ACCEPT 15,J
15      FORMAT(I2)
        CALL D
        CALL E
        TYPE 20,J
20      FORMAT(2X,'DATO LEIDO EN SUB.B= ',I2)
        RETURN
        END
PIP>TI:=C.FTN
        SUBROUTINE C
        TYPE 10
        ACCEPT 15,K
15      FORMAT(I2)
10      FORMAT(2X,'DAR DATO ENTERO(XX) PARA SUB.C= ',$)
        CALL F
        CALL G
        TYPE 20,K
20      FORMAT(2X,'DATO LEIDO EN SUB.C= ',I2)
        RETURN
        END
PIP>TI:=D.FTN
        SUBROUTINE D
        TYPE 10
10      FORMAT(2X,'ENTRANDO A SUB.D')
        TYPE 20
 20      FORMAT(2X,'SALIENDO DE SUB.D')
         RETURN
         END

TI:E.FTN
        SUBROUTINE E
        TYPE 10
10      FORMAT(2X,'ENTRANDO A SUB.E')
        TYPE 20
20      FORMAT(2X,'SALIENDO DE SUB.E')
        RETURN
        END
PIP TI:=F.FTN
```

I. PROGRAM TO COUNT NEGATIVE NUMBERS IN A TABLE

```
        ;20. SIGNED WORDS
        ;BEGINNING AT LOC VALUES
        ;COUNT HOW MANY ARE NEGATIVE IN RØ

        RØ=%Ø
        R1=%1
        R2=%2
        SP=%6
        PC=%7

        .=5ØØ

START:  MOV #.,SP               ;SET UP STACK
        MOV #VALUES,R1          ;SET UP POINTER
        MOV #VALUES+4Ø.,R2      ;SET UP COUNTER
        CLR RØ

CHECK:  TST (R1)+               ;TEST NUMBER
        BPL NEXT                ;POSITIVE?
        INC RØ                  ;NO, INCREMENT COUNTER
NEXT:   CMP R1,R2               ;YES, FINISHED?
        BNE CHECK               ;NO, GO BACK
        HALT                    ;YES, STOP

VALUES: Ø
        .END START
```

II. PROGRAM TO COUNT ABOVE AVERAGE QUIZ SCORES

```
        ;LIST OF 16. QUIZ SCORES
        ;BEGINNING AT LOC SCORES
        ;KNOWN AVERAGE IN LOC AVRAGE
        ;COUNT IN RØ SCORES ABOVE AVERAGE

        RØ=%Ø
        R1=%1
        R2=%2
        R3=%3
        SP=%6
        PC=%7

        .=5ØØ

START:  MOV #.,SP               ;SET UP STACK
        MOV #16.,R1             ;SET UP COUNTER
        MOV #SCORES,R2          ;SET UP POINTER
        MOV #AVRAGE,R3
        CLR RØ

CHECK:  CMP (R2)+,(R3)          ;COMPARE SCORE AND AVRAGE
        BLE NO                  ;LESS THAN OR EQUAL TO AVRAGE?
        INC RØ                  ;NO, COUNT
NO:     DEC R1                  ;YES, DECREMENT COUNTER
        BNE CHECK               ;FINISHED? NO, CHECK
        HALT                    ;YES, STOP
AVRAGE: 65.

SCORES: 25.,7Ø.,1ØØ.,6Ø.,65.,8Ø.,8Ø.,4Ø.
        55.,75.,1ØØ.,65.,9Ø.,7Ø.,65.,7Ø.

        .END START
```

- EJERCICIOS DE COMANDOS MCR -

1.- A) Entrar al sistema de tal manera que éste nos reconozca.

B) Inicializar el dispositivo DISCO., cuyo nombre físico es DK1, con el nombre de PRUEB 1.

C) Monte el DISCO. cambiando las protecciones de DEFAULT

D) Crear una UFD en el DISCO con un grupo y un miembro de 113, 2 respectivamente.

E) Reservar 20 bloques para el archivo de CHECKPOINT en el DISCO..

F) Definir el nombre lógico D11 al dispositivo físico DISCO

G) Desplegar en la terminal las asignaciones LOGIN Y LOCALES

H) Borrar el área que reservamos para el archivo de CHECKPOINT en el DISCO..

I) Borrar las asignaciones lógicas especificadas como locales

J) Instalar la tarea PIP que se encuentra en la cuenta 1.54 con una prioridad de 103

K) Desplegar en la terminal el estado de la tarea instalada PIP y también su descripción general.

L) Cambiar la prioridad de la tarea PIP, a una prioridad de - DEFAULT.

LL) Desplegar en la terminal, información acerca de todas las terminales reconocidas por el sistema.

M) Terminar anormalmente la tarea PIP TI, si previamente la activamos con la siguiente instrucción:

PIP TI:= (113,1) CARTO1.LCS

DONDE TI Es la terminal donde estamos: cuando el número de terminal sea menor de - 10 se pondrá TTn:

N) Igual que la M pero sin indicarle la tarea que vamos a ter minar anormalmente.

Ñ) Desmontar el DISCO.

O) Concluir la sesión con el sistema

2.- A) Entrar al sistema, (Sin que nos pregunte ACCOUNT OR NAME)

B) Inicializar al DISCO. DK1, con un nombre PRUEB 2 y cambiando las protecciones de DEFAULT.

C) Montar el DISCO con una prioridad de DEFAULT

D) Crear una UFD en el DISCO con un grupo y un miembro de 11e,3 respectivamente; y cambiando las protecciones de DEFAULT.

E) Con qué instrucción se instala, CORRE y REMUEVE la tarea EDI que está en la cuenta 1,54

F) desplegar en la terminal los números de unidades lógicas asociados a los dispositivos físicos de la tarea EDI.

G) Manejar minúsculas en la terminal .

H) Listar velocidades de las terminales y, cambiar la velocidad en la que se esté.

I) Hacer lo contrario de la G.

J) Dejar la velocidad como estaba antes de cambiarla

K) Desplegar todas las terminales que estén LOGON

L) Si por alguna razón no sirve la terminal 4 y vamos a ejecutar un programa que hace uso de ella; para no modificar el programa, hacemos que use la TT3 ¿Qué comando le indicamos?

LL) Para probar lo anterior, en lugar de un programa usaremos la siguiente instrucción:

PIP TT4:=(112,6) PRUEBA.LCS

NOTA: Usar los números de terminales que esten LOGON.

M) Desmontar el DISCO DK1.

O) Crear una UFD con un grupo y un miembro de 113,3, respectivamente y con un protección de DEFAULT, en el DISCO

P) Mandar el mensaje: "HOLA,MI NOMBRE ES:" a la terminal O

Q) Mandar el mensaje: "ADIOS, MI NOMBRE ES:" a todas las terminales que esten LOGON

R) Desplegar la descripción de cada participación en memoria existentes en el sistema

S) Desplegar las tareas activas de todas las terminales.

T) Correr la tarea PIP que está instalada, (checar que esté instalada) después de transcurridos 3 minutos y ejecutar la cada 90 segundos (checar el estado de la tarea)

U) Igual que la T pero sincronizado a segundos

V) Desmontar el DISCO DK1

W) Dar por concluída la sesión con el sistema

EVALUACION DE COMANDOS DE MCR

1.- Tenemos un proceso que debe ser checado cada minuto por un programa que hace interfase con los controles que afectan al mecanismo de control.

Escriba el comando para que el programa CONTRL corra cada 4 minutos.

2.- Tiene usted una cinta cuya etiqueta es 'CINTA' y contiene un archivo que usará su programa

¿Qué comando y formato deberá dar para que la cinta pueda ser utilizada por su programa?

3.- El programa 'RECORD' es muy utilizado en su instalación y quiere usted evitarse el trabajo de instalarlo, correrlo y removerlo, las veces que se use.

¿Cómo hace para lograrlo?

4.- Entra usted por la cuenta (11,27), pero necesita estar en la (300,300).

¿Cómo le hace?

5.- A diez oficinas de usted se encuentra la máquina y el operador principal, usted necesita que está montado el disco con etiqueta 'PROCESI' en la unidad 2.

Notifiquelo desde su terminal al operador.

6.- Si se ha realizado la sig. secuencia de comandos:

```
INS (1Ø,1Ø) PROG/TASK= PROG1
SET /UIC= (2Ø,2Ø)
```

a) Si deseamos ejecutar la tarea PROG instalado que hacemos?

b) Si damos el comando:
```
RUN (1Ø,1Ø)  PROG
```
como abortamos el programa

c) Si damos el comando:
```
RUN  PROG
```
que programa ejecuta (y como lo abortamos)

d) Si damos el comando:
```
RUN  PROG1
```
que programa ejecuta (y como lo abortamos)

7.- Si se instala la tarea 'CRAX', como podemos averiguar sus asignaciones de unidades lógicas.

8.- Si la tarea tiene la unidad lógica ó asignada CL:. como se la asignamos a LP: ?

9.- Si una tarea 'XXX' esta contruída para correr siempre en- la partición 'XXXPAR', y en el momento en que queremos - ejecutarla, esta partición se encuentra ocupada, ¿Cómo po demos ejecutarla en la partición 'gen'.

10.- Si se estan ejecutando 20 tareas de una misma prioridad - 50., y en la tarea 'AA', que también esta ejcutandose, esta elaborando un reporte que le urge al gerente de la empresa, y por la cantidad de tareas ejecutandose el proceso se vuelve algo lento, como podemos hacer para apresu rar la generación del reporte?.

'ihl

PIP -EJEMPLOS-

Editar 2 archivos con los datos que se desee. Deberán llamarse:

PRUEBA.DAT (cada uno con varias versiones)

DATOS. DAT

También se usarán los archivos A.DAT y B.DAT de la cuenta que se crearon para practicar EDITOR. .

COPIA DE ARCHIVOS:

1) Copiar a su propia cuenta los archivos A.DAT y B.DAT
2) Crear una nueva versión de A.DAT por medio de PIP
3) Crear un archivo SY: C.DAT que sea la unión de los archivos A.DAT y B.DAT.
4) Imprimir los archivos PRUEBA.DAT y B.DAT, de manera que salgan como un sólo archivo en la impresión.
5) Hacer el archivo DATOS.DAT contiguo.
6) Teniendo el archivo A.DAT;2, copiar el archivo B.DAT; 1 a otro archivo que se llame también A.DAT;2 sin usar el switch/NV; ver que sucede, y luego hacer lo mismo pero usando el switch/NV.
7) Copiar DATOS.DAT a un archivo que se llame PRUEBA.DAT de manera que éste último contenga sólo la información de - DATOS.DAT, perdiéndose los datos que tenía PRUEBA.DAT anteriormente.
8) Copiar de una cuenta a otra todos los archivos que se llamen A, de cualquier extensión y sólo las últimas versiones.
9) Realizar A.DAT;Ø = B.DAT;Ø/SU y ver que sucede.

CONTROL DE ARCHIVOS:

1) Listar el directorio para saber que archivos se tienen en la propia cuenta.
2) Ejecutar el comando necesario para conocer los números de identificación de todos los archivos que se llamen PRUEBA
3) Listar en forma reducida el directorio propio, en una terminal diferente a aquella en la que se esta trabajando.

4) Conocer el número de bloques que se esta ocupando en total

5) Borrar las versiones más antiguas de todos los archivos de la propia cuenta.

6) Cambiar la protección del archivo A.DAT de manera que sólo puedan leerlo tanto el dueño como los demás miembros del - mismo grupo.

7) Listar este archivo en la terminal en que se trabaja. No se desea el directorio, sino el contenido del archivo.

8) Comprobar que la protección A.DAT realmente se cambió.

9) Borrar todas las versiones del archivo A.DAT y volverlo a copiar de la cuenta

10) Purgar todos los archivos de nombre B y de nombre PRUEBA, en un sólo comando.

11) Teniendo DATOS.DAT;3 y DATOS.DAT;5, realizar el siguiente comando entre DATOS.DAT/PU:2 y ver que versiones de ese archivo conserva.

12) Conocer cuanto espacio libre hay en el dispositivo de DEFAULT

13) Copiar al dispositivo de DEFAULT, bajo la cuenta en que corre PIP, un archivo A.DAT de otra cuenta, sin especificar archivo de salida, y ver que sucede.

14) Establecer un UIC de default, listar el directorio correspondiente a ese UFD y regresar a la propia cuenta.

15) Agregar el archivo C.DAT a el final del archivo B.DAT.

16) Realizar el proceso anterior usando como archivo de salida *.*;* y ver que sucede.

18) Cambiarle el nombre a C.DAT por el de CAMBIO.DAT.

19) Cambiarle a la versión más antigua de B.DAT el nombre por - el de CAMBIO.DAT, usando el subswitch necesario para que no marque error (Ambos CAMBIO.DAT deben tener igual número de de versión).

20) Cambiar la extensión de los archivos DATOS.DAT y PRUEBA.DAT por .DTO en un sólo comando.

21) Crear una nueva entrada en el UFD del archivo DATOS.DTO que se llame INF.DTO

22) Crear otra entrada de DATOS.DTO sin especificar archivo de entrada y ver que sucede

23) Remover la entrada INF.DTO del UFD

24) Listar en la propia terminal el archivo PRUEBA.DAT;Ø por medio de su número de identificación.

J.A.TORRES 73

CHAPTER 5

INDIRECT COMMAND FILES

5.1 INDIRECT COMMAND FILES

An indirect command file is a text file containing a list of commands exclusive to, and interpretable by, a single task, usually a system-supplied component of RSX-11M, such as MACRO-11 or the Task Builder. In addition to task components, MCR has an indirect file processor (...AT.) which interprets commands and either acts on them or passes them on to MCR for processing.

Indirect files are initiated by replacing the command string required by a task with a file specifier preceded by an at sign (@).

For example, to initiate a file of MACRO-11 commands, the user would input:

```
>MAC @INPT.CMD
```

After MACRO-11 is initiated, it accesses the file INPT.CMD for its commands.

To initiate a file of MCR commands, the user would simply enter the file specifier preceded by an at (@) sign, as shown below,

```
>@filespec
```

anytime MCR will accept input.

The default file type for indirect command files is .CMD. An indirect file may contain any command interpretable by the task to which it is directed, but no others.

Indirect files for tasks other than MCR may not contain indirect file references.

MCR indirect files may reference other MCR indirect files, but the maximum nesting depth is limited to four levels of files. MCR will display on the requesting terminal every command retrieved from an MCR indirect command file.

5.2 MCR INDIRECT FILE PROCESSOR

The MCR indirect file processor task (AT.) is capable of reading a command input file and interpreting each line as either a command to be passed directly to MCR or a request for action by the task itself.

Commands for MCR are simply entered on a line, requiring no special prefix characters, whereas commands to the indirect command file processor always begin with a period (.). Comments are indicated by a leading exclamation point or semicolon and are printed on the entering terminal.

Commands interpreted by the indirect file processor provide the following capabilites:

Define a label (.label:);
Ask a question and wait for reply (.ASK);
Delay execution for a specified period of time (.DELAY);
Branch to a label (.GOTO);
Determine whether a task is active or not (.IFACT/.IFNACT);
Determine whether a symbol is defined or not (.IFDF/.IFNDF);
Determine whether a task is installed in the system or not (.IFINS/.IFNINS);
Determine whether a symbol is true or false (.IFT/.IFF);
Branch to a label on detecting an error (.ONERR);
Pause for operator action (.PAUSE);
Set the value of a symbol to true or false (.SETT AND .SETF);
Wait for a task to finish execution (.WAIT);
Start a task, pass a command line to it and continue (.XQT), and provide commentary ("!text" OR ";text").

5.2.1 Symbols

It is possible to define symbols which represent true or false values and which may be tested to control flow through the indirect command file. Symbols are one to six ASCII characters in length. Numbers, upper and lower case letters, and special characters are all valid. Symbols are defined by the .ASK directive or by the .SETT and .SETF directives. The first appearance of the symbol in one of these directives defines the symbol and sets it to true or false; subsequent directives may test the value or redefine it.

There is one special symbol which is always defined and is used to determine whether the system on which the user is running is mapped or unmapped. This symbol is

<MAPPED>

The brackets are required syntax. If the system is unmapped, <MAPPED> has a value of false; if the system is mapped, it has a value of true.

Example:

```
.IFT <MAPPED> SET /UIC=[1,54]
```

If the system is mapped, the Set command is passed to MCR.

5.2.2 Commands to MCR

A command is normally passed directly to MCR for processing, and the indirect file processor waits until MCR has processed it before going on to the next command in the file. However, if the first character in the line is a period (.), this signals that the command is to be interpreted by the indirect file processor and appropriate action is to be taken. The command is scanned and processed from left to right. When the final command on the line is processed and the conditions are true, the rest of the line will be passed to MCR. If the conditions are false, the rest of the line will be ignored and the next line will be processed.

Example:

```
.IFNINS ...PIP INS [1,50]PIP
```

If task ...PIP is not installed, the command INS [1,50]PIP will be passed to MCR for execution. If ...PIP was installed, the INS [1,50]PIP command would not be passed to MCR. Each command passed to MCR is also displayed on the entering terminal as a log of executed commands.

5.2.3 Switches

The indirect file processor accepts two switches: /TR and /DE.

/TR Indicates that a trace of each line processed is to be provided on the entering terminal. This function is useful for debugging an indirect command file. Each command line, including those that begin with a period (directive), is displayed. The period on the first directive in the line is changed to an exclamation mark (comment) and displayed. If the command causes some action to occur, the next line printed will indicate the action; usually, this line consists of the MCR commands issued as a result of the previous directive.

/DE Indicates that the indirect command file is to be deleted when processing is complete.

These switches may be prefixed with a minus sign (-) or "NO" to negate the action of the switch, e.g., /NOTR will not generate a trace. The defaults are /NOTR and /NODE.

5.2.4 Multi-Level Indirect Files

Up to four levels of indirect command files may be specified. Each time a new level is entered, all symbols previously defined are masked out of the symbol table, and only symbols defined in the current level are available; symbols are local to the level of command file in which they are defined. When control returns to a previous level, the symbols defined in that level are available again.

5.2.5 Syntax

All directives are separated from their arguments by at least one space. MCR commands are separated from directives by at least one space.

5.2.6 Directives

5.2.6.1 Define a Label — .label:

Labels are always at the beginning of the line; they may be on a line with additional directives and/or an MCR command, or on a line by themselves. When control is passed to a line with a label, the line is processed as if the label was not there. Commands do not have to be separated from the label by a space. Only one label is permitted per line. Labels are 1 to 6 characters long and must be preceded by a period and terminated by a colon. Any valid ASCII character is allowed in a label.

Examples:

```
.100:
.HERE:
.10$:
```

5.2.6.2 Ask a Question and Wait for a Reply — .ASK

The .ASK directive will ask a question, wait for a reply, and set a specified symbol to the value of true or false, depending on the reply. If the symbol is not defined, an entry will be made in the symbol table. If the symbol is already defined, its value will be set by the answer to the question.

Format:

```
.ASK ssssss txt-strng
```

where:

ssssss = 1- to 6-character symbol which is assigned a true/false value.

txt-strng = any ASCII string of characters which must be preceded by at least one blank.

The text-string will be prefixed with an asterisk and suffixed with "? (Y/N):" when it is displayed. Three answers are recognized by the indirect file processor:

1. Y<CR> - set symbol ssssss to true.
2. N<CR> - set symbol ssssss to false.
3. <CR> - set symbol to false. <CR> indicates carriage return.

Example:

```
.ASK A DO YOU WANT TO INSTALL PIP
```

will display

```
>* DO YOU WANT TO INSTALL PIP? [Y/N]:
```

on the entering terminal. Symbol A will be set to true or false.

5.2.6.3 Delay Execution for a Specified Period of Time .DELAY

The .DELAY directive is used to .delay further processing of the indirect file for a specified period of time.

Format:

```
.DELAY nnu
```

where:

```
nn = Number of time units to delay
 u = T - ticks;
     S - seconds;
     M - minutes, and
     H - hours.
```

The parameter nn is octal, unless terminated with a period. For example:

```
10S is 10(8) seconds
10.S is 10(10) seconds
```

When the DELAY directive is executed, the message

```
AT. -- DELAYING
```

is issued.

When the time period expires and the task resumes, the message

```
AT. -- CONTINUING
```

is issued.

Example:

The directive

```
.DELAY 20M
```

will delay processing for 20(8), or 16(10) minutes.

5.2.6.4 Branch to a Label

.GOTO

Branches from one line in an indirect file to another are accomplished with the .GOTO directive. All commands between the .GOTO directive and the specified label are ignored. Branches may go forward or backward in the file.

Format:

```
.GOTO label
```

where label is the name of the label, but without the leading period and trailing colon. The label must be preceded by at least one space.

Example:

The directive

```
.GOTO 100
```

will transfer control to the line containing the label .100:.

5.2.6.5 Logical Test (IF)

There are a number of directives which make tests; if the test is true, the rest of the command is processed. Logical tests may be combined into a compound logical test with the .AND and .OR directives.

5.2.6.5.1 Test if Symbol Is True or False

.IFT/.IFF

The value of a symbol may be tested for true or false; if the test is true, the rest of the command is processed.

Format:

```
.IFT  ssssss      If symbol ssssss is true.
.IFF  ssssss      If symbol ssssss is false.
```

The symbol must be preceded and followed by at least one blank, except at the end of a line, in which case, it need not be followed by a blank.

Example:

```
.IFT  A  .GOTO  100
.IFF  B  .GOTO  200
```

5.2.6.5.2 Test if Symbol Is Defined or Not Defined

.IFDF/.IFNDF

A test can be made to determine whether a symbol has been defined or not defined; if the test is true, the rest of the command is processed. This directive does not test the value of the symbol.

Format:

```
.IFDF  ssssss      If symbol ssssss is defined.
.IFNDF ssssss      If symbol ssssss is not defined.
```

The symbol ssssss must be preceded and followed by at least one blank, except at the end of a line, in which case, it need not be followed by a blank.

Example:

```
.IFDF  A  .GOTO 100
.IFNDF  A  .ASK A DO YOU WANT TO SET TIME
```

5.2.6.5.3 Test if Task Is Installed or Not Installed

.IFINS/.IFNINS

A test can be made to determine whether a task is installed in the system; if the test is true, the rest of the command is processed.

Format:

```
.IFINS  tttttt      If task tttttt is installed.
.IFNINS  tttttt     If task tttttt is not installed.
```

The task name tttttt must be preceded and followed by at least one blank, except at the end of a line, in which case, the trailing blank is not required.

Example:

```
.IFINS  ...PIP  .GOTO 250
.IFNINS  ...PIP  INS [1,50]PIP
```

5.2.6.5.4 Test if Task Is Active or Not Active

.IFACT/.IFNACT

A test can be made to determine whether a task is active; if the test is true, the rest of the command is processed.

Format:

```
.IFACT  tttttt      If task tttttt is in execution.
.IFNACT  tttttt     If task tttttt is not in execution.
```

The task name tttttt must be preceded and followed by at least one blank, except at the end of a line, in which case, it need not be followed by a blank.

Example:

```
.IFACT   REPORT  .GOTO 350
.IFNACT  REPORT  RUN REPORT
```

5.2.6.5.5 Compound Tests

"If" tests may be combined via the .AND and .OR directives. In addition, an implied .AND is effected when multiple IFs appear on the same line without being separated by an .AND directive. In this case, the command is executed only if all tests are true. The compound operators .AND and .OR must be preceded and followed by at least one blank.

Examples:

```
.IFT A .AND .IFF B  .GOTO C
.IFT A .IFF B  .GOTO C          Same effect as first line
.IFT A .OR  .IFF B RUN PIP
```

5.2.6.6 Branch to Label on Detecting an Error

.ONERR

If one of the following errors is detected:

1. Undefined symbol referenced,
2. Symbol table overflow,
3. Undefined label, or
4. Syntax error,

control will be passed to the line containing the specified label. This feature provides the user with a means of gaining control to terminate command file processing in an orderly manner.

Format:

```
.ONERR  label
```

Control is passed to the line starting with ".label:" upon detecting an error. The .ONERR directive may be issued at any place in the command file. If the directive is executed, error processing will be passed to that label until a new .ONERR directive is executed. If the label specified by the .ONERR directive does not exist and an error condition has occurred, the task will exit.

Example:

```
.ONERR  100
```

5.2.6.7 Pause for Operator Action

.PAUSE

It is possible to interrupt the processing of an indirect file and wait for operator action. This is done using the .PAUSE directive. Issuing a .PAUSE directive will cause the indirect file processor task to supend itself. The operator may perform some specified operations and then cause the task to resume.

Format:

```
.PAUSE
```

When the indirect file processor task suspends itself, it displays the following message on the entering terminal:

```
AT. -- PAUSING.  TO CONTINUE TYPE "RES tttttt"
```

where:

tttttt = is the name of the task.

When the operator is ready to resume, he types:

```
>RES  tttttt
```

The task then prints out:

```
AT. -- CONTINUING
```

and it continues processing where it left off.

5.2.6.8 Set Symbol to True or False

.SETT/.SETF

A symbol may be defined or its value may be changed by use of the .SETT/.SETF directives. If the symbol is not defined, a symbol table entry is built and set to the appropriate value; if the symbol exists, the value is set appropriately.

Format:

```
.SETT  ssssss     Set symbol ssssss to true.
.SETF  ssssss     Set symbol ssssss to false.
```

The symbol ssssss must be preceded by at lerst one blank.

Example:

```
.SETT  X
;SETF  ABCDE
```

5.2.6.9 Wait for a Task to Finish Execution — .WAIT

Processing of the indirect command file may be suspended until a particular task has terminated by issuing the .WAIT directive.

Format:

```
.WAIT  tttttt
```

The task name tttttt must be preceded by at least one blank.

Example:

```
.WAIT  PIP
```

5.2.6.10 Initiate Parallel Task Execution — .XQT

Normally, a command is passed to MCR and the indirect file processor waits until the command is completely executed. However, it is possible to initiate a task and not wait for it to complete before executing the next indirect file command. The MCR Run command initiates tasks, and the indirect file processor will continue as soon as the Run command is processed. But the user is not able to pass a command string to the target task via the Run command. Using the .XQT directive, it is possible to start a task, pass a command string to it, and continue processing command lines in parallel with the initiated task.

Format:

```
.XQT  cmd command-string
```

where:

cmd	is the name of the task (e.g., MAC, PIP, DMO).
command-string	is the command to the task.

Using the .XQT command provides a facility to initiate parallel processing of tasks. To synchronize the execution of parallel tasks, the .WAIT directive can be used to suspend command file processing until the specified task has completed.

Example:

```
.XQT  MAC TEST,TEST=TEST
.XQT  TKB BLD,BLD=BLD
.WAIT MAC
.WAIT TKB
```

The example starts an assembly and a task build executing in parallel and then waits for the two tasks to complete.

AT. -- FILE READ ERROR

An error was detected in reading the indirect file. Usually due to records which are more that 80. bytes long.

AT. -- INVALID ANSWER

In response to a question from .ASK, the operator entered something other than Y, N, or null, followed by carriage return. The question will be repeated.

AT. -- MAXIMUM INDIRECT FILES EXCEEDED

An attempt has been made to reference an indirect file at a nested depth greater than four.

AT. -- PAUSING. TO CONTINUE TYPE "RES tttttt"

Not an error message. The indirect file processor just executed a .PAUSE directive.

AT. -- RECORD LARGER THAN 80. BYTES

A record larger than 80. bytes has been encountered. Maximum record size is 80. bytes.

AT. -- REDEFINING SPECIAL SYMBOL

An attempt has been made to change the value of a special symbol (i.e., those bracketed with "<" ">"). They may not be redefined.

AT. -- SYMBOL TABLE OVERFLOW ssssss

The symbol table is full and there is no space for symbol ssssss.

AT. -- UNDEFINED LABEL llllll

The label llllll specified in a .GOTO or .ONERR directive cannot be found.

AT. -- UNDEFINED SYMBOL ssssss

The symbol ssssss is being tested, but it has not been defined.

Notes:

1. The default file type for indirect command files is .CMD.
2. When the indirect file processor reaches the end-of-file at the top level file, it will display:

   >@ <EOF>

   and exit.

```
@ IMPRIMIR
>* TIPO DE PROCESO (COPIA,FIN): [SJ: COPIA
>* DISPOSITIVO DE ENTRADA: [SJ: SY
>* DISPOSITIVO DE SALIDA: [SJ: TI
>* NOMBRE DEL PROGRAMA: [SJ: IMPRIMIR
>* TIPO DEL PROGRAMA: [SJ: CMD
>* VERSION DEL PROGRAMA: [SJ: 0
>PIP TI:=SY:IMPRIMIR.CMD;0
.SETS COP "COPIA"
.SETS FIN "FIN"  .
.10:
.ASKS TIPO TIPO DE PROCESO (COPIA,FIN):
.IF TIPO EQ COP .GOTO 20
.IF TIPO EQ FIN .GOTO 100
.GOTO 10
.20:
.ASKS ENT DISPOSITIVO DE ENTRADA:
.ASKS SAL DISPOSITIVO DE SALIDA:
.ASKS PROG NOMBRE DEL PROGRAMA:
.ASKS EXT TIPO DEL PROGRAMA:
.ASKS VER VERSION DEL PROGRAMA:
.ENABLE SUBSTITUTION
PIP'SAL':='ENT':'PROG'.'EXT';'VER'
.DISABLE SUBSTITUTION
.ASK OK DESEA COPIAR OTRO ARCHIVO
.IFT OK .GOTO 20
.100:
/
>* DESEA COPIAR OTRO ARCHIVO?  [Y/NJ:N
>MCR
```

centro de educación continua
división de estudios de posgrado
facultad de ingeniería unam

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

SISTEMA OPERATIVO

ING. DANIEL RIOS ZERTUCHE ORTUÑO

JULIO, 1980

CHAPTER 2

OPERATING SYSTEMS

OVERVIEW

The success of the PDP-11 family of operating systems is largely attributable to its ability to handle many diverse data processing applications. For example, RT-11 provides a single-user environment with foreground/background processing; RSTS/E provides a multi-user environment with economical timesharing; RSX-11M provides a multi-user on-line environment with data collection and process control; DSM-11 (MUMPS) provides the same with data base information systems; and IAS provides a multi-user environment with simultaneous timesharing, real-time, and batch processing.

Basic concepts pertaining to the structure of these systems are presented in this chapter.

FEATURE TOPICS

- Components and Functions
- Processing Methods
- Data Management
- Data Storage and Transfer Mode
- I/O Devices and Physical Data Access Characteristics
- Physical Device Characteristics
- File Structures and Access Methods
- Directories and Directory Access Techniques
- File Protection/File Naming
- User Interfaces
  - Special Terminal Commands
  - I/O Commands
  - Monitor and Command Language Commands
- System Utilities
- Operating System Comparative Chart

INTRODUCTION

Operating systems have two basic functions: they provide services for application program development and act as an environment in which application programs run. The character that an operating system has, that is, the services and environment it supplies, is appropriate only for a certain range of program development and application requirements, in order to serve selected needs efficiently. Operating systems for the PDP-11 family of computers, however, share many similar program development techniques and processing environments.

COMPONENTS AND FUNCTIONS

An operating system is a collection of programs that organizes a set of hardware devices into a working unit that people can use. Figure 2-1 illustrates the relationship between users, the operating system, and the hardware. PDP-11 operating systems basically consist of two sets of software: the monitor (or executive) software and the system utilities.

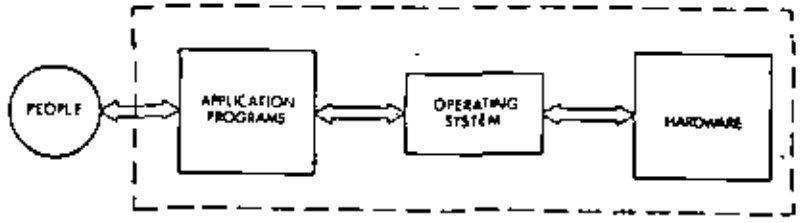


Figure 2-1 Computer System

An operating system monitor is an integrated set of routines that acts as the primary interface between the hardware and a program running on the system, and between the hardware and the people who use the system. The monitor's basic functions can be divided among the routines that provide the following services:

- device and data management
- user interface
- programmed processing services
- memory allocation
- processor time allocation

In general, a monitor can have two distinct operating components: a permanently resident portion and a transient portion. When a monitor

is loaded into memory and started, all of the monitor is resident in memory. Its first duty is to interface with the operator running the system. The monitor simply waits until an operator requests some service, and then performs that service. In general, these services include loading and starting programs, controlling program execution, modifying or retrieving system information, and setting system parameters. In most systems, these functions are serviced by transient portions of the monitor.

In some cases, when the monitor initiates another program's execution, the transient portion of the monitor can be over-written by the loaded program or swapped out. The permanently resident portion remains in memory to act on requests from the program. These generally include I/O services such as file management, device dependent operations, blocking and unblocking data, allocating storage space, and managing memory areas. In large systems, these services might also include inter-task communication and coordination, memory protection and parity checking, and task execution scheduling.

The dividing line between permanently resident and transient portions of the monitor, however, is not strictly based on user-interface functions and program-interface functions. In some systems, special monitor routines that service either the operator or programs might be stored on the system device, and are called into memory only as needed. The concern for space in small systems usually determines what portions of the monitor are resident at any time. The programmer or operator can control the size of the monitor, based on the needs for memory.

In some cases, the user can adjust the size of the monitor by eliminating features that are not needed in an application environment. RSTS/E, RSX-11M, and RSX-11S are examples of such systems. The RSX-11S system's monitor (called an executive) is always permanently resident when the system is operating. In this case, the user concerned with size can eliminate routines that perform unneeded operations. In general, however, all PDP-11 operating systems are designed to be flexible enough to operate in a relatively wide range of hardware environments.

System utilities are the individual programs that are run under control of the monitor to perform useful system-level operations such as source program assembly or compilation, object program linking, and file management.

System utility programs enhance the capabilities of an operating system by providing users with commonly performed general services. There are three classes of system utilities: those used solely or primarily for program development, those used for file management, and those used to perform special system management functions.

Program development utilities include text editors, assemblers and compilers, linkers, program librarians, and debuggers. File management utilities include file copy, transfer, and deletion programs, file format translators, and media verification and clean-up programs. System management utilities vary from system to system, depending on the purpose and functions the system serves. Some examples are system information programs, user accounting programs, and error logging and on-line diagnostic programs.

PROCESSING METHODS

The basic distinctions among operating systems are in the processing methods they use to execute programs. The distinctions to be discussed here are:

- single-user vs. multi-user
- single-job vs. foreground/background
- foreground/background vs. multi-programmming
- timesharing vs. event-driven multi-programming

A single-user operating system views demands upon its resources as emanating from a single source. It has only to manage the resources based on these demands. As a result, these systems do not require account numbers to access the system or data files. RT-11 is a single-user operating system.

A multi-user operating system receives demands for its resources from many different individuals. The system must manage its resources based on these demands. For example, several users may want sole access to the same device at the same time. The system must control access to these devices. In addition, the individuals may be using the system for different purposes, implying that some privacy must be maintained. As an effect, a multi-user system normally has an account system to manage different user's files. The IAS, RSTS/E, and RSX-11M systems are all multi-user systems, and all provide device allocation control and file accounts. In the case of the IAS, RSTS/E and systems, the file account structure is also used to keep track of the amounts of system resources an individual uses. Furthermore, the RSTS/E system extends privacy by protecting individual users at a system level from the effects of any other users of the system.

An RT-11 system can operate in two modes: as a single-job system, or as a foreground/background system. In a foreground/background system, memory for user programs is divided into two separate re-

gions. The foreground region is occupied by a program requiring fast response to its demands and priority on all resources while it is processing; for example, a real-time application program. The background region is available for a low-priority, preemptable program; for example, a compiler.

Two independent programs, therefore, can reside in memory, one in the foreground region and one in the background region. The foreground program is given priority and executes until it relinquishes control to the background program. The background program is allowed to execute until the foreground program again requires control. The two programs effectively share the resources of the system. When the foreground program is idle, the system does not go unused. Yet, when the foreground program requires service, it is immediately ready to execute. I/O operations are processed independently of the requesting job to ensure that the processor is used efficiently as well as to enable fast response to all I/O interrupts.

The basis of foreground/background processing is the sharing of a system's resources between two tasks. An extension of foreground/background processing is multiprogramming. In multiprogrammed processing, many jobs, instead of only two, compete for the system's resources. While it is still true that only one program can have control of the CPU at a time, concurrent execution of several tasks is achieved because other system resources, particularly I/O device operations, can execute in parallel. While one task is waiting for an I/O operation to complete, for example, another task can have control of the CPU.

The RSX-11 family of operating systems employs multiprogrammed processing based on a priority-ordered queue of programs demanding system resources. In this case, memory is divided into several regions called partitions, and all tasks loaded in the partitions can execute in parallel. Program execution, as in the RT-11 foreground/background system, is event-driven. That is, a program retains control of the CPU until it declares a significant event—normally meaning that it can no longer run, either because it has finished processing, or because it is waiting for another operation to occur. When a significant event is declared, the RSX-11 executive gives control of the CPU to the highest priority task ready to execute. Furthermore, a high-priority task can interrupt a lower-priority task if it requires immediate service.

The RSTS/E and MUMPS-11 systems also perform concurrent execution of many independent jobs. RSTS/E and MUMPS-11, however, process jobs on a timesharing rather than an event-driven basis, since

this is best suited for an interactive processing environment. Each job is guaranteed a certain amount of CPU time (a time slice), and jobs receive time one after another, in a round-robin fashion based on job priority levels set by the system. The system itself manages timesharing processing to obtain the best overall response depending generally on whether jobs are compute-bound or I/O-bound. The system manager or privileged users can also specify the minimum guaranteed time for a particular job when it gets service, as well as modifying its priority.

The IAS system effectively combines event-driven and timeshared processing in order to handle both real-time processing needs and interactive timesharing needs. In IAS, I/O tasks and any user-designated real-time tasks are assigned high priorities and receive service on an event-driven basis. All other tasks run at lower priorities on a timeshared basis, using any CPU time remaining after real-time, high-priority tasks have been serviced.

DATA MANAGEMENT

Digital computers deal with binary information only. The way in which people interpret and manipulate the binary information is called data management.

This section describes PDP-11 software data management structures and techniques, from the physical storage and transfer level to the logical organization and processing level. This includes:

- ASCII and binary storage formats — how binary data can be interpreted
- physical and logical data structures — the difference between how data storage devices operate and how people use them
- file structures — how physical units of data are logically organized for easy reference
- file directories — how files are located and retrieved
- file protection — how files are protected from unauthorized users
- file naming conventions — how files are identified

Physical and Logical Units of Data

Physical units of data are the elements which digital computer devices use to store, transfer and retrieve binary information. A bit (binary digit) is the smallest unit of data that computer systems handle. An example of a bit is the magnetic core used in some processor memories that is polarized in one direction to represent the binary number 0 and in the opposite direction to represent the binary number 1.

In PDP-11 computers, a byte is the smallest memory-addressable unit of data. A byte consists of eight binary bits. An ASCII character code can be stored in one byte. Two bytes constitute a 16-bit word. A word is the largest memory-addressable unit of data. Some machine instructions are stored in one word.

The smallest unit of data that an I/O peripheral device can transfer is called its physical record. The size of a physical record is usually fixed and depends on the type of device being referenced. For example, a card reader can read and transfer 80 bytes of information, stored on an 80-column punched card. The card reader's physical record length is 80 bytes.

A block is the name for the physical record of a mass storage device such as disk, DECtape or magnetic tape. An RK05 disk block consists of 512 contiguous bytes. Its physical record length is 512 bytes.

Physical blocks can be grouped into a collection called a device or a physical volume. This collection generally has a size equal to the capacity of the device medium. The term physical volume is generally used with removable media, such as disk packs or magnetic tape.

Logical units of data are the elements manipulated by people and user programs to store, transfer and retrieve information. The information has logical characteristics, for example, data type (alphabetic, decimal, etc.) and size. The logical characteristics are not device dependent; they are determined by the people using the system.

A field is the smallest logical unit of data. For example, the field on a punched card used to contain a person's name is a logical unit of data. It can have any length determined arbitrarily by the programmer who defines the field.

A logical record is a collection of fields treated as a unit. It can contain any logically related information, in any one of several data types, and it can be any user-determined length. Its characteristics are not device dependent, but can be physically defined. For example, a logical record can occupy several blocks, or it can reside in a single block, or several logical records can reside in a single block. Its characteristics are determined by the programmer.

A file is a logical collection of data that occupies one or more blocks on a mass storage device such as a disk, DECtape or magnetic tape. A file is a system-recognized logical unit of data. Its characteristics can be determined by the system or the programmer.

A file can be a collection of logical records treated as a unit. An example is a employee file which contains one logical record in the file for each employee. Each record contains an employee's name and address and other pertinent information. If the logical record length is 50 bytes and there are 200 employees, the complete employee file could be stored in 20 512-byte blocks. Depending on the file structure used in the system, the blocks could be scattered over the disk, or could be located one after the other.

A logical volume is a collection of files that reside on a single disk or DECtape. It is the logical equivalent of a physical device unit (a physical volume) consisting of physical records, such as a disk pack. The files on a volume may have no specific relationship other than their residence on the same magnetic medium. In some cases, however, the files on a volume may all belong to the same user of the system.

Figure 2-2 illustrates some of the kinds of physical and logical units of data that PDP-11 computer systems handle.

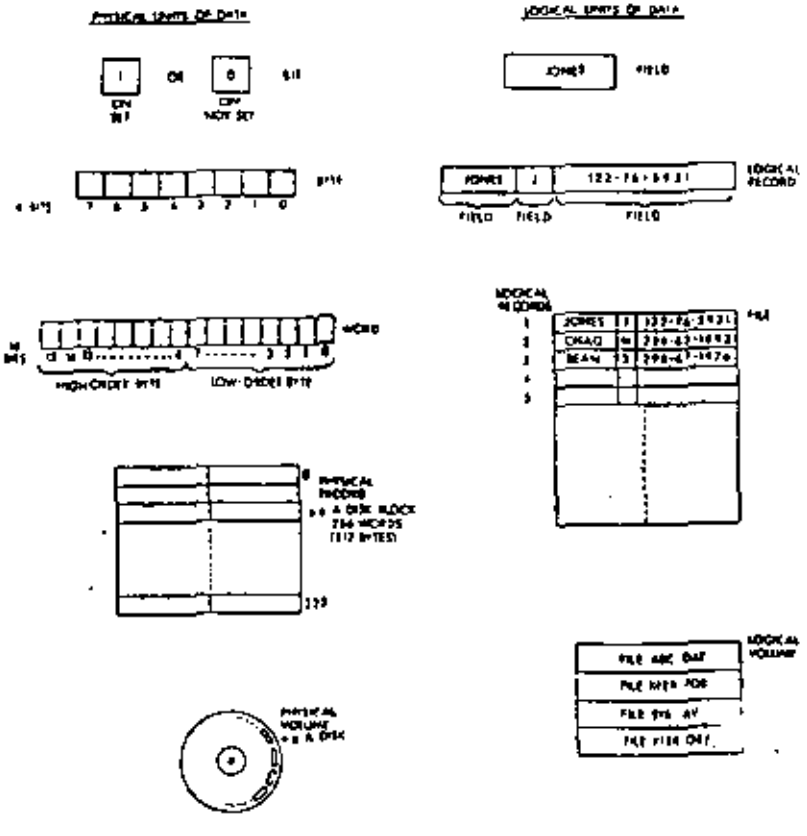


Figure 2-2 Physical and Logical Data Storage

Data Storage and Transfer Modes

All PDP-11 operating systems use two basic methods of data storage: ASCII and binary. Data stored in ASCII format conform to the American National Standard Code for Information Interchange, in which each character is represented by a 7-bit code. The 7-bit code occupies the low-order seven bits of an 8-bit byte. Depending on the operating system's storage techniques, the high-order bit may be used for parity checking and special formatting, or it may be ignored. Text files such as source programs are examples of data stored in ASCII format.

Binary storage always uses all eight bits of a byte to store information. The significance of any bit varies depending on the kind of information to be stored. Machine instructions, 2's complement integer data, and floating point numeric data are some examples of data stored in binary format.

Figure 2-3 illustrates the way in which binary data can be interpreted as either ASCII data or machine instructions. The figure shows two examples of a word of storage containing the same sequence of bits, interpreted first as two ASCII characters and second as a machine instruction. When a word of storage is interpreted as two ASCII characters, the binary digits are grouped into octal digits in a bytewise manner. Each byte is grouped into three octal digits. The low-order two octal digits contain three binary digits. The high-order octal digit contains two binary digits. When a word of storage is interpreted as a machine instruction, the binary digits are grouped into six octal digits in a wordwise manner. Proceeding from the low-order binary digit, each group of three binary digits is interpreted as an octal digit. The single remaining high-order binary digit is interpreted as an octal digit.

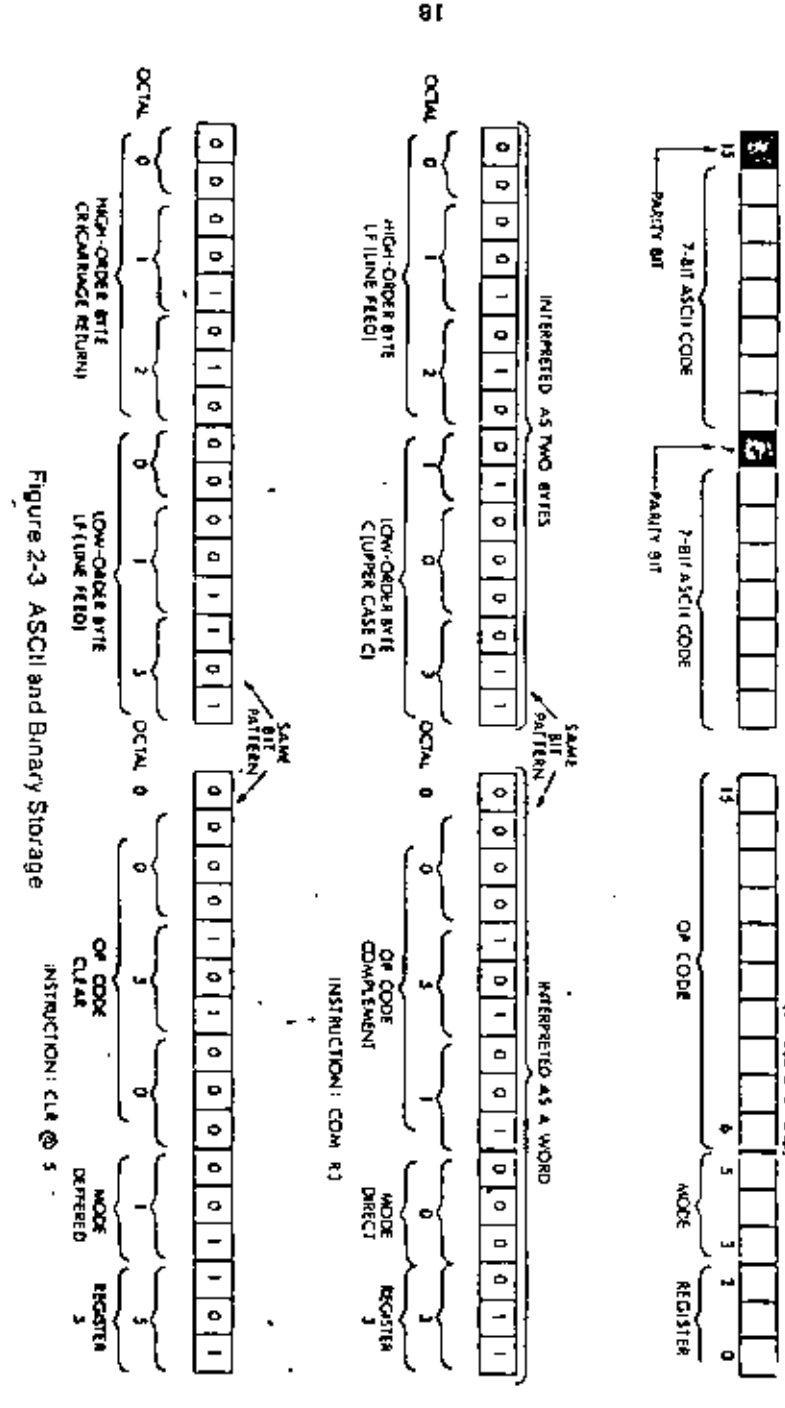


Figure 2-3 ASCII and Binary Storage

In large, sophisticated systems such as RSTS/E, RSX-11, and IAS, the way in which data are stored on the byte or bit level is rarely a concern of the application programmer. The operating system handles all data storage and transfer operations. In smaller systems such as RT-11, the programmer can become involved in data storage formats. A particular application may require the selection of a particular storage format.

The data storage format is related to the way in which data are transferred in an I/O operation.

Formatting can also be applied at a higher level to define the type of data file being processed. In the RT-11 system, there are four types of binary files; each type signifies that a special interpretation applies to the kind of binary data stored. For example, a memory image file is an exact picture of what memory will look like when the file is loaded to be executed. A relocatable image file, however, is an executable program image whose instructions have been linked as if the base address were zero. When the file is loaded for execution, the system has to change all the instructions according to the offset from base address zero.

I/O Devices and Physical Data Access Characteristics

In a PDP-11 computer system, data moves from external storage devices into memory, from memory into the CPU registers, and out again. The window from external devices to memory and the CPU is called the I/O page. Each external I/O device in a computing system has an external page address assigned to it. Figure 2-4 illustrates the data movement path in a PDP-11 computing system.

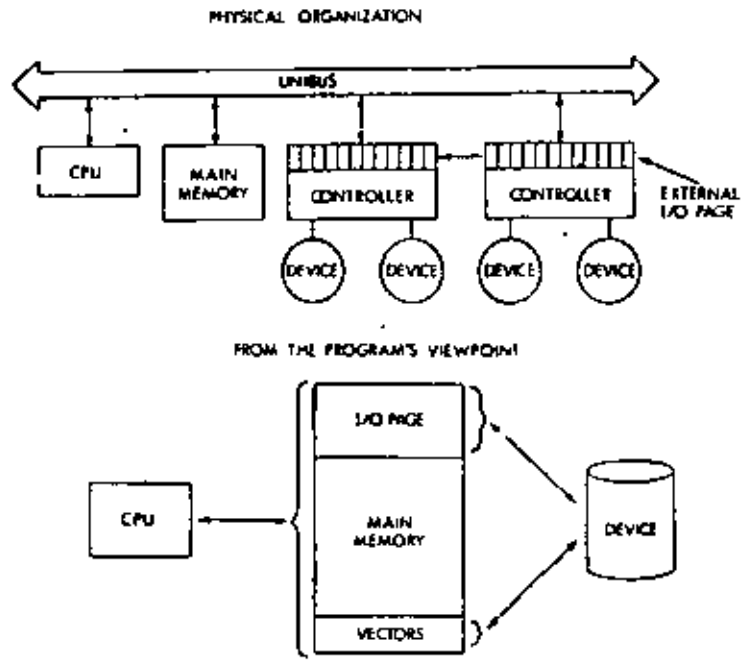


Figure 2-4 Memory and I/O Devices

Although all external devices transmit and receive data through the UNIBUS, devices differ in their ability to store, retrieve or transfer data. Almost all PDP-11 operating systems provide device independence between devices that have similar characteristics and, where possible, between differing devices in situations where the data manipulation operations are functionally identical. Primarily, PDP-11 operating systems differentiate between:

- file-structured and non-file-structured devices
- block-replaceable and non-block-replaceable devices

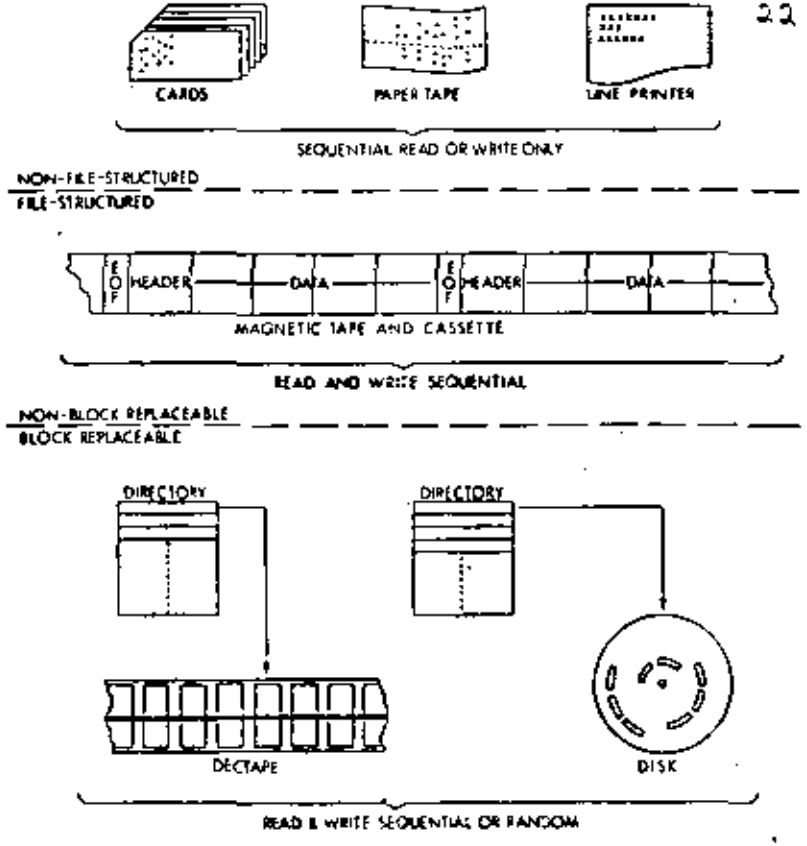
Terminals, card readers, paper tape readers, paper tape punches and line printers are examples of devices that do not provide any means to store or retrieve physical records selectively. They can transfer data only in the sequence in which they occur physically.

In contrast, mass storage devices such as disk, DECtape, floppy disk, magnetic tape and cassette have the ability to store and retrieve physical records selectively. For example, an operating system can select a file from among many logical collections of data stored on the medium.

Mass storage devices are called file-structured devices since a file, consisting of a group of physical records, can be stored on and retrieved from the device. Terminals, card readers, paper tape readers/punches and line printers are called non-file structured devices because they do not have the ability selectively to read or write the physical records constituting a file.

Finally, mass storage devices differ in their ability to read and write physical records. Disk and DECtape devices are block-replaceable devices because a given block can be read or written without accessing or disturbing all the other blocks on the medium. Magnetic tape and cassette are not block-replaceable devices.

A device's physical data access characteristics determine which data transfer methods are possible for that device. Non-file structured devices allow sequential read or write operations only. Non-block replaceable devices allow sequential or random read operations, but allow sequential write operations only. Block-replaceable devices allow both sequential and random read or write operations. Figure 2-5 summarizes the read/write capabilities of each category of I/O device.



File Protection

Master File and User File Directories form the basis for file access protection in multi-user systems. Unauthorized users cannot access a file unless they know the account under which it is stored and can obtain access to that account. Account systems and file access protection techniques are related.

Multi-user systems identify the individuals who use the system by account numbers called User Identification Codes (UIC). The system manager normally gives a user an account number under which the user can log in to the system and obtain access to its services. In general, a UIC consists of two numbers: the first number is used to identify a group of users, the second number is used to uniquely identify an individual user in the group.

In RSTS/E systems, an individual file can be protected against read access or write access where distinctions are made on the basis of the UIC account number under which a file is stored. For example, a file can be read protected against all users who are not in the same account group and write protected against all users except the owner.

The RSX-11/IAS file system provides a protection scheme for both volumes and files. It is possible to specify protection attributes for an entire volume as well as for the files in the volume. A file or an entire volume can be read-, write-, extend- or delete-protected. Distinctions are made on the basis of account number, where the system recognizes four groups of users: privileged system users, owner, owner's group, and all others.

File Naming

The most common way users communicate their desire to process data is through file specifications. A file specification uniquely identifies and locates any logical collection of data which is on-line to a computer system.

A compiler, for example, needs to know the name and location of the source program file that it is to compile; it also needs to know the name that the user wants to use for the output object program and listing files it produces. Most PDP-11 operating systems share the same basic format for input and output file specifications.

In the RT-11 system, a file specification consists of the name of the device on which the file resides, a file name, and a file name extension in the following format:

dev:filnam.ext

The colon is part of the device name, separating it from the file name

on the right. The period is part of the file name extension, separating it from the file name on the left.

PDP-11 operating systems use the same device names for the devices they can access. A device name consists of a two-letter mnemonic and, for multiple devices of the same kind, a one-digit number indicating the device unit number. For example, the name "DK1:" is used to identify the RK11 disk drive unit number 1. The name "DP0:" identifies the RP11 disk drive unit number 0.

In the RT-11 system, a file name is a 1- to 6-character alphanumeric name designated by the user. For example, "SYMBOL", "RL12", and "NORT4" are examples of file names. In the RSTS/E and RSX-11M systems, a file name can be up to nine characters long.

A file name extension is a 1- to 3-character alphanumeric name preceeded by a period. The extension can either be assigned by the user or, if unspecified, assigned by the system. The extension generally indicates the format of a file. System-assigned and recognized extensions make it easy for the user and the system to distinguish between different forms of a file. For example, a file having the extension ".FOR" is recognized by the FORTRAN compiler as a source program written in FORTRAN. A file with the extension ".OBJ" is recognized by the Linker as an object program, a legal input file. When in the process of compiling and linking a FORTRAN program, the user has only to specify a file name to the compiler and Linker. The FORTRAN compiler will compile the file whose extension is ".FOR" and produce a file with the same file name whose extension is ".OBJ". The Linker will link the file whose extension is ".OBJ".

In multi-user systems such as RSTS/E and RSX-11M, a distinction must be made between files stored under various accounts on a device. Two different users can have a file named "REFER.OBJ" on a disk. In these systems, therefore, a file specification has an additional component to identify the user file directory or account under which the file is stored. The basic file specification is expanded to use the following format:

dev:[ufd]filnam.ext

The account number or user file directory is always enclosed in brackets. It consists of the project or group number followed by a comma and a programmer or user number. For example, "[12,4]" is an example of an account or user file directory.

RSTS/E systems also include a protection code as part of the file specification, to indicate the protection that the file receives. A complete RSTS/E file specification could be:

DK1:[200,210]BINFOR.DAT<60>

RSX-11 systems extend the basic file specification format by adding a version number identification after the file name extension. For example, when a file is first created using the editor, it is assigned a version number of 1. If the file is subsequently opened for editing, the editor keeps the first version for backup and creates a new file using the same file specification, but with a version number of 2. A complete RSX-11 file specification could be:

DP0:[15,7]PREPT.MAC;1

In most cases, the user does not have to issue a complete file specification. The PDP-11 operating systems use default values when a portion of a file specification is not supplied. The file name extension defaults, as indicated previously, depend on the kind of operation being performed.

The device name, if omitted, is normally assumed to be the system device. For example, the file specification "FILE.DAT" is equivalent to the specification "DK0:FILE.DAT", if the system device is RK11 drive unit 0. Most systems also allow the user to omit the unit number. If omitted, the unit number is assumed to be unit number 0. For example, DT: is equivalent to DT0:; it signifies DECtape drive unit 0.

If the account number is omitted from the file specification, the system assumes that it is the same as the UIC under which the user logged in or under which the operation is being performed. For example, if the user logged in under UIC 200,200 and issues a file specification "DK3:SAMPL.DAT", it is interpreted as "DK3:[200,200]SAMPL.DAT".

If the version number is omitted from an RSX-11/IAS file specification, the system assumes that the file specification refers to the latest version of the file.

For references to file-structured devices, a file specification must include a file name. The device mnemonics, however, are also used to refer to non-file structured devices. In this case, a file name is irrelevant. For example, an operation to read through a deck of cards and print the information on a line printer is issued in most systems as follows:

#LP:=CR:

The # indicates that an input/output command is being issued; it is printed on the terminal by the program that requests the I/O command. The user types the command LP:=CR:. The = separates the input file specification on the right from the output file specification on the left. The device name LP: signifies that the line printer is to be used as the output device, and the device name CR: signifies that the card

reader is to be used as the input device. A file name, if used, would be ignored, since the system can not symbolically reference data on non-file structured devices.

In addition to relying on defaults in the file specification, the user can also put an asterisk in place of a file name, file name extension, account number, or version number to indicate a class of files. The asterisk convention, also called the wildcard convention, is commonly used in PDP-11 operating systems when performing the same operation on related files. For example, the file specification DP1:[2,1]PROG.* refers to all files on DP1: under account [2,1] with a file name PROG and any extension. The file specification DK:[*,*]FILE.SAV refers to the files under all accounts on RK11 drive unit 0 named FILE.SAV. The file specification DT:*.OBJ refers to all files on the DECtape mounted on drive unit 0 that have the extension .OBJ.

USER INTERFACES

A user interface refers to both the software that passes information between an operator and a system and the language that a system and an operator use to communicate. In the latter sense, a user interface consists of commands and messages. Commands are the instructions that the user types on a terminal keyboard (or gives to a batch processor) to tell the system what to do. Messages are the text that a system prints on a terminal (or line printer) that tells the operator what is going on; for example, prompting messages, announcements and error messages. This section discusses commands, the portion of the user interface that tells the system what to do, and prompting messages, the messages the system prints when it is ready to receive commands.

There are basically four types of commands used in PDP-11 operating systems:

- monitor or command language commands — used to request services from the system as a whole
- I/O commands — used to direct any kind of I/O operation (often a part of monitor commands)
- special terminal commands — these use keys on a terminal for special functions
- system program commands — commands used in system programs that perform operations relevant only for the individual program

Since system program commands are relevant only for individual system programs, and not for operating systems in general, this section discusses monitor and command language commands, I/O commands and special terminal commands only.

Special Terminal Commands

Special terminal commands are a set of keys or key combinations that, when typed on a terminal, are used to perform special functions. For example, a user normally types the carriage return key at the end of an input command string to send the command to the system, which responds immediately by performing a carriage return and line feed on the terminal. The key labeled RUBOUT or DELETE is used to delete the last character typed on the input line.

The most significant special terminal commands are those used with the key labeled CTRL (control). When the CTRL key is held down (like the shift key) and another key is typed, a control character is sent to the system to indicate that an operation is to be performed.

For example, a line currently being entered (whether as part of a command or as text) will be ignored by the system by typing a CTRL/U combination (produced by holding down the CTRL key and typing a U key). The user can then enter a new input line. The CTRL/U function is the same as typing successive RUBOUT keys to the beginning of a line. CTRL/U is standard on PDP-11 operating systems.

Another example is the CTRL/O function. If, during the printing of a long message or a listing on the terminal, the user types a CTRL/O, the teleprinter output will stop. The program printing the output, however, will still continue. The user can type a CTRL/O again to resume output. CTRL/O is a standard function on PDP-11 operating systems.

Physical Device Characteristics and Logical Data Organizations

One of the most important services an operating system provides is the mapping of physical device characteristics into logical data organizations. Users do not have to write the software needed to handle input and output to all standard peripheral devices, since appropriate routines are supplied with the operating system.

There are generally two sets of routines provided in any operating system, depending on its complexity:

- device drivers or handlers
- file management services

Device drivers and handlers can perform the following operations to relieve the user of the burden of I/O services, file management, overlapping I/O considerations and device dependence:

- drive I/O devices
- provide device independence
- block and unblock data records for devices, if necessary

- allocate or deallocate storage space on the device
- manage memory buffers

These routines may exist in the system as part of the monitor or executive, as in RT-11, MUMPS-11, RSTS/E, RSX-11M or RSX-11S, or they may be provided as separate tasks, as in IAS.

An operating system can also provide a uniform set of file management services. For example, the RT-11 system provides file management services through the part of the monitor called the User Service Routine (USR). The User Service Routine provides support for the RT-11 file structure. USR loads device handlers, opens files for read/write operations, and closes, deletes and renames files.

In summary, an operating system maps physical device characteristics into logical file organizations by providing routines to drive I/O devices and to interface with user programs. Figure 2-6 illustrates the transition between the user interface routines and the I/O devices.

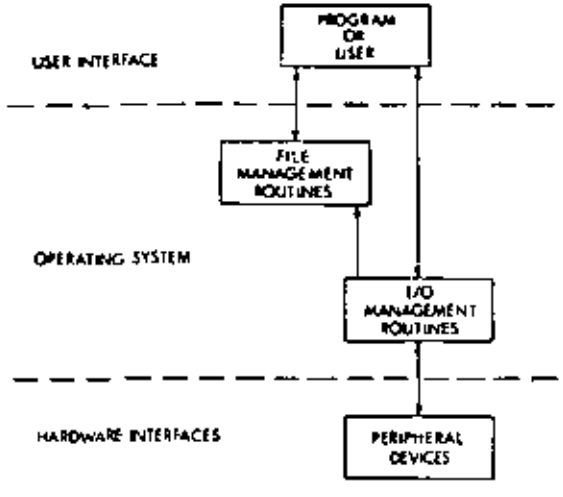


Figure 2-6 Device Control and File Management Services

As an example of the mapping of physical characteristics into logical organizations, the RSX-11 and IAS systems' device drivers and handlers and file management services allow the user application program to treat all file-structured devices in the same manner. All of these devices appear to the user program to be organized into files consisting of consecutive 512-byte blocks which are numbered start-

ing from block one of the file to the last block of the file. In reality, the blocks may be scattered over the device and, in some cases, the device's actual physical record length may not be 512 bytes.

In RSX-11/IAS terminology, the actual physical records on the device (for example, the sectors on a disk) are called physical blocks. At the device driver or handler level, the system maps these physical blocks into logical blocks. Logical blocks are numbered in the same relative way that physical blocks are numbered, starting sequentially at block zero as the first block on the device to the last block on the device. At the user interface level, the operating system maps logical blocks into virtual blocks. Virtual block numbers become file relative values, while logical block numbers are volume relative values.

Figure 2-7 illustrates the mapping between physical, logical and virtual blocks in RSX-11 and IAS systems. The figure shows two disk device types which have different physical record lengths. In this case, the blocks constituting a file are scattered over the disk. The file is a total of 5 blocks long. At the logical block level, the operating system views the file as a set of non-contiguous blocks. At the virtual block level, the user software views the file as a set of contiguous, sequentially numbered blocks.

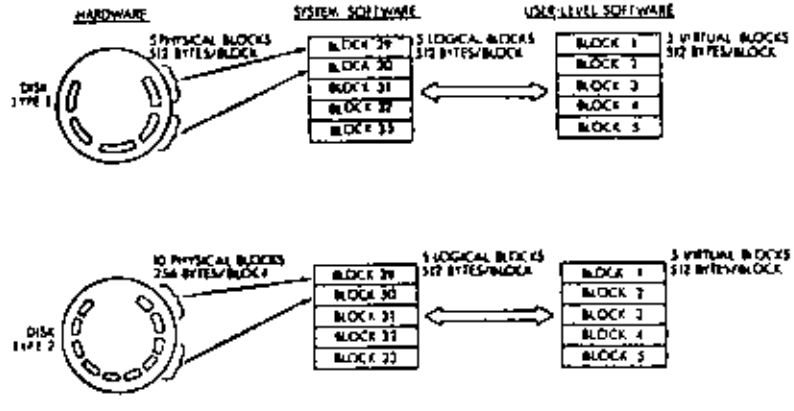


Figure 2-7 Physical, Logical and Virtual Blocks

File Structures and Access Methods

A file structure is a method of organizing logical records into files. It describes the relative physical locations of the blocks constituting a file. The file structure or structures that a particular operating system employs is a product of the way in which the system views the particular I/O devices and the kinds of data processing requirements the system fulfills.

File structure is important because a file can be effective in an application only if it meets specific requirements involving:

SIZE	Growth of the file may require a change in the file structure or repositioning of the file.
ACTIVITY	The need to access many different records in a file or frequently access the same file influences data retrieval efficiency.
VOLATILITY	The number of additions or deletions made to a file may affect the access efficiency.

An access method is a set of rules for selecting logical records from a file. The simplest access method is sequential; each record is processed in the order in which it appears. Another common access method is direct access: any record can be named for the access. The non-block replaceable devices, such as paper tape and magnetic tape, can only be processed sequentially. The block-replaceable devices, such as disk and DECtape, can be processed by either access method, but direct access takes greatest advantage of the device characteristics.

PDP-11 operating systems provide a variety of file structures and access methods appropriate to their processing services. All PDP-11 file structures are, however, based on some form of the following basic file structures:

FILE STRUCTURE	ACCESS METHODS
Linked	Sequential
Contiguous	Sequential or Direct Access
Mapped	Sequential or Direct Access

Linked files are a self-expanding series of blocks which are not physically adjacent to one another on the device. The operating system records data blocks for a linked file by skipping several blocks between each recording. The system then has enough time to process one block while the medium moves to the next block to be used for recording. In order to connect the blocks, each block contains a pointer to the next block of the file. Figure 2-8 shows the format of a linked file.

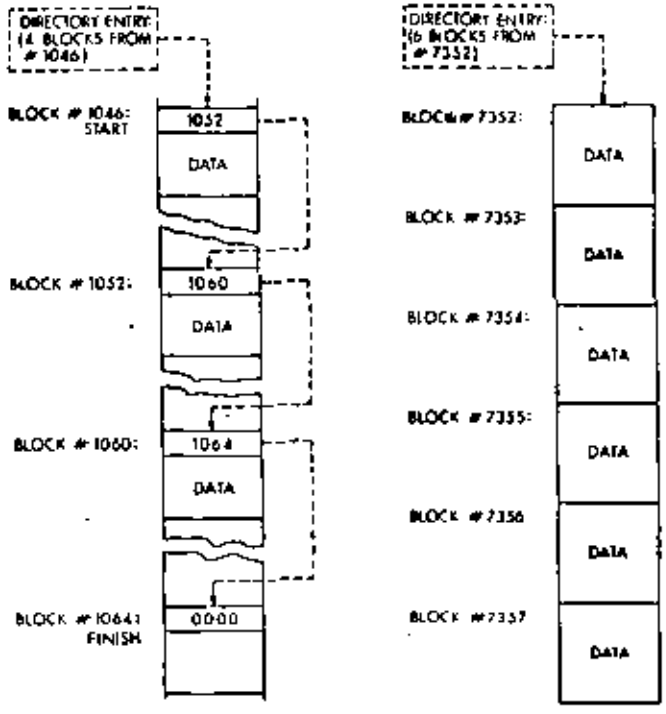


Figure 2-8 Linked and Contiguous File Structures

Linked file structure is especially suited for sequential processing where the final size of the file is not known. It readily allows later extension, since the user can add more blocks in the same way the file was created. In this way, linked files make efficient use of storage space. Linked files can also be joined together easily.

The blocks of contiguous files are physically adjacent on the recording medium. This format is especially suited for random (direct access) processing, since the order of the blocks is not relevant to the order in which the data is processed. The system can readily determine the physical location of a block without reference to any other blocks in the file. Figure 2-8 also shows the format of a contiguous file.

Mapped files are virtually contiguous files; they appear to the user program to be directly addressable sets of adjacent blocks. The files may not, however, actually occupy physically contiguous blocks on the device. The blocks can be scattered anywhere on the device. Separate information, called a file header block, is maintained to identify all the

blocks constituting a file. This method provides an efficient use of storage space and allows files to be extended easily, while still maintaining a uniform program interface. Figure 2-9 illustrates a mapped file format.

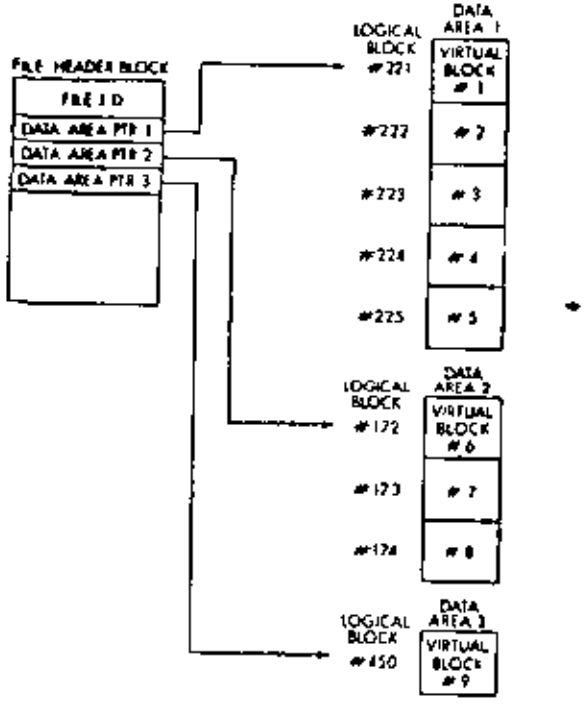


Figure 2-9 Mapped File Structure (Non-Contiguous File)

If desired, a mapped file can be created as a contiguous file to ensure the fastest random accessing, in which case it is both virtually and physically contiguous.

The basic file structures discussed above can be modified or combined to extend the features of each type for special-purpose logical processing methods. Some examples are indexed files and global array files.

Indexed files are actually two contiguous files. One file acts as an ordered map of a second file containing the target data. The index portion or map contains either an ordered list of key data selected from the target data records or pointers to data records in the second

file, or both. The target data records can be processed in the order of the index portion, or the target data records can be selected by searching through the index portion for the key data identifying the records. These methods of logically processing the target data are called indexed sequential access and random access by key, respectively.

Global DSM-11 (MUMPS) array files display a special form of linked file structure. The arrays themselves are a logical tree-structured organization consisting of one or more subscripted levels of elements. All elements on a particular subscripting level are stored in a single chain of linked blocks. At the end of each block in the chain is a pointer to the next block in the chain. The levels of the array (all the block chains) are linked together through pointers in the first block of each chain. This file structure ensures that the time it takes to access any element of the array is minimal.

Directories and Directory Access Techniques

Just as file structure and access methods are required to locate records within files, directory structures and directory access techniques are required to locate files within volumes.

A directory is a system-maintained structure used to organize a volume into files. It allows the user to locate files without specifying the physical addresses of the files. It is a direct access method applied to the volume to locate files.

RT-11 supports the simplest kind of file directory. When disk and tape media are initialized for use, the system creates a directory on the device. Each time a file is created, an entry is made in the directory that identifies the name of the file, its location on the device, and its length. When access to the file is requested thereafter, the system examines the directory to find out where the file is actually located. The system can access the file quickly without having to examine the entire device.

In multi-user systems such as RSTS/E, IAS, and RSX-11M, two different kinds of directories are used to enable the system to differentiate between files belonging to different users. They are the Master File Directory and the User File Directories. These directories are maintained as files themselves, stored on the volume for which they provide a directory.

A Master File Directory (MFD) is a directory file containing the names of all the possible users of a particular device. A User File Directory (UFD) is a directory file containing the names of all the files created by a particular user on a device. The system first checks the Master File

Directory to locate the User File Directory for the particular user, and then checks the User File Directory to locate the file. Figure 2-10 illustrates the use of the Master and User File Directories.

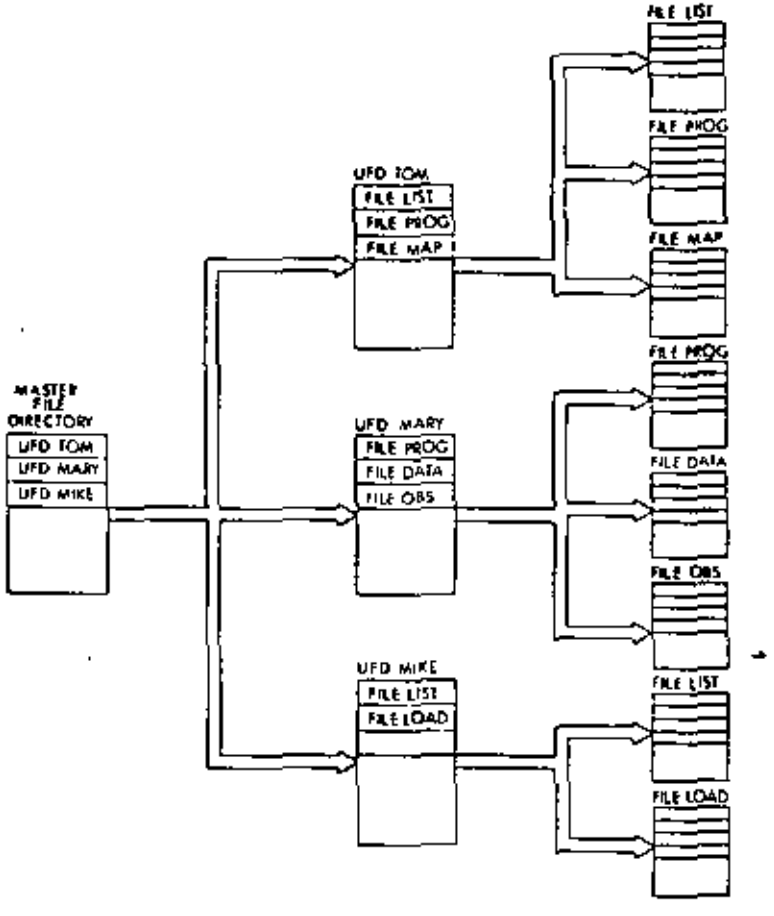


Figure 2-10 Master and User File Directories

RSTS/E creates an MFD on each disk when it is initialized. On all disks except the system disk, the MFD catalogs other user accounts on the disk. The MFD on the system disk has a special purpose, since it maintains a catalog of the accounts under which users can log in, in addition to the user accounts on the disk. A UFD exists on each disk for each account under which files are created. A UFD for an account

is not created until a file is created by the user under that account. DECtape devices are considered to be single-user devices, and the RSTS/E system maintains only a single directory on DECtapes.

The RSX-11M and IAS systems also employ MFD and UFD files on file-structured volumes. As with RSTS/E systems, the number of directory files required depends on the number of users of the volume. For single-user volumes, only an MFD is needed. For multiple-user volumes, an MFD and one UFD for each user are required. An MFD is automatically created when the volume is initialized for use. A UFD is created only by the system manager or priviledged user.

File access in RSX-11M and IAS systems, however, is not limited to using the MFD and UFD files. The basis of file access using the MFD and UFD in these systems is a special file called the index file. Like the MFD, an index file is created on each volume when it is initialized. Files in these systems are mapped files, and the Index File contains the file header for each file stored on the volume, including the MFD. Each file is uniquely identified by a file ID. A file header contains the file's ID and the physical location (logical record number) of each series of contiguous blocks constituting a file. By knowing a file's ID and searching through the index file, a program can locate a file (and any block within the file) without having to use the MFD and UFD directories. Figure 2-11 illustrates how an index file is used to access files on a volume.

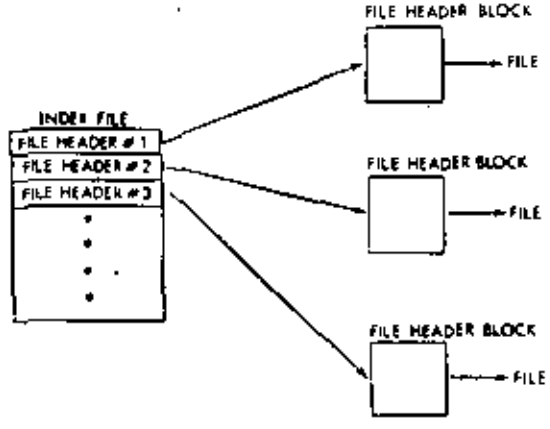


Figure 2-11 Index File Access

All of these operating systems also permit non-block replaceable media, such as cassettes and magnetic tape, to be given a file structure. These media have no directory because a directory could not be updated and replaced. Instead, each file is preceded by one or more header records which contain the directory information such as the file's name. The operating system can locate a file by scanning the volume and reading each file header until the correct one is found. The file can then be processed by a sequential access method.

I/O Commands

As mentioned above, users communicate their intentions to process data files by issuing I/O commands consisting of at least one file specification. Normally, the I/O commands used in a system are standard throughout that system; in addition, most PDP-11 operating systems share the same basic I/O command string format.

For example, in RT-11 systems, the monitor includes a command string interpreter routine that parses and validates I/O command strings. The command string interpreter routine is used both by the monitor and the system programs to obtain a definition from the user of the input file or files to be processed and a definition of the output file or files to be created. User-written programs can also call the command string interpreter to obtain I/O specifications from the operator at a terminal.

A standard I/O command string consists basically of one or more input and/or output file specifications. In all systems except IAS, an I/O command string uses the following general format:

filespec=filespec

where filespec is a file specification and the equal sign (=) represents a character (usually equal sign or less-than sign) that separates an input file specification on the right from an output file specification on the left. If there is more than one input file specification or output file specification, they are separated from each other by commas. For example, if there are two output file specifications and three input file specifications:

filespec,filespec=filespec,filespec,filespec

If the program requesting an I/O command string does not need either an input or output file specification, the equal sign (or less-than sign) is not present in the I/O command string.

As an example, the user can run the RT-11 operating system's Linker system utility to link one or more object program files and produce an executable program file and a load map. The I/O command issued to the Linker could be:

*DK:RESTOR.SAV,DK1:RESTOR.MAP=DK:RESTOR.OBJ/B:500

Where:

*	Is the prompting character printed by the Linker program indicating that it wants an I/O command string. After it is printed, the user types the remaining characters on the line.
DK:RESTOR.SAV	Is the name of the executable program file to be created. It will be stored on the disk cartridge mounted on the RK11 drive unit zero.
DK1:RESTOR.MAP	Is the name of the load map file to be created. It will be stored on the disk mounted on RK11 drive unit 1.
DK:RESTOR.OBJ	Is the name of the object module (input file) to be used to create RESTOR.SAV.
/B:500	Is a command string switch indicating that the RESTOR.SAV program is to be linked with its starting address at location 500.

Command string switches are simply ways of appending qualifying information to an I/O command string. The switches used vary from program to program. They are not usually required in an I/O command string, since most programs assume default values for any switch.

Monitor and Command Language Commands

The primary system/user interface is provided in PDP-11 operating systems by either monitor software or special command language interface programs that run under the monitor. The monitor software and command languages allow the user to request the system to set system parameters, load and run programs, and control program execution.

An input command line consists of the command name (an English word that describes the operation to be performed) followed by a space and a command argument. For example, the command to run a program is the word RUN followed by the name of the file containing the program. If the command name is long, it can usually be abbreviated. For example, the command to set the system's date to August 15, 1984 could be DA 15-AUG-84. The system could also accept "DA 27-AUG-75". A command input line is normally terminated by typing the

carriage return key on the console keyboard, although in some systems the key labeled ALTMODE is also used. Typing the carriage return key (or ALTMODE key) tells the system that the command line is ready to be processed.

In the RT-11 system, a monitor component called the keyboard monitor performs the function of notifying the user that the monitor is ready for input by printing a period at the left margin. The user enters a command string on the same line following the period, and terminates the command string by typing the carriage return key.

In the RSTS/E system, the monitor and the BASIC-PLUS language processor share the responsibility for interpreting commands. The system prints the word READY on the terminal and then spaces down two lines. The user then enters a command on the new line and terminates the line by typing the carriage return key. There are three types of commands the user can issue: RSTS/E monitor commands, such as RUN, ASSIGN, or RENAME; BASIC-PLUS immediate mode statements, such as PRINT, INPUT, or OPEN; or Concise Command Language commands.

A Concise Command Language (CCL) command is used to run and pass arguments automatically to designated programs stored in the system library. The programs can be system utilities supplied with the operating system, or can be user-written console routine programs that perform special application operations. For example, RSTS/E includes a system utility called PIP that performs a variety of file manipulation operations, including a file copy opdration. The dialog normally used to run the PIP utility and issue a copy command is:

READY	The system prints READY.
RUN $PIP	The user runs PIP.
PIP Vnnn	PIP announces itself.
*FILEB.DAT=FILEA.DAT	PIP prints an asterisk to request
*↑C	an I/O command and the user issues a copy command. PIP prints an asterisk, indicating that the operation was performed and that it is ready to accept another command; the user types a CTRL/C to abort PIP and return to the monitor.
READY	The system prints READY.

The standard RSTS/E system also includes a CCL command named PIP that can be issued to perform any of PIP's normal function

used as a CCL command, the dialog to perform the same copy operation is:

READY	The system prints READY.
PIP FILEB.DAT=FILEA.DAT	The user issues the CCL command and the argument that tells PIP to copy FILEA.DAT to FILEB.DAT.
READY	The system prints READY.

A CCL command not only provides an easy-to-use command interface, it can also provide protection from unauthorized use of certain programs. For example, if a particular program performs several operations, some of which should not be available to unauthorized users, the system manager can prevent those users from issuing the RUN command to run the program, but can allow them to perform safe operations by using CCL commands.

In the RSX-11 systems, a command interface called the Monitor Console Routine (MCR) allows the user to perform system level operations. When MCR is activated, it prints the characters MCR> on the terminal. The user enters a command on the same line as the prompt, and terminates the line with a carriage return or an ALTMODE. If the line is terminated with a carriage return, MCR prints a prompt and is ready to receive another command. If the line is terminated with an ALTMODE, MCR does not reactivate. To reactivate MCR at a terminal, the user types a CTRL/C.

There are two kinds of commands that MCR accepts: general user commands and privileged user commands. General user commands provide system information, run programs, and mount and dismount devices. Privileged user commands control system operation and set system parameters.

To run a system utility, the user can type the utility's name in response to an MCR prompt. When the utility is loaded, it prints a prompt to request a command string. The user can then enter a command string. When it completes the operation, the user can enter another command or type CTRL/Z to terminate the program. For example, to run the PIP utility program:

```
MCR>PIP
PIP>command string
PIP>↑Z
MCR>
```

If the user wants to issue only one command to the utility, the user can type the command string on the same line with the MCR request to run the utility. For example:

```
MCR>PIP command string
MCR>
```

In the IAS system, system/user interfaces are provided by programs called Command Language Interpreters (CLI). The standard system includes a CLI called the Program Development System. When it is activated, it prints the prompt PDS> on the terminal to indicate it is ready to accept and process commands. The user has several options for command string formats. If the user is uncertain about a command's syntax, the user can simply type the command name and a carriage return. PDS will ask the user to supply each portion of the command string individually. Users can write their own Command Language Interpreters.

PROGRAMMED SYSTEM SERVICES

All PDP-11 operating systems provide access to their services through requests that programs or tasks can issue during execution.

The RT-11 system provides a variety of programmed requests. There are programmed requests that perform file manipulation, data transfer and other system services such as loading device handlers, setting a mark time for asynchronous routines, suspending a program, and calling the Command String Interpreter. Monitor services are requested through macro instructions in assembly language programs, or through calls to the system library in FORTRAN programs. The basis of the programmed requests in RT-11 are the Emulator Trap (EMT) instructions. When an EMT is executed, control is passed to the monitor, which extracts appropriate information from the EMT instruction and executes the operation requested. When the operation is performed, the monitor returns control to the program.

In the RSTS/E system, users writing BASIC-PLUS programs have access to the monitor's services through system function calls. The function calls allow a program to control terminal operation, to read and write core common strings, and to issue calls to the system file processor. The file processor calls enable a program to set program run priority and privileges, scan a file specification, assign devices, set terminal characteristics, and perform directory operations. A system function is called in a manner similar to normal BASIC-PLUS language calls. When the function operation is performed, the program continues execution.

The RSX-11 and IAS executives include programmed services called

executive directives. Directives can be executed in MACRO programs using system macro calls provided with the system. The FORTRAN or BASIC-PLUS-2 programmer can invoke directives through a subroutine call. The system uses only the EMT 377 instruction to implement directives. The directives allow the program to obtain system information, control task execution, declare significant events, and perform I/O operations. After the directive is processed, control is normally returned to the instruction following the EMT.

The RSX-11M and IAS systems also include programmed file control services. The file control services enable the programmer to perform record-oriented and block-oriented I/O operations. These services are provided as macro calls.

The IAS system includes a special set of programmed services called Timesharing Control Primitives. These are available for use by any program that is written as a Command Language Interpreter (CLI). They enable a CLI to start or control execution of other timesharing tasks, and share access to devices with other timesharing users.

SYSTEM UTILITIES

PDP-11 operating systems provide, in general, three kinds of system utility programs: program development utilities, file management utilities, and special system management utilities.

Most PDP-11 operating systems include the following kinds of program development utilities:

Text Editor — An editor is used for on-line interactive creation and editing of source programs or data files. An editor uses several sets of commands that search for character strings, insert, move or delete characters or lines, and insert, move, delete or append whole buffers of data. Although a text editor is designed for interactive use, it can also usually be run under a batch processor if the operating system supports batch processing.

Assembler — An assembler accepts a source program written in PDP-11 machine language and produces an object module as output.

Linker — A linker is a program that accepts relocatable object programs created by an assembler or compiler and produces an executable program module. Some linkers provide facilities for overlaid program segments to enable a large program to execute in a small memory area.

Librarian A librarian is a program that enables a programmer to create, update, modify, list and maintain library files. A library file is an object module (or modules) that is used several times in a program, used by more than one program, or routines that are related and simply gathered together to incorporate easily into a program.

Debugger A debugger is a program which enables a user to troubleshoot program errors dynamically through a terminal keyboard. It is normally linked with a program and runs as part of the program.

Some of the file management utilities available on many operating systems include:

PIP The Peripheral Interchange Program (PIP) is a general-purpose file utility package for both the general user and programmer and the system manager. PIP normally handles all files with the operating systems standard data formats. In general, the program transfers data files from any device in the system to any other device in the system. PIP can also delete or rename any existing file. Some operating systems include special file management operations in the PIP utility, such as directory listings, device initialization and formatting, and account creation.

FILEX The File Exchange program is a special-purpose file transfer utility similar in operation to PIP. It provides the ability to copy files stored in one kind of format to another format. This enables a user to create data on one system in a special format and then transfer the data to a device in a format that another system can read.

DUMP DUMP displays all or selected portions of a file on a terminal or line printer. In general, DUMP enables the user to inspect the file in any of three modes: ASCII, byte, and octal. In ASCII mode, the content of each byte is printed as an ASCII character. In byte mode, the content of each byte is printed as an octal value. In octal mode, the content of each word is printed as an octal value.

VERIFY In general, a VERIFY program checks the readability and validity of data on a file-structured device.

Most system management utilities included in an operating system are dependent on the function the operating system serves. The RSX-11M, IAS, and RSTS/E systems provide special system management utilities. For example, RSX-11M and RSTS/E include system error logging and report programs. RSTS/E, and IAS and include user accounting programs.

CHAPTER 1

INTRODUCTION TO THE RSX-11M V3.1 OPERATING SYSTEM EXECUTIVE

This introduction is a tutorial for those who are beginning to learn the RSX-11M Executive internal logic. However, this manual assumes that you have at least read and understood the RSX-11M Introduction, the RSX-11M Operator's Procedures Manual, the RSX-11M Task Builder Reference Manual, and the RSX-11M System Generation Manual. If you are familiar with the RSX-11M Executive or you are an experienced system programmer, you may want to begin this manual with Chapter 2, which assumes that you have a basic knowledge of the Executive and describes the memory structures of RSX-11M.

1.1 RSX-11M SYSTEM

RSX-11M is a real-time operating system. This means that RSX-11M responds quickly to input conditions or input data. RSX-11M is also a multiprogramming system. This combination allows real-time activity (for example, process control) to occur along with program development (interactive terminals) and other user jobs. At one extreme, RSX-11M can be a dedicated process control system, and at the other, a system for developing and running applications programs.

1.2 SYSTEM GENERATION

RSX-11M offers a wide range of services and utilities from which to choose. Each installation selects from these options to shape its version of RSX-11M according to the processor and peripherals available and the purpose of the system. You perform a system generation (SYSGEN) process to select these options.

Every installation intitially receives an RSX-11M system on distribution media. You run this system and use its resources to generate a target system configured to your installation's needs.

System generation is done in two phases. During the first phase SYSGEN defines and assembles the Executive (the kernel or "brain" of the operating system that responds to external requests) by conducting a dialogue with you. Query programs pose questions at a terminal. Your answers to the questions determine the Executive service options, processor options, and peripheral devices to be incorporated into the system. During the second phase, SYSGEN builds the Executive, allows you to define memory structures called partitions, and builds and installs the system programs.

You complete the SYSGEN process by saving and bootstrapping the new system. Saving a system means writing the image of an RSX-11M system that has been resident in main memory into the system image file from

which it was bootstrapped. You do this with the Save command, which saves the image to allow a hardware bootstrap or the Boot. command to later reload and restart the system.

You bootstrap (boot) a system by either using the switches on the processor control panel or using the Boot command. The Boot command bootstraps a system that exists as a system image file on a Files-11 (the RSX-11M file structure) volume. The Boot command immediately terminates the system in operation and starts another. The Save command, the Boot command, and the process of booting a system with the switches are all described in the RSX-11M Operator's Procedures Manual.

To change either the hardware or software configuration of an installation, you must perform another system generation. The RSX-11M System Generation Manual describes the system generation process in detail.

1.3 MAJOR COMPONENTS OF RSX-11M

RSX-11M requires the organized interaction of the following components:

- Memory resource management. Memory is the processor storage medium in which loaded user programs, the Executive , and control blocks of data reside. Much of the Executive's work involves memory resource management and control.
- Task scheduling and processing. Tasks are system or user programs that perform needed functions and manipulate data to achieve some goal. The Executive controls task processing and handles specific requests issued by the tasks.
- Interrupt processing. The Executive processes synchronous and asynchronous events that occur as a result of task processing. Examples of these events include software errors, I/O completion, illegal instructions, and power failure.

1.4 MEMORY

1.4.1 Memory Partitions

A partition is a continuous area of memory in which executable programs called tasks can be run. The typical memory organization consists of an area for the Executive and areas for system- or user-controlled partitions. A partition has the following characteristics:

- A name
- A defined size
- A fixed base address
- A defined type

1.4.2 Partitions In Mapped And Unmapped Systems

RSX-11M runs on almost all models of the PDP-11 processor. The PDP-11 addressing scheme allows a program to address directly only 32K words of memory. For larger memories, DIGITAL has a KT-11 Memory Management Unit (hardware) available for all models of the PDP-11 except the PDP-11/03/04/05/10/20 processors. The KT11 Memory Management Unit associates addresses expressed in programs ("virtual" addresses in the range 0 to 32K) with actual locations in memory ("physical" addresses). Physical addresses can range from 0 to 124K words on all processors other than the PDP-11/70. Physical addresses on a PDP-11/70 can range from 0 to 1920K words.

Mapping is the process that associates virtual addresses with physical addresses. Therefore, a PDP-11 system that includes a KT11 Memory Management Unit is called a mapped system. Conversely, systems without a KT11 are called unmapped systems. In a mapped system, a task can be installed in any system partition or user partition large enough to contain it. In an unmapped system, the task is bound to physical memory and must be installed in the partition that starts at the same memory address as the partition for which it was built.

Whether a system is mapped or unmapped affects the way in which you create tasks. Before a compiled program (object code) can be run, it must be processed by the Task Builder program (linker). The Task Builder produces a task image that runs in a memory partition.

If a system is unmapped, you must specify to the Task Builder the base address of the partition in which the task is to be run. You cannot run the resulting task in a partition that has a base address different from the address you specified to the Task Builder.

In a mapped system, however, every task (other than a privileged task mapped into the Executive) has a virtual base address of 0. Transparently to the user, the KT11 maps the virtual addresses of a task to the actual physical addresses in which the task resides. A task in a mapped system can therefore run in any partition large enough to contain it.

You need not rebuild nonprivileged tasks in a mapped system when physical partition boundaries move. This is true because nonprivileged tasks on a mapped system run at a virtual base address of 0, rather than at a physical base address.

If you move the symbols that are referenced in the code, you must rebuild privileged tasks in either system because they are linked to the Executive symbol table file. You may be required to rebuild nonprivileged tasks only if you change any of the task's attributes such as checkpointability. The task's attributes can be changed when you use the Install command to install the task. You use the Task Builder to establish the attributes when building a task. Consult the RSX-11M Task Builder Reference Manual for a comprehensive discussion of task attributes and associated Task Builder switches. See the RSX-11M Operator's Procedures Manual for a description of the Install command.

1.4.3 Partition Types

RSX-11M supports two types of partitions in which tasks can execute:

1. System-controlled
2. User-controlled

In a system-controlled partition, the Executive allocates available space to accomodate as many tasks as possible at any one time. This allocation may involve shuffling resident tasks to arrange available space into a continuous block large enough to contain a requested task. The Shuffler, which is a privileged task and a SYSGEN option, shuffles the tasks and memory space to make the needed space for the requested task. Only mapped systems support system-controlled partitions.

A user-controlled partition is exclusively allocated to one task at a time. Both mapped and unmapped systems support this type of partition.

1.4.4 Subpartitions

You can subdivide a user-controlled partition into as many as seven nonoverlapping subpartitions. Like its parent main partition, a subpartition can contain only one task at a time. Because the subpartitions occupy the same physical memory as the main partition, tasks cannot be simultaneously resident in both the main partition and one of its subpartitions. However, because each subpartition can contain a task, up to seven tasks can potentially run in parallel within a main partition.

The purpose of subpartitioning is to reclaim large memory areas in unmapped systems. For example, when a large task that requires a main partition is either no longer active or can be checkpointed (written out to a disk to make room for a higher priority task), subpartitioning allows a number of smaller tasks to use the partition space.

1.4.5 Memory Structure

RSX-11M memory in a typical system can be divided into the following parts:

The Executive, which consists of:

- Trap vectors. The trap vector area contains the hardware and interrupt vectors; it requires 128 words. During SYSGEN, you can expand this area to 256 words.

- System stack. The system stack area is an internal storage area for Executive use. The Executive uses it for nesting interrupts, saving registers and data, and internal calls. The stack requires 60 to 110 words depending upon options selected at system generation time.

- System common data. This area contains system pointers that are filled in during system generation and used by the Executive and privileged tasks during execution.

- The Executive code. The Executive coordinates and manages system resources and processes specialized system functions. System generation options determine the size and abilities of the Executive.

- Dynamic Storage Region (DSR). The Executive continually uses temporary storage in memory. The Executive acquires, uses, and then returns the memory that it used to the available memory pool. If a given Executive service routine requests dynamic storage and it is unavailable, the Executive informs the user task, which usually waits for some memory to become available. The size of this region is important. If it is too small, long waiting periods or system deadlocks can occur. If it is too large, fewer tasks can fit into the remaining memory. The size of the region is a system generation parameter.

  You can extend the initial allocation of dynamic storage on line by issuing the MCR command, Set /Pool, from the console. However, the use of this command is limited in that this expansion can occur only into space that is not being used. This space, if it exists, is between the Dynamic Storage Region space and the first partition of memory.

- Device drivers:

  You can include three drivers in the 8K Executive during SYSGEN:

  1. A disk driver
  2. A cassette, DECtape, magtape, line printer, or floppy disk driver
  3. A terminal driver

  In general, Executives larger than 8K contain additional resident drivers which you include during system generation. Some drivers can be made loadable; that is, they reside on disk and are loaded into memory when they are needed. Therefore, loadable drivers save memory space because they occupy memory only when needed and they do not occupy Executive virtual address space.

- Loader:

  The Loader is a task that runs in its own partition, which is resident in the Executive. Thus, it can run in parallel with system and user tasks. The Loader, which is device independent:

  1. Loads tasks upon initial load requests
  2. Writes checkpointable tasks to disk when required (see checkpointing, in this chapter)
  3. Reloads previously checkpointed tasks when memory becomes available, allowing them to actively compete for processor resources.

- MCR and TKTN tasks:

  - The Monitor Console Routine (MCR) processes system commands that you enter at a terminal. These commands are directed to the MCR processor. MCR either executes the commands itself, or activates a system or user-written task that can service the commands.

- The Task Termination Notification Task (TKTN) performs two functions:

  1. It prints out messages and tries to print the contents of the registers of a task that has been aborted due to an error.

  2. It prints out messages for device drivers.

  Ideally, TKTN runs either in a partition in which all tasks are checkpointable or execute quickly, or in its own partition. The reason for this is that TKTN must be in memory in order to print messages. If TKTN cannot get memory space to execute, the Executive queues up messages to TKTN, thereby using up Dynamic Storage Region space. It is conceivable that all the Dynamic Storage Region could be used up for this purpose; this would cause the system to hang up.

- The file system:

  Files-11 is a system of formatting files that are held on volumes. Files-11 volumes are magnetic media (tapes or disks) that have been specially formatted by the MCR command, Initialize Volume. Volumes that are not properly formatted are considered to be "foreign." RSX-11M includes a file exchange utility that translates files in DIGITAL's DOS or RT-11 format into Files-11 format.

  Your tasks that run on RSX-11M access data within files on Files-11 volumes through the use of two sets of subroutines:

  - File Control Services (FCS)
  - Record Management Services (RMS)

  Both FCS and RMS provide the ability for your tasks to perform record- or block-I/O operations on Files-11 volumes. FCS and RMS are system interfaces between the I/O programs that you write and the files on the Files-11 volumes that you want to access. These interfaces provide device independence and allow you to take advantage of different methods of file organization.

  FCS imposes a single logical organization on your files. This logical organization is called the sequential file organization and FCS imposes it on all files regardless of medium.

  In contrast to FCS, RMS provides three file organizations - sequential, relative, and indexed.

  The MACRO-11 I/O programming that you do differs between FCS or RMS. Therefore, you must become familiar with the contents of the manuals that describe each one. The respective manuals are:

  For FCS:

  - IAS/RSX-11 I/O Operations Reference Manual
  - RSX-11M I/O Drivers Reference Manual

For RMS:

- Introduction to RMS-11
- IAS/RSX-11M RMS-11 MACRO Programmer's Reference Manual

The Files-11 Ancillary Control Processor (F11ACP) is a group of Executive subroutines that process and control the I/O control structures and devices for RMS or FCS. The Executive, FCS, and RMS use F11ACP; however, its operation is transparent to you.

F11ACP is available in three versions. The first and smallest (FCPNMH.TSK) requires 2K of memory. FCPNMH does not support multi-header files or RMS record blocking. The second (FCP.TSK) requires 2.5K of memory. The third version (BIGFCP.TSK) requires from 4.5K to 8K of memory. You select these versions during SYSGEN. The RSX-11M System Generation Manual fully describes these versions, the reasons for their use, and the methods of installation.

- The print spooler:

  The print spooler task (PRT) speeds up the operation of MACRO-11, the Task Builder, and compilers because they do not have to wait for I/O to complete on the relatively slow line printer. Instead, the listing files are written to a disk. Subsequently, PRT prints the files as they appear in a queue.

  Any task that uses the line printer to print files may use the print spooler. For example, the RSX-11M Peripheral Interchange Program (PIP) can optionally use the PRT task to print files.

- User task partitions

  User tasks run in the remaining memory in the memory structure. The partitions and tasks can be configured to the system user's requirements.

1.4.6 Example Of A 16K Unmapped System

Figure 1-1 illustrates the memory layout of a sample 16K unmapped system. The Executive region, which requires 8K, consists of the Executive and the user-controlled main partition named SYSPAR. This partition contains the file system (FCPNMH), the Monitor Console Routine (MCR), and the Task Termination Notification routine (TKTN). The file system is checkpointable and has a lower priority than MCR or TKTN. Therefore, if the file system is running and a system user requests MCR, the Executive checkpoints the file system and loads and starts MCR.

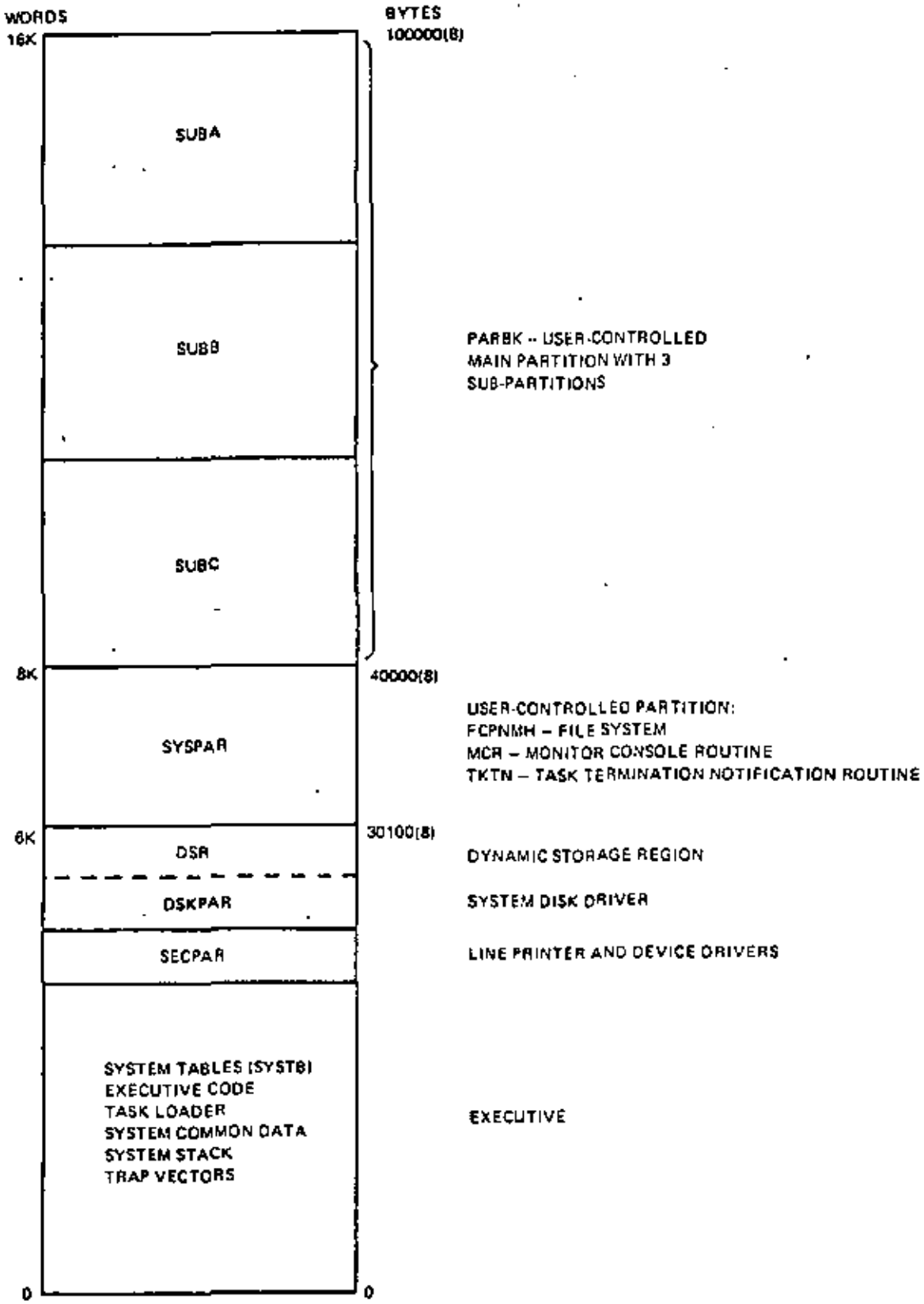


Figure 1-1 Sample Unmapped 16K System Memory Layout

The user area contains a user-controlled main partition named PAR8K, 8K in length. PAR8K contains three subpartitions, named SUBA, SUBB, and SUBC. Language processors and the Task Builder use the 8K partition for program preparation. These programs usually have a low priority and may be checkpointable.

The three subpartitions are available for real-time tasks. A task in the main partition is checkpointed if:

- It is checkpointable
- Another higher priority task needs the partition, or a subparition

If tasks occupy the partitions SUBA, SUBB, SUBC, and SYSPAR and the tasks are ready to run, the Executive gives CPU resources to the task with highest priority.

1.4.7 Example Of A Mapped 124K-word RSX-11M System

Figure 1-2 is an example of a large mapped system.

Besides the Executive, the system contains DRVPAR, which is a system-controlled partition for loadable device drivers including the terminal driver. Loadable drivers residing on a disk are loaded by a user command when they are needed.

SYSPAR is a 2K user-controlled partition that contains the Monitor Console Routine Multi-user (MCRMU) task, TKTN, and the Shuffler task. The Shuffler is discussed later in this chapter.

FCPPAR is a 6K partition for the primary file control system, BIGFCP. The 6K size is sufficient to allocate approximately 50 file control blocks (FCBs).

All other tasks run in the system-controlled GEN partition.

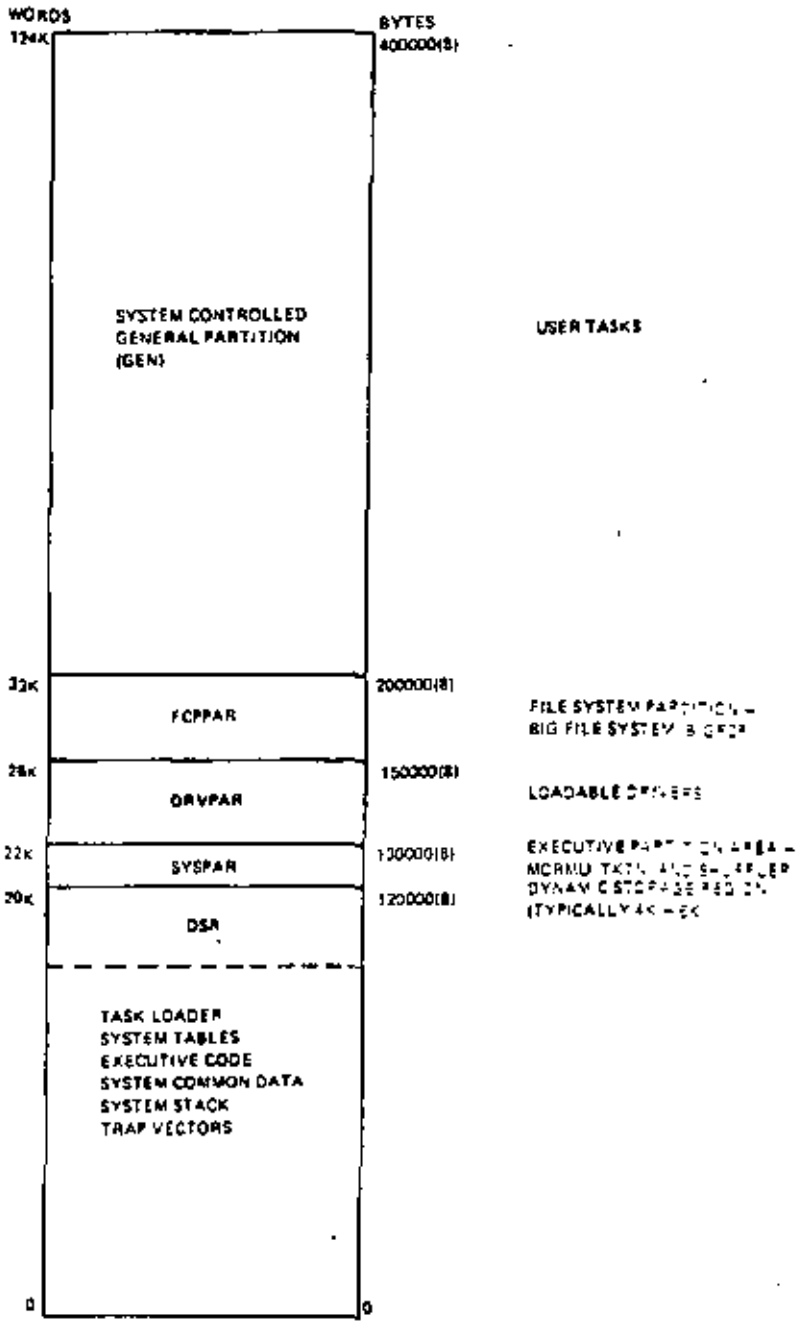


Figure 1-2 Example of a Mapped 124K RSX-11M System

1.5 TASK PROCESSING

To make a task known to the Executive, you install it. When you install a task (by issuing the Install command; see the RSX-11M Operator's Procedures Manual), the system records a number of task parameters in a system-resident table called the System Task Directory (STD). The recorded parameters include the name and size of the task, the disk address at which the task's image starts, and the address of the Partition Control Block (PCB) of the partition in which the task is to run.

1.5.1 Task States

An installed task is defined as a task that has an entry in the STD. It is neither resident in memory nor competing for system resources. The Executive considers it to be dormant until a running task or a command issued from a terminal requests the Executive to activate it. The Executive therefore recognizes two task states:

- Dormant. A dormant task is one that has been installed (has an entry in the STD), but has not been requested to run.
- Active. An active task is an installed task that has been requested to run. It remains active until it exits, or gets aborted. It then returns to the dormant state.

  An active task can be in one of two substates, ready-to-run or blocked.

  1. Ready-to-run. A ready-to-run task competes with other tasks for CPU time on the basis of priority. The ready-to-run task having the highest priority obtains CPU time and thus becomes the current task.
  2. Blocked. A blocked task is unable to compete for CPU time for synchronization reasons or because a needed resource is not available.

The distinction between dormant tasks and active tasks is important in a real-time system. A dormant task uses little memory; and yet when the task is needed to service a real-time event, the Executive can quickly and efficiently introduce it into active competition for system resources. An installed task's STD entry enables this quick response because it contains all the parameters the system needs to retrieve the requested task. Note that the number of installed, dormant tasks can, and usually will, far exceed the number of active tasks.

When the Executive receives a request to activate a dormant nonresident task, it performs a series of actions:

- It allocates the required memory resources on the basis of the task's priority.
- It brings the task into memory.
- It places the task in active competition for system resources with other resident tasks.

If tasks fully occupy the partition in which a task is installed and no resident task can be checkpointed (see Checkpointing in this chapter), the task is placed in a queue by priority with other

activated tasks, each waiting for space to become available in its partition.

1.5.2 Fixed Tasks

A task can be loaded and locked into its partition. Such a task is called a fixed task. The Fix command (see the RSX-11M Operator's Procedures Manual) allows you to fix a task in memory. The Executive services subsequent requests to run the task more quickly because the task is resident in memory and does not have to be loaded from the disk before it can run. The system can fix a task in memory only when the partition in which it is to be fixed becomes available.

Fixed tasks remain physically in memory even after they finish execution. Therefore, the Executive does not have to reload them when they are again requested to run. However, tasks that can be fixed in memory must have re-entrant code if it is to be reused by another program or system user. Re-entrant code must be used because the task cannot be allowed to change its own internal data base if another program uses it. Only an Unfix or Remove command can free the memory that the task occupies.

NOTE

See the RSX-11M Operator's Procedures Manual for the Fix, Unfix, and Remove commands.

The following restrictions apply to tasks that you want to fix in memory:

- You must first install the task.
- You cannot fix an active task.
- You cannot fix a checkpointable task.

1.5.3 Priority

Active tasks compete for system resources on the basis of their relative priorities. The Executive gives control of the processor to the active task that has the highest priority and that also has access to all the other resources it needs. When this task becomes blocked (while waiting for I/O to complete, for example), the Executive looks for another task to use the processor. The chosen task is again the one that has the highest priority and has access to all the resources it needs.

You initially assign a default priority to a task when you task-build it. This priority is a number between 1 and 250 (decimal); higher numbers indicate higher priority. Later, you can change the priority when you install the task; or the system can change the priority while the task is running (see Swapping).

In an RSX-11M installation that mixes real-time applications with less urgent work, higher priority numbers should be assigned to the real-time tasks. This assignment ensures that the Executive gives processor time to the real-time tasks ahead of the others. Text

editors are an example of real-time tasks, because they must respond within a short time period. Text editors, commonly used for program development or text processing, spend a large part of their time waiting for terminal I/O to complete and are therefore out of competition for processor time. However, when the I/O operation ends, the terminal needs a rapid response. To get the higher response, the installation system manager can assign to text editors a higher priority than that of more processor bound tasks like the Task Builder or Assembler.

1.5.3.1 **Establishing Task Priority** - You can establish task priority when you use the Task Builder to build a task from an object module. See the RSX-11M Task Builder Reference Manual for a description of the priority option.

1.5.3.2 **Installed Priority** - When you install a task using the Install command, you can specify a priority different from the one that you specified when you built the task. The priority specified in the Install command overrides the priority that was specified for the task in the Task Builder command. See the RSX-11M Operator's Procedures Manual for a complete description of the Install command.

1.5.3.3 **Altering Priority** - You may want to alter a task's priority after it is installed. The Alter command provides a way to change priority. With the Alter command you can change the task's static installed priority or change the task's running priority. However, you can make these changes only if the system supports the Alter Priority directive. See the RSX-11M Operator's Procedures Manual for a complete description of the Alter command.

1.5.4 Round-robin Scheduling

When numerous competing memory-resident tasks have equal priorities, the Executive tends to give processor time more often to those tasks that appear first in the System Task Directory (STD) queue. Entries with equal priorities normally appear in the STD in the order in which the tasks were installed. Therefore, the Executive favors tasks that were installed first. To avoid this problem, RSX-11M provides a system generation option called round-robin scheduling. Round-robin scheduling uses an algorithm that periodically rotates the execution of tasks of equal priority in the STD. The overall effect is that processor time is distributed more evenly among tasks. Each equal-priority task has its turn toward the head of the STD queue.

1.5.5 Checkpointing

In a programming system where many tasks of equal or different priorities are competing for memory space and system resources, the Executive must have a method of distributing processor usage and resources to all the tasks. The RSX-11M Executive uses a process called checkpointing to allocate system resources among tasks. The Executive uses task priority as the basis for the checkpointing scheme.

In some instances, an active task cannot get into memory and compete for processor resources because the partition in which it was installed is fully occupied. If the partition contains a task that has a lower priority and is checkpointable, the Executive moves that task out of memory and writes it on a disk to make room for the higher priority task. When the high priority task is finished, the Executive reloads the low priority task, which is now on the disk, to allow it to continue processing from the point at which it was interrupted. This roll-out, roll-in process is called checkpointing.

RSX-11M supports checkpointing in both user-controlled and system-controlled partitions. The objective is to avoid preempting a lower priority task, unless a higher priority task can be brought in to make use of the freed memory. This optimizes the use of system resources while maintaining a priority scheduling discipline.

1.5.5.1 Disk Space for Checkpointing - To checkpoint a task, checkpoint space equal to the size of the partition that contains the task must be available on disk. (Checkpoint space contains the checkpointed task while a higher priority task executes.) You can allocate checkpoint space either statically when building the task, or dynamically at run time. You can use both kinds of checkpointing to balance the advantages and disadvantages of the different allocation methods.

When you use the Task Builder to create a task from an object module, you can request checkpoint space allocation in the task image file on the disk; this is the same disk as the one on which the task resides. The task image file is the executable task on the disk. While the task is running, its checkpoint space is always allocated on disk, whether or not the Executive actually checkpoints the task.

You can use disk space more efficiently if you allocate checkpoint space dynamically. Instead of reserving disk space equal to the size of each checkpointable task, you can create one or more checkpoint files on disk to contain all checkpointed tasks. The size of the files depends on an estimation of the checkpoint space required at any given time. When the system allocates checkpoint space dynamically, tasks need not be built as checkpointable. Instead, you decide if a task can be checkpointed when the task is installed. You create a checkpoint file, independent of individual tasks, by issuing the ACS (Allocate Checkpoint Space) command from the terminal. Then, when the Executive needs to checkpoint a task, it writes the task out into the available space in the checkpoint file. A drawback to dynamic allocation of checkpoint space is that space in a checkpoint file may, at times, be filled. However, system performance may be improved if the checkpoint file is on a fast disk.

See the RSX-11M Operator's Procedures Manual for the Allocate Checkpoint Space (ACS) command.

1.5.6 Task Swapping

The Executive must deal with the situation that occurs when several active tasks with equal priorities compete for partition space in memory. A task cannot normally cause the Executive to checkpoint another task with the same priority. Therefore, a task of equal priority cannot get into memory. The Executive includes a task swapping algorithm that uses checkpointing to allow tasks of equal priority to successfully compete for memory.

Swapping is a variation of checkpointing that enables the Executive to checkpoint tasks with equal priorities in and out of memory. Swapping does not work, of course, unless the tasks are checkpointable. When an eligible task begins to run, the Executive adds a number to the task's normal running priority. This number is called the swapping priority and is used for swapping only. The old running priority still exists. As the task runs, the Executive decrements the swapping priority. Eventually, the sum of the decremented swapping priority and the task's running priority causes the running task to have a priority (for swapping) less than that of a competing task. When this occurs, the Executive checkpoints the running task to make room for the competing task. The Executive then places the checkpointed task in the queue of active tasks that are competing for memory. The swapping priority does not affect task scheduling or I/O dispatching, which are governed solely by the task's running priority.

1.5.7 The Shuffler Task

In trying to accommodate the execution of as many tasks as possible, the Executive moves tasks in and out of memory depending upon available space, priority, etc. This operation can result in fragmented memory, a situation in which many small tasks occupy memory with unused spaces in between. Taken individually, these spaces may not be large enough to allow large tasks to be loaded and executed. The Shuffler task, a system generation option, solves this problem by performing memory compaction in a system-controlled partition.

The Shuffler starts at the beginning of the system-controlled partition and tries to move (shuffle) all tasks that are sitting above a gap of free space down to the base of the free space. When possible, it also checkpoints any tasks that it encounters that are waiting for terminal input.

If there are some tasks still actively competing for memory in the partition, the Shuffler creates an ascending, priority-ordered list of the tasks in the partition. If the sum of the free space now in the partition and the space occupied by the low priority, checkpointable tasks in the partition is enough to allow the waiting task to run, the Shuffler checkpoints the lower priority tasks. The Shuffler then compacts memory again to make room for the waiting task.

The foregoing Shuffler action should result in all free space being at the top of the partition. However, there may be additional holes below tasks because some things (drivers and regions) cannot be shuffled. These additional holes cannot be reclaimed.

1.5.8 Extended Logical Address Space

An RSX-11M task specifies an address in a 16-bit word. The largest address that can be expressed in a 16-bit word is 65,536 bytes or 32,768 words (commonly referred to as 32K words). To avoid limiting the effective size of a task to only 32K words, a task can use overlays that you define when you use the Task Builder to build the task. Another option is that the task can use memory management directives to access greater amounts of memory.

1.5.8.1 Overlays - An overlaid task has parts called segments. The segments are the parts that overlay one another. The segments are also sometimes called overlays. The root segment, which is always in memory and never overlaid, and one or more overlay segments compose an overlaid task. The overlay segments can be read into memory as required. However, all the segments in memory at one time cannot exceed 32K words.

See the RSX-11M Task Builder Reference Manual for a complete description of overlay segments.

1.5.8.2 Memory Management Directives - Memory management directives allow task segments resident in memory to access more than 32K words of physical memory. The memory management directives, a subset of the Executive directives, use the KT11 hardware to map task addresses to different logical areas within the task. Instead of displacing task segments in memory, the task can reside entirely in memory and map its virtual addresses to different physical addresses.

RSX-11M defines three kinds of address space:

- Physical address space. Physical address space consists of the physical memory in which tasks reside and execute.
- Logical address space. Logical address space is the total amount of physical address space to which the task has access rights.
- Virtual address space. Virtual address space corresponds to the 32K of addresses that the task can explicitly specify in a 16-bit word. If a task does not use memory management directives, its logical and virtual address spaces directly correspond one to the other. However, if the task uses these directives, it can map its virtual addresses to different parts of its logical address space. The net effect is to allow a task's logical address space to exceed 32K.

The memory management directives also allow a task to expand dynamically its logical address space. In other words, a task can access logical areas that are not part of its static task image (the executable task produced by the Task Builder). A task can issue directives that create a new region of logical space and then map a range of virtual addresses to the newly created region. A task can also map its virtual addresses to logical areas that belong to another task. The mapped area then becomes part of the former task's logical address space.

The ability to create and map to a new region allows tasks to communicate with one another by means of shared regions. For example, at run time a task can create a new region of logical space, into which it writes a large amount of data. Any number of tasks can then access that data by mapping a range of their virtual addresses to the region. Another benefit of mapping to different regions is an ability to use a greater number of common routines. Tasks can map to the required routines at run time, rather than link to them when the tasks are built by the Task Builder.

1.6 RSX-11M INTERRUPT PROCESSING

The RSX-11M system recognizes two kinds of hardware interrupts: processor traps and external interrupts. Processor traps occur synchronously; that is, the same sequence of instructions causes the same processor trap to occur at the same place and time in the program. Processor traps usually have a cause originating from within the processor. See Processor Traps, in this chapter, for the causes of processor traps. External interrupts, which are usually caused by I/O devices, are asynchronous in that they may occur anywhere or at any time in the program's execution.

Programs that use input and output routines would spend most of their time waiting for I/O devices to complete their operations if it were not for the program interrupt facility of RSX-11M. The program interrupt facility allows asynchronous events, such as I/O completion, to interrupt the running program so that a routine can service the interrupting device. An interrupt is analagous to a subroutine jump. However, to preserve program integrity, interrupts are allowed to occur only after the completion of an instruction and before the start of the next instruction.

As an example of the program interrupt facility, programs can continue operation after starting a device, then allow the device to interrupt when it is ready to signal the program about its resulting status.

The addresses of the interrupt processing routines must be made known to the Executive. These addresses are called interrupt vectors and they are in the Executive's low memory area.

1.6.1 Interrupt Vectors

Each peripheral device controller in the RSX-11M system has a hardware pointer to its own pair of memory words. These words are located in the low memory of the Executive. One word contains a vector (address) for the device's interrupt service routine. The vector may be an entry point address or an Interupt Control Block address. If this vector is an entry point address, it becomes the contents of R7 the program counter or PC word) when the service routine begins its execution. For loadable drivers, the vector points to an Interrupt Control Block.

The other word is the processor status (PS) word. It contains the mode and the priority of operation for the interrupt routine. The hardware saves the status of the interrupted program (the PC and PS) before the interrupt routine begins its processing.

The Executive has an area called a stack in which it saves status, register contents, parameters, or any other data that it may need.

1.6.2 System Stack

RSX-11M maintains a push-down stack using general register 6, which is the stack pointer or SP. External interrupts, subroutine calls, and processor traps use this stack to save program status. When an interrupt occurs, the hardware first saves the current processor status word (PS) and the program counter (PC) on the stack. It then uses the new PS and PC from the trap and interrupt vector area in low memory, and begins processing the interrupt routine that handles that particular interrupt. A return from interrupt (RTI) instruction restores the original PS and PC values from the stack, thereby restoring the original interrupted program.

1.6.3 Processor Traps

A variety of errors and programming conditions cause the processor to trap to a set of fixed locations. These locations contain the PC and PS for the trap processing routines. Processor traps include the following:

- Power failure
- Odd addressing errors
- Stack errors
- Timeout errors
- Non-existent memory errors
- Memory parity errors
- Memory management violations
- Floating point processor exceptions
- Use of reserved instructions
- Use of the T-bit in the PS word
- Use of the IOT, EMT, and TRAP instructions

Processor traps cannot be masked off. That is, when they occur, the processor immediately enters the trap sequence of pushing the current PS and PC onto the current stack, retrieving the new PS and PC from a specific hardware trap vector, and executing the code that begins at the location specified by the trap vector.

Although there are several processor traps (see Interrupt and Trap Vectors, below), the trap of main interest is the emulator trap. The EMT instruction causes the emulator trap. This instruction calls the Executive whenever a user task has an Executive directive written into it that requests the Executive to perform some specific function (see Executive Directives below).

1.6.4 External Interruptions

External interrupts are hard-wired into one of four priority levels of the processor (labeled 4 to 7, with 7 being the highest priority). These interrupts are maskable in that they can cause an interrupt only if the priority level held in the processor status word is less than the priority of the interrupting source. When an interrupting device causes a new priority level to be loaded from its vector PS word, interrupts at the same or lower levels are blocked out. The system, however, remembers that the interrupts occurred and it processes them in turn by priority.

Certain traps, however, cannot be masked by the priority field in the PS word. These traps are: parity error, memory management violation, stack limit yellow, power failure (power down), and floating-point exception.

1.6.5 Interrupt And Trap Vector Locations

The following chart shows some of the interrupt and trap vectors used by RSX-11M interrupt and trap processing. The PC for the interrupt routine is taken from the specified memory location. The next word contains the new PS word.

Memory Location	Interrupt and Trap Vector
000	Reserved for DEC use
004	CPU errors
010	Illegal and reserved instructions
014	Breakpoint trap (BPT)
020	Input/output trap (IOT)
024	Power fail
030	Emulator trap
034	TRAP instruction
.	
.	
.	
244	Floating-point error
250	Memory management

For a complete list of vectors, see the pertinent PDP11 Processor Handbook.

1.6.6 System Traps

System traps are transfers of control (also called software interrupts) that provide tasks with another means of monitoring and reacting to events. The Executive initiates system traps when certain events occur. The trap transfers control to the task associated with the event and gives the task the opportunity to service the event by entering a user-written routine.

There are two distinct kinds of system traps:

- Synchronous System Traps (SSTs). SSTs detect events directly associated with program instruction execution. They are "synchronous" because they always occur at the same point in the program when previous instructions are repeated. For example, an illegal instruction causes an SST to occur.
- Asynchronous System Traps (ASTs). ASTs detect significant events that occur asynchronously to the task's execution; that is, the task has no direct control over the precise time that the event occurs. For example, the completion of an I/O transfer may cause an AST to occur.

To use the system traps, a task issues system directives that establish entry points for user-written service routines. Entry points for SSTs are specified in a single table. AST entry points are set by individual directives for each kind of AST. When a trap occurs, the task enters the appropriate routine via the specified entry point.

Debugging aid programs (On-line Debugging Tool and Executive Debugging Tool) can be entered from points, which are called breakpoints, that you insert into a memory-resident task. These breakpoints cause a breakpoint trap that transfers execution to the debugging aid program. The debugging aid, by means of its own table of trap vectors, can

execute special processing for certain SSTs that can occur. The IAS/RSX-11 ODT Reference Manual discusses the On-line Debugging Tool in detail. The Executive Debugging Tool (XDT) is described in the RSX-11M Guide to Writing an I/O Driver.

1.7 EXECUTIVE DIRECTIVES

An Executive directive is a request from a task to the Executive to perform an indicated operation. A programmer uses Executive directives to control the execution and interaction of tasks.

Executive directives enable tasks to perform functions such as the following:

- Obtain task and system information
- Measure time intervals
- Perform I/O operations
- Manipulate a task's logical and virtual address space
- Suspend and resume execution of tasks
- Request the execution of another task
- Exit from a task

System directives allow tasks to exploit some major system functions, including the following:

- Event flags
- System traps
- Extended logical address space

RSX-11M MACRO programs execute Executive directives by using macro calls and the EMT 377 instruction. FORTRAN uses DIGITAL-supplied library routines to use the directives.

You should always use macro calls instead of directly executing the directive. Then, if system changes are made in the directive specifications, you need only to reassemble the program rather than edit the source code.

Listed below is a brief summary of the directive functions that are possible for RSX-11M. For a complete description of RSX-11M Executive directives, see the RSX-11M Executive Reference Manual.

Task Execution Control Directives

Abort Task	Causes the Executive to terminate the execution of the task named in this directive.
Cancel Time Based Initiation Requests	Causes the Executive to cancel all time-synchronized initiation requests for the execution of the task named in this directive, regardless of the source of each request.

Task Exit	Informs the Executive that the task issuing the Exit has completed its execution. Unless the exiting task is fixed, its memory is freed for use by other tasks.
Extend Task	Causes the Executive to modify the size of the task that issues this directive by a positive or negative number of 32-word blocks.
Request Task	Causes the Executive to request immediate execution of the task named in the directive.
Resume Task	Causes the Executive to resume the execution of a task that has issued a Suspend directive.
Run Task	Causes the Executive to schedule the execution of the task named in this directive at a time specified in terms of a time period from the issuance of the directive.
Suspend	Causes the Executive to suspend execution of the task that issued the Suspend until explicitly resumed, either by a Resume directive from another task or the MCR command, Resume.

Task Status Control Directives

Alter Priority	Causes the Executive to change the running priority of the installed and active task named in this directive.
Disable Checkpointing	Causes the Executive to make the task that issues this directive no longer checkpointable.
Enable Checkpointing	Causes the Executive to nullify the previously issued Disable Checkpointing directive.

Informational Directives

Get Partition Parameters	Causes the Executive to fill a 3-word buffer, which is specified in this directive, with parameters related to the memory partition specified in this directive or related to the task that issues this directive.
Get Region Parameters	Causes the Executive to fill a 3-word buffer, which is specified in this directive, with region parameters.
Get Sense Switches	Causes the Executive to return the settings of the 16 console switches to the task that issues this directive.

Get Task Parameters	Causes the Executive to fill a 16-word buffer with parameters related to the task that issues this directive.
Get Time Parameters	Causes the Executive to return the current time parameters (year, month, day, hour, minute, second, tick and ticks/second) of the task.

Event-associated Directives

Clear Event Flag	Causes the Executive to clear an event flag specified in the directive and return the previous polarity of the flag.
Cancel Mark Time Requests	Causes the Executive to cancel MARK TIME requests that have been made by the task that issues this directive.
Declare Significant Event	Causes the Executive to declare a significant event. The Executive scans the STD for the highest priority task capable of execution. It then saves the context of the currently executing task and starts the execution of the new highest priority task.
Exitif	Causes the Executive to cause an exit of the task that issues the directive if, and only if, a specified event flag is clear.
Mark Time	Causes the Executive to declare a significant event after the expiration of the time interval specified in the directive. If an event flag is specified in the directive, it is cleared when the directive is issued and set when the significant event occurs. If an Asynchronous System Trap (AST) entry point address is specified in the directive, an AST occurs at the time of the significant event.
Read All Event Flags	Instructs the Executive to return to the task that issued this directive the polarities of all 64 event flags in a 4-word buffer.
Set Event Flag	Causes the Executive to set an indicated event flag and return the previous polarity of the indicated flag (without a declaration of a significant event).
Wait For Significant Event	Causes the Executive to suspend the execution of the task that issues the directive until the next significant event occurs.

Wait For Logical Or Of Event Flags	Causes the Executive to suspend the execution of the task that issues the directive until one or more specified event flags of a group of event flags is set.
Wait For Single Event Flag	Instructs the executive to suspend the execution of the task that issues the directive until an event flag that is specified in the directive is set.

Trap-associated Directives

AST Service Exit	Causes the Executive to terminate the execution of the AST service routine.
Disable AST Recognition	Causes the Executive to disable AST recognition for the task that issues this directive. The ASTs are queued and only their recognition is inhibited.
Enable AST Recognition	Causes the Executive to enable AST recognition for the task that issues this directive.
Specify FPP Exception AST	Informs the Executive that the specified AST routine within the task is to begin execution whenever a floating-point processor exception occurs, or that floating-point processor exception ASTs are no longer wanted.
Specify Power Recovery AST	Informs the Executive whether or not power recovery ASTs are wanted for the task that issues this directive. If the ASTs are wanted, this directive indicates where control is to be transferred when the AST occurs.
Specify Receive Data AST	Informs the Executive whether or not receive data ASTs for the task issuing this directive are wanted. If the ASTs are wanted, task execution is transferred to the address of the AST service routine within the task when data is placed in the task's receive queue.
Specify Receive By Reference AST	Informs the Executive to transfer control to an address in the task specified in the directive when the Receive-by-Reference AST occurs, or that receive-by-reference ASTs are no longer desired for the task that issued this directive.
Specify SST Vector Table For Debugging Aid	Specifies the address of a table of synchronous system trap service routine entry points for use by ODT or other debugging aids.

Specify SST Vector Table For Task	Informs the Executive that the task that issues this directive contains a table of addresses of service routines to be executed upon task trap or fault conditions.

I/O and Intertask Related Directives

Assign LUN	Causes the Executive to assign a physical device unit to a logical unit number (LUN). The LUN, device name, and device unit number are specified in this directive.
Connect To Interrupt Vector	Causes the Executive to allow a task to process hardware interrupts by a routine specified in the directive. The Interrupt Service Routine (ISR) must be in the task's own space.
Get LUN Information	Causes the Executive to return information regarding the logical unit specified in the directive to the task that issues this directive.
Get MCR Command Line	Causes the Executive to transfer an MCR (terminal) command line to the task that issues this directive.
Queue I/O Request	Causes the Executive to queue an I/O request for the issuing task. This request is queued by priority for a logical unit which is assigned to a physical unit. An event flag, an AST, and an I/O status block may be specified as I/O completion indications.
Queue I/O Request And Wait	Similar to the queue I/O request directive except for one aspect. The Queue I/O Request And Wait directive specifies an event flag and the Executive executes an implied Wait For Single Event Flag directive.
Receive Data	Informs the Executive that the task that issues this directive is ready to receive data (in a 13-word data block) that has been sent from another task by means of the Send directive.
Receive Data Or Exit	Causes the Executive to attempt to receive data (dequeue a 13-word data block) for the task that issues this directive. If no data is received, the task that issues this directive exits.
Send Data	Causes the Executive to declare a significant event and to queue the 13-word block of data that the task named in this directive is to receive.

Memory Management Directives

Attach Region	Causes the Executive to attach the task that issues this directive to a static common region or to a named dynamic region.
Create Address Window	Causes the Executive to create a new virtual address window by allocating a window block from the header of the task that issues this directive and establishing the window's virtual address base and size.
Create Region	Causes the Executive to create a dynamic region in a system-controlled partition and, as an option, attach it to the task that issues this directive.
Detach Region	Causes the Executive to detach the task that issues this directive from the previously attached region that is specified in this directive.
Eliminate Address Window	Causes the Executive to delete an existing address window, unmapping it first if necessary.
Get Mapping Context	Causes the Executive to return a description of the current window-to-region mapping assignments.
Map Address Window	Causes the Executive to map an existing window to an attached region.
Receive By Reference	Causes the Executive to dequeue the next packet in the receive-by-reference queue of the task that issues this directive.
Send By Reference	Causes the Executive to insert a packet containing a reference to a region into the receive-by-reference queue of a receiver task that is specified in this directive.
Unmap Address Window	Causes the Executive to unmap the window that is specified in this directive.

1.7.1 Event Flags

The execution of certain directives causes significant events to occur. In fact, most significant events are caused, either directly or indirectly, by system directives.

A significant event occurs when a task issues a system directive that implicitly or explicitly suspends a task's execution, or when an external interrupt occurs that can affect a task's execution. Event flags are associated with significant events. When a significant event occurs, the event flag indicates the specific cause of the significant event.

The Executive uses significant events and event flags to manage task execution. However, tasks can also use significant events to coordinate internal task activity and to communicate with other tasks. For example, a task can issue an Executive directive to associate an event flag with a specific significant event. When that event occurs, the Executive sets the associated flag. Therefore, by testing the state of the flag, a task can determine whether or not the event has occurred.

Sixty-four event flags are available to enable tasks to distinguish one event from another. Each event flag has a corresponding event flag number. The first 32 flags are local to each task and are set or cleared as a result of each task's requirements. The second 32 flags are common to all tasks and are therefore called global or common event flags. Global flags can be set or cleared as a result of any task's operation. Tasks use global flags to communicate with other tasks because one task cannot refer to another task's local flags. Eight of the local event flags and eight of the common event flags are reserved exclusively for the Executive.

1.8 THE MCR INTERFACE

You communicate with RSX-11M by entering commands at a terminal. The terminal driver directs the commands to the Monitor Console Routine (MCR) processor. The MCR processor either executes the commands itself, or it activates a system or user-written task that can service the commands.

MCR commands allow you to:

- Start up the system
- Manage peripheral devices
- Control task execution
- Obtain system and task information
- Activate system or user-written tasks that request input from the terminal

The MCR commands that control task execution are particularly significant to system performance. You must use an MCR command (Install) to install a task into the system. Therefore, you establish the base of installed tasks, which the Executive, other installed and active tasks, and further MCR commands can manipulate.

1.8.1 Privileged Commands

To restrict the use of commands that directly affect system performance, RSX-11M considers some MCR commands and command options to be privileged. You can issue a privileged command only from a privileged terminal. In multiuser protection systems, individual users are either privileged or nonprivileged; when a user logs on, the terminal assumes the privilege status assigned to that user's identification code (UIC). A user can issue an MCR command at a privileged terminal to modify the privilege status of any other terminal connected to the system. If multiuser protection support is not included during system generation, all terminals are privileged.

1.8.2 External Scheduling Of Task Execution

An important MCR function is the external scheduling of task execution. This type of scheduling works in conjunction with the Executive's priority driven internal scheduling of active tasks. You can include time parameters with the command that activates an installed task. The time parameters request the Executive to run a task:

- At a specified time from the current moment
- At a specified time from clock unit synchronization
- At an absolute time of day
- Immediately

All of these time options are available with or without periodic rescheduling. RSX-11M also supports an unlimited number of programmed timers for each task in the system. The user task can create its own timer, which the Executive then decrements at regular intervals. When the timer reaches zero, the Executive sets an event flag or generates an Asynchronous System Trap (AST) that passes control to the task at a prespecified address.

1.9 TERMINAL OPERATION

In RSX-11M, a variable number of terminals can operate concurrently. In addition, each terminal operates independently of others in the system to allow each to run a different task. In a system that supports multiuser protection, a user must log onto a terminal before issuing further commands. In other RSX-11M systems, a user can issue commands whenever the terminal displays an appropriate prompt.

1.9.1 Attached Terminals

RSX-11M allows tasks to request input from a terminal. To ensure that a requesting task receives input intended for it, the task usually attaches to the terminal. While the task is attached, the terminal directs all input to the attached task, with one exception. The exception is a control C character (the C key pressed while pressing the CTRL key), which gains the attention of the MCR processor. An attached terminal ensures that a soliciting task properly receives its input; but it also allows a user to interrupt the task's control of the terminal to communicate with MCR. Note that attaching to the terminal is a function of the task rather than of a user.

Some applications may require that a user be denied access to MCR but allowed access to a specific task only. In this case, a task can attach to the terminal with a special subfunction. The subfunction causes the system to generate an AST for the attached task whenever someone enters unrequested input, including CTRL/C, at the terminal. However, making the terminal a slave terminal is another way of doing this.

1.9.2 Slave Terminals

When your installation needs to dedicate a terminal exclusively to one or more tasks, you issue an MCR command (or a task issues a special I/O function) that sets the terminal to slave status. The difference between a slave terminal and an attached terminal is that the system ignores all unsolicited input, including CTRL/C, that is entered at a slave terminal. Until you issue another MCR command to delete the slave status, the terminal can only be used to communicate with the task soliciting input from the terminal. An I/O function issued by a task can also delete the slave status of the terminal. Slave terminals are often dedicated to real-time applications.

1.10 MULTIUSER PROTECTION

Multiuser protection, a system generation option, allows an RSX-11M installation to monitor and control individual users of the system. Individual users are either privileged or nonprivileged. The system manager, who is the one assigned responsibility for system configuration and operation, assigns a user identification code (UIC) to each user, which determines the user's privilege status. When logging onto a terminal, the user supplies a last name or UIC and a password. If the user gives a name, the system finds the associated UIC. The system then checks that the password matches the last name or UIC, and sets the terminal to privileged or nonprivileged status, according to the user's UIC.

1.10.1 Public And Private Devices

In a multiuser protection system, some commands allow you to do things that are not allowed in systems without multiuser protection. For example, the Allocate command allows you (or any user) to allocate a device (a disk drive) as your private device; allocating the device prevents other nonprivileged users from accessing it.

A nonprivileged user can access a private device that he has allocated to perform MCR functions that are normally privileged. These functions include preparing a disk or magnetic tape for use by the RSX-11M file system.

To complement the private device feature, multiuser protection allows the system manager or privileged user to declare certain devices to be public. Public devices cannot be allocated to individual users. By declaring a line printer to be public, for example, the system manager can ensure that all users have access to that commonly used output device.

1.11 SYSTEM MAINTENANCE

1.11.1 Error Logging

RSX-11M provides an error logging facility as a system generation option for systems that are 24K words or larger. The error logging facility monitors the hardware reliability of an RSX-11M system; it continually detects and records information about disk, DECtape,

magtape, and memory errors as they occur, regardless of whether or not the error is recoverable. The Executive automatically retries recoverable errors. However, you might be unaware that the error occurred. Therefore, at user-determined intervals, a formatting task can be run to generate individual error and summary reports on some or all of these errors.

Please note that only the following four types of errors are loggable:

- Device errors (disks, magtapes, DECtapes)
- Undefined interrupts
- Timeout
- Memory parity errors

In summary, the error logging facility performs the following functions:

- Detects a hardware error as it occurs (done by Executive modules)
- Gathers information about the error
- Stores the information in a file
- Formats the information to produce an error report

Control of the facility is shared between routines in the Executive and specific error logging tasks. These routines and tasks interface with each other to carry out the four operations described above.

You can generate a wide variety of error reports. Among many options, you can specify a report that covers only a certain time period, a certain device or group of devices, or perhaps a certain type of error. You can also request a report that contains only information on individual errors, one that contains only summary information, or one that contains both kinds of statistics.

Because the error log files may be written to a removable volume, an operator can generate the reports either on site or at any other RSX-11M installation that supports the error logging facility.

1.11.2 Diagnostic Tasks

RSX-11M also provides a group of diagnostic tasks which you can incorporate into the Executive support at system generation time. A diagnostic task tests a specific device to identify the source of any errors. RSX-11M diagnostic tasks test for malfunctions on most disks, DECtapes, magnetic tapes, and terminals. The tasks are simple to use and require little memory space.

When used in connection with error logging reports, the diagnostic tasks can significantly reduce system downtime. The system manager should regularly generate error reports to check on hardware performance. When a number of errors indicates that a particular device is beginning to malfunction, the manager can run the diagnostic task for the erring device to help isolate the source of the errors.

Each diagnostic task has two modes of operation: customer mode and service mode. In customer mode, the user activates the appropriate

task, which then runs to completion and reports its findings. (Because the tests destroy any data resident on the device being tested, only authorized users should be allowed to run diagnostic tasks.) Service mode is intended for use by DIGITAL Field Service engineers. Service mode allows the user to modify the test content initially and to interrupt the running test to make further modifications.

1.11.3 Power Failure Restart

RSX-11M can execute a power failure restart that smooths out intermittent short-term power fluctuations with little loss of service or data. Power failure restart functions in four phases:

- When power begins to fail, the CPU traps to the Executive, which stores volatile register contents, thereby bringing system operations to a controlled halt.

- When power is restored, the Executive again receives control and restores the preserved state of the system.

- The Executive then schedules all device drivers that were active at the time of the power failure at their power-fail entry points. Drivers have the option of being scheduled one of two ways:

  1. Whenever power fails

  2. Only when power fails while the driver is servicing an I/O request

  The drivers can then make any necessary restorations of state (repeat an I/O transfer, for example).

- The Executive then determines if any user-level tasks have requested notification of power failure by issuing a system directive requesting an AST on power recovery. The Executive initiates ASTs for any tasks that have requested them.

REFERENCIAS

1. Holt-Graham-Lazowska-Scott, "Structured Concurrent Programming with Operating - Systems Applications", Addison-Wesley - 1978.

2. Madnick-Donovan, "Operating Systems", Mc Graw Hill 1974.

3. Per Brinch Hansen, "Operating System - Principles", Prentice Hall, Inc.

4. Dionysios C. Tsichritzis y Philip A. - Bernstein, "Operating Systems", Academic Press.

5. "PDP-11 Software Handbook", Digital - Equipment Corporation.

6. "RSX-11M System Logic Manual", Digital Equipment Corporation.

Directorio de Alumnos del Curso Introducción a las Minicomputadoras
1980.

1. Héctor Javier Arrona Urrea
   Centro de Cálculo
   Facultad de Ingeniería
   UNAM
   550.52.15 Ext. 4150

   Fco. Morazán 219
   M. Balbuena
   México DF
   768.06.62

2. Jesús René Acosta Chavira
   Aluminio y Vidrio Ruiz, SA.
   Insurgentes Sur 1864
   México 20, D.F.
   548.60.61

   Perseo 149
   Prado Churubusco
   Z.P.13
   582.06.38

3. Silvia Aguilar Morales
   PEMEX
   Marina Nal. 329
   Z.P.17
   531.57.56

   Penitenciaria 12-4
   Morelos
   Z.P.2
   789.15.64

4. José Guillermo Almeida Reynoso
   S C T
   Subsecretaría de P. y Marina Mercante
   Insurgentes Sur 465
   México DF
   564.57.28

   Troncoso y del Paso 226 D 15
   Z.P.9
   552.29.07

5. Jorge Alvarado Trejo
   Comisión de Desarrollo Urbano del D.F.
   Fdo. Alva Ixtlixochitl 175
   Col. Tránsito
   Z.P. 1
   542.10.21

   Fresno 116
   Sta. Ma. la Ribera
   Z.P.4

6. Octavio Aranda Espinosa
   Cía. Contratista Nal, S.A.
   COCONAL SA
   Periférico Sur 6501
   Tepepan, Xochimilco
   Z.P.23
   6765555 Ext. 105

   Convento de Tepozotlán 45
   Los Reyes Iztacala, Edo. de Méx.
   537.88.78

7. Miguel Alejandro Arenas Rodríguez
   COCONAL S.A.
   Periférico Sur 6501
   Z.P.22
   6765555 Ext. 116

   Relojeros 46-1
   El Retoño
   Z.P.13
   532 94 09

8. Roberto Astudillo Sarabia
   PECK SA
   Calz. de las Aguilas 101-4°
   Las Aguilas
   Z.P.20

   Calle 13 # 150
   I. Zaragoza
   Z.P.9

9. Yolanda Patricia Avila Azúa
Alberto J. Pani 112
Cda. Satélite, Edo. de Méx.
562 79 19

10. María Elena Barraza Gutiérrez
SAHOP
Analista de Precios Unitarios
Xola y Av. Universidad
Z.P.12
530 33 36

Mitla 113
Narvarte
Z.P.12
519. 76 06

11. Juan Rafael Callejas Castañeda
SAHOP
Dir. Gral. de Carr. Federales
México 12, DF

12. Rubén Carranza Vázquez
CÍa. Contratista Nal, S.A.
Periférico Sur 6501
Z.P.23
676 55 55

Edif. Charango Depto. 202
V. Panamericana
Z.P. 21 655 07 05

13. Carlos Córdova S.
Televisa SA
Blvd. Adolfo L.Mateos 232
Z.P.20
595.74.69

Paseo Sta. Clara 53
Jardines S. a. Clara
Ecatepec, Edo. de Méx.

14. Conrado Dávila García
D D F
Jefe de la Secc. de Ana. de Sist.
Sn. A. Abad 231-7°
Z.P.8
588 32 27

Miguel Laurent 61-202
Z.P.12

15. Arturo de los Santos Díaz
Tlevisa SA
Blvd. A. L. Mateos 232
Z.P.20
595 74 69

Limones 60 Mza. 94 L.51
Jardines Ojo de Agua
Edo. de Méx.

16. José de Jesús de los Santos y del Angel
Banco de Mex SA
Insurgentes Sur 2375
Z.P.20
550 70 11

Laguna de Guzmán 119-5
Z.P.17

17. Rubén Díaz
Cerro Gordo 168
C.Churubusco
Z.P. 21
549.71.19

Ing. B. Romo A 21
Coo. Industrial
México D F
537.34.29

18. Sixto Domínguez Martínez
S A R H
Clavijero 19-3°
Xalapa, Ver.
7 53 37

H. De nacozari 242-3
Veracruz, Ver.
2 25 49

19. José Luis Estrada Bolaños
Comisión Nal. de la Ind. Azucarera
Av. Morelos 104
Z.P.1

Valle de Toluca 39
El Mirador
Edo. de Méx.
560. 02 25

20. Gonzalo Flores Girón
Serv. Profesionales de Ing. SA
Melchor Ocampo 445
Z.P.5
525 02 09

Río Verde 44
Fracc. Viveros del Río
Tlanepantla
398 30 34

21. María del Carmen Gil Rivera
Centro Latinoamericana de Tecnología
Educacional para la Salud AC.
Presidente Carranza 162
Z.P.21
554 86 55

Sur 111 # 635
Sector Popular
Z.P.13

22. Edmundo Godínez V
Televisa SA
Blvd. A.L. Mateos 232
Z.P.20
595 74 69

Casahuete 9
Unifam. INFONAVIT
Iztacalco
Z.P.8
657 98 93

23. Sergio Gómez de la Barrera
D D F
Cerro Gordo 168
C. Churubusco
Z.P. 21
549 71 19

Platanales 174
Nva. Sta. Ma.
México D F
556 65 37

24 Juan David Gómez Melo
Evatepec Automotriz
Vía Morelos 169 Km 23.5
Carr. Mex. Pachuca
Ecatepec, Edo. de Méx.
569 81 00

Cerro de la Libertad 398
C. Churubusco
Z.P. 21
549 03 39

25 Rolando González Soto
Comisión de Desarrollo Urbano
Fdo. de Alva Ixtlixochitl 175
Z.P.12
5 22 04 61

Torres Adalid 1977
Z.P.12
5 79 95 98

26 Víctor Manuel Hernández González
Servicios Profesionales de Ing. SA
Melchor Ocampo 445
Z.P.5
525.02 09

Sur 111 # 335
Sector Popular
Z.P.13

27 Mauro Iñíguez Covarrubias
S A R H
Reforma 45 10°
México 1, D.F.
592.01 08

Bucareli 101-11
Col. Juárez
México DF
592 38 49

28 Ernesto Iñíguez González
UAM
Unidad Xochimilco
México, D.F.

Av. Tres # 72-5
Sn. Pedro de los Pinos
Z.P.18
277 64 01

29 Leonel Hernández Hernández
U A G
Esc. de Ing'.
Av. de la Juventud s/n
Chilpancingo, Gro.

18 de Marzo No. 5
Chilpancingo, Gro.

30. Antonio Irigoyen Reyes
SAHOP
Av. Universidad y Xola
Z.P.12
519.27 70

Edificio D 11 Depto. 701
Torres de Mixcoac
Z.P.19
593 35 32

31 José D. Izquierdo Buenrostro
COCONAL SA.
Periférico Sur 6501
Z p. 22
676 55 55

Condor 156
Col. Alpes
Z.P.20
593 56 86

32 Luis Alejandro Fraustro Jiménez
D D F Analista de Sistemas
S. A. Abad 231
Z.P. 8
588 28 06

Andrés Molina Enríquez 761
S. A. Tetepilco
Z.P. 13
532 34 47

33 Rafael José Salín
C M. La Raza, IMSS
Jacarandas y Vallejo
México, DF.
583 63 66 Ext 2106

Londres 25-1
Col. del Carmen
Z.P. 21
689 04 98

34 Jorge Humberto Lozano Medina
Dir. Gral. de Const. y Operación Hidráulica
D D F
S. A. Abad 231-7°
588 32 27

Calle Central E 31-21
U.H.Alianza Popular Revolucionaria
México 21, D.F.

35 Javier Mardueño S.
Cía. Minera Autlán S.A. de C.V.
Mariano Escobedo 510
Z.P.5
250 19 77

Thiers 251-9°
Z.P. 5
250 17 99

36 Vilfulfo Rolando Martínez Navarrete
Televisa SA
Blvd. A. L. Mateos 232
Z.P. 21
595 74 69

Emiliano Zapata 93
Sto. Domingo
Z.P.21
597 73 99

37 José Luis Millán Vargas
PACK SA
Las Aguilas 101-4
Z.P.20
593 63 00

Esperanza 28
Vergel
México 13, DF
593 63 00

38 Gabino Gaspar Monterrosa Reyes
D D F
Dir. Gral. de Const. y Ope. H.
S.A.Abad 231-7°
Z.P.8
588 32 27

Saratoga 1017
Portales
Z.P.13
532 46 59

39 Eric Moreno Mejía
Facultad de Ing.
Profesor
UNAM
548 96 69

Chiapas 202
Z.P.7
584 82 65

40 Eduardo A. Nandayapa Gómez
S P I Ingenieros SA de C V
Melchor Ocampo 445
Z.P.5
525 02 90 Ext.61

Pennsylvania 31-13
Coyoacán
Z.P.21
549 83 32

41 Germán Naranjo Torres
Tesorería D D F Depto. de Soporte Técnico
Niños Héroes y Dr. Lavista
Z.P.7
761 22 66

Sur 75 # 4423
Col. V.Piedad
Z.P. 13
538 33 62

41 Carlos Mario Ocampo Cano

42 Armando Ocaranza García
Bco. de México S A
Insurgentes Sur 2375
México 20, D.F.
550. 70. 11 Ext. 251

Av. Sánchez Colpin 40
Ahuizotla , Edo. de Méx.
576. 12.65

43 Carlos Peregrina Ramírez
Libros Mc Graw Hill de Méx. S A de CV
Atlacomulco 499 y 501
Col. Ind. S. Andrés Atoto
Naucalpan
576 90 44 Ext 123

Alcázar 26-8
Col. Tabacalera
Z.P.1
591 07 07

44 Arturo Ponce de León Huerta
Dir. Gral. de Planificación
Plaza de la Constitución y Pino Suarez 4°
Z.P. 1
512 70 94

Futbol 118-2
Col Country Club
Z.P. 21
549 41 82

45 Ignacio Quiñones Padilla
Servicios Profesionales de Ing.
Melchor Ocampo 445
Z.P.5
525 02 90

Lucerna 75-6
Col. Juárez
Z.P.6
591 04 70

46 Moisés Ramírez M.
Televisa S A
Blvd. A. L. Mateos 232
Z.P.20
595 74 69

Calle 651 # 133 IV y V SEC.
Sn. juan de Aragón
Z.P. 20
595 74 69

47 Juan Manuel Rebollar Pérez
Tesorería del D D F
Jefe de la Ofi. de Mantenimiento e
Instalaciones
Niños Héroes y Dr. Lavista
Z.P. 7
588 11.06

Calimaya 36
Lomas de Atizapán
Edo. de Méx.

48 Jorge Arturo Rodríguez Córdoba
U A G. ESC. DE INGENIERIA
Av. de la Juventud S/N
Chilpancingo, Gro.
2 27 41

A. Guerrero 10-B-2
Chilpancingo, Gro.

49 Enrique Ruiz Martínez Garza
BANCOMER S.A.
Av. Universidad 1200
Z.P. 13
534 00.34 Ext. 4067

Luis Kuhne 25
Col. Aguilas
Z.P. 20
593 14. 06

50 Jaime Sierra Ibarrola
SAHOP
Dir. Gral. de Maquinaria y Transportes
Miguel Laurent 840- 5°
Z.P.12
559 16 38

Av. del Arbolillo 55
Col. Hadas Coapa
Z.P.22
671 25 88

51. Jorge Arturo Soto García
Comisión del Plan Nacional Hidráulico
Tepic No. 40
Z.P.7
584 12 01

Fco. I. Madero 67
Col. Huizachal
Edo. de Méx.
589 90 17

52. Juan Germán Velenzuela Ramos
DESFI
UNAM
México 20, D.F.

Nte. 82 B # 5816
Col. G. Sánchez
Z.P. 14
551. 16 36

53. Mario Villafán González
S A H OP
Analista de Precios Unitarios
Av. Universidad y Xola
Z.P.12
530.33 36

Xochicalco 705-8
Z.P.13
559 87 83

54. Betty Zimring Nicolajevsky
COCONAL S.A.
Periférico Ote. 6501
Z.P.23
676 55 55

Av. Panamericana Casa C-34
Pedregal de Carrasco
Z.P.22
655 06 83