DIRECTORIO DE PROFESORES DEL CURSO INDTRODUCCION A LAS MINICOMPUTADORAS PDP-11

1984

1. ING. DANIEL RIOS ZERTUCHE (COORDINADOR)
DIRECTOR DE INFORMATICA
SUBSECRETARIA DE PLANEACION DEL
DESARROLLO,
SECRETARIA DE PROGRAMACION Y PRESUPUESTO
IZAZAGA NO. 38-11° PISO
MEXICO,D.F.
521 98 98

2. ING. LUIS G. CORDERO BORBOA
JEFE DEL DEPTO. DE COMPUTACION
DIVISION DE INGENIERIA MECANICA Y ELECTRICA
FACULTAD DE INGENIERIA
UNAM
MEXICO,D.F.
550 52 15 EXT. 3750

3. ING. ENRIQUE HERSCH MARTINEZ BARRANCO
GERENTE DE DESARROLLO
ACCIONES BURSATILES SOMEX, S.A.
HAMBURGO ESQ. VARSOVIA
MEXICO,D.F.
533 06 25 EXT. 147

4. ING. JORGE IVAN EUAN AVILA
PROFESOR DE TIEMPO COMPLETO
DEPARTAMENTO DE COMPUTACION
CUBICULO NO. 15
EDIFICIO DIME
FACULTAD DE INGENIERIA
UNAM
MEXICO,D.F.
550 52 15 EXT. 3746

'edcs.

Programa del Curso: Introducción a las Minicomputadoras (PDP-11) que se Impartirá del: 19 de Octubre al 24 de Noviembre

Fecha		Tema	Expositor	Lugar
Octubre	19	Introducción	Ing. Daniel Rios Zertuche O.	PM
	20	Practica Introductoria	Ing. Luis G. Cordero Borboa	DIME
	26	Elementos de una Computadora Arquitectura de la PDP-11	Ing. Enrique Hersch Martínez	PM
	27	Modos de Direccionamiento Conjunto de Instrucciones	Ing. Luis G. Cordero Borboa	PM
Noviembre	9	Manejo de Entrada/Salida Jerarquía de Memoria y su Manejo	Ing. Luis G. Cordero Borboa	PM
	10	Practica Programación	Ing. Luis G. Cordero Borboa	DIME
	16	VAX-11/780 La Familia PDP-11	Ing. Daniel Rios Zertuche O.	PM
	17	Practica I/0	Ing. Daniel Rios Zertuche O.	DIME
	23	Aplicaciones	Ing. Jorge I. Euan Avila	PM
	24	Practicas Aplicaciones	Ing. Jorge I. Euan Avila	DIME

EVALUACION DEL PERSONAL DOCENTE

CURSO: INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

FECHA: Del 19 de octubre al 24 de noviembre de 1984.

	CONFERENCISTA	DOMINIO DEL TEMA	EFICIENCIA EN EL USO DE AYUDAS AUDIOVISUALES	MANTENIMIENTO DEL INTERES. (COMUNICACION CON LOS ASISTENTES, AMENIDAD, FACILIDAD DE EXPRESION).	PUNTUALIDAD	
1.	NG. DANIEL RIOS ZERTUCHE O.					
2.	NG. LUIS G. CORDERO BORBOA					
3.	NG. ENRIQUE HERSCH MARTINEZ					
4.	NG. JORGE L. EUAN AVILA					
5.						
6.						
7.						
8.						
9.						

edcs

ESCALA DE EVALUACION: 1 a 10

EVALUACION DE LA ENSEÑANZA

SU EVALUACION SINCERA NOS AYUDARA A MEJORAR LOS PROGRAMAS POSTERIORES QUE DISEÑAREMOS PARA USTED.

TEMA	ORGANIZACION Y DESARROLLO DEL TEMA	GRADO DE PROFUNDIDAD LOGRADO EN EL TEMA	GRADO DE ACTUALIZACION LOGRADO EN EL TEMA	UTILIDAD PRACTICA DEL TEMA	
INTRODUCCION					
PRACTICA INTRODUCTORIA					
ELEMENTOS DE UNA COMPUTADORA					
ARQUITECTURA DE LA PDP-11					
MODOS DE DIRECCIONAMIENTO					
CONJUNTO DE ENTRADA/SALIDA					
JERARQUIA DE MEMORIA Y SU MANEJO					
PRACTICA PROGRAMACION					
VAX-11/780					
LA FAMILIA PDP-11					

ESCALA DE EVALUACION: 1 a 10

EVALUACION DE LA ENSEÑANZA

SU EVALUACION SINCERA NOS AYUDARA A MEJORAR LOS PROGRAMAS POSTERIORES QUE DISEÑAREMOS PARA USTED.

TEMA	ORGANIZACION Y DESARROLLO DEL TEMA	GRADO DE PROFUNDIDAD LOGRADO EN EL TEMA	GRADO DE ACTUALIZACION LOGRADO EN EL TEMA	UTILIDAD PRACTICA DEL TEMA	
PRACTICA 1/0					
APLICACIONES					
PRACTICAS APLICACIONES					

'edcs.

ESCALA DE EVALUACION 1 a 10

EVALUACION DEL CURSO

CONCEPTO		EVALUACION
1.	APLICACION INMEDIATA DE LOS CONCEPTOS EXPUESTOS	
2.	CLARIDAD CON QUE SE EXPUSIERON LOS TEMAS	
3.	GRADO DE ACTUALIZACION LOGRADO CON EL CURSO	
4.	CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
5.	CONTINUIDAD EN LOS TEMAS DEL CURSO	
6.	CALIDAD DE LAS NOTAS DEL CURSO	
7.	GRADO DE MOTIVACION LOGRADO CON EL CURSO	

ESCALA DE EVALUACION DE 1 A 10

1. ¿Qué le pareció el ambiente en la División de Educación Continua?

MUY AGRADABLE	AGRADABLE	DESAGRADABLE

2. Medio de comunicación por el que se enteró del curso:

PERIODICO EXCELSIOR ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA	PERIODICO NOVEDADES ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA	FOLLETO DEL CURSO

CARTEL MENSUAL	RADIO UNIVERSIDAD	COMUNICACION CARTA, TELEFONO, VERBAL, ETC.

REVISTAS TECNICAS	FOLLETO ANUAL	CARTELERA UNAM "LOS UNIVERSITARIOS HOY"	GACETA UNAM

3. Medio de transporte utilizado para venir al Palacio de Minería:

AUTOMOVIL PARTICULAR	METRO	OTRO MEDIO

4. ¿Qué cambios haría usted en el programa para tratar de perfeccionar el curso? ______________________________

____

____

5. ¿Recomendaría el curso a otras personas?

SI	NO

6. ¿Qué cursos le gustaría que ofreciera la División de Educación Continua?

____

____

7. La coordinación académica fue:

EXCELENTE	BUENA	REGULAR	MALA

8. Si está interesado en tomar algún curso intensivo ¿Cuál es el horario - más conveniente para usted?

LUNES A VIERNES DE 9 A 13 H. Y DE 14 A 18 H. (CON COMIDAS)	LUNES A VIERNES DE 17 A 21 H.	LUNES, MIERCOLES Y VIERNES DE 18 A 21 H.	MARTES Y JUEVES DE 18 A 21 H.

VIERNES DE 17 A 21 H. SABADOS DE 9 A 14 H.	VIERNES DE 17 A 21 H. SABADOS DE 9 A 13 Y DE 14 a 18 H.	O T R O

9. ¿Qué servicios adicionales desearía que tuviese la División de Educación Continua, para los asistentes? ______________________________

____

10. Otras sugerencias:

____

____

____

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

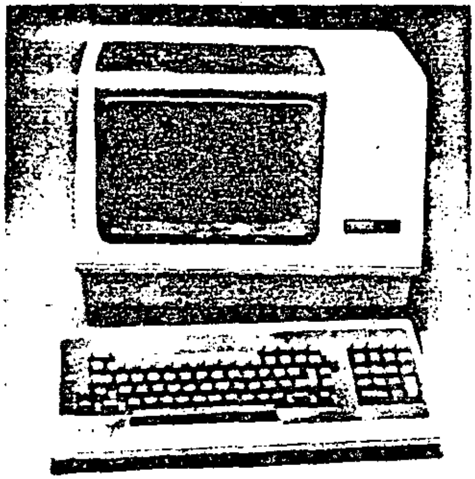
INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

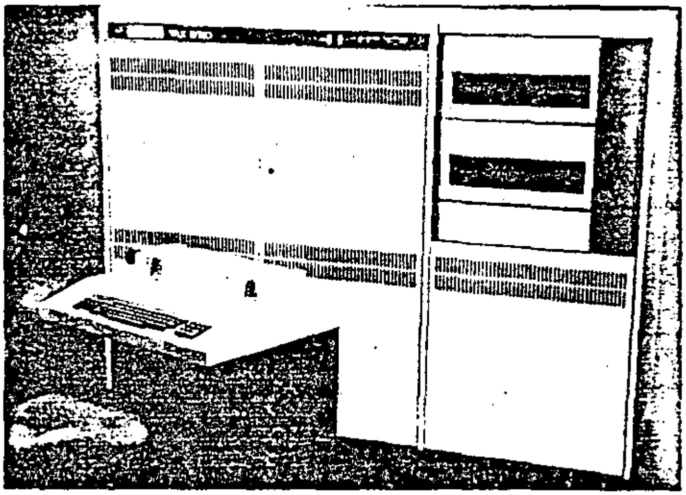
THE PDP-11 FAMILY

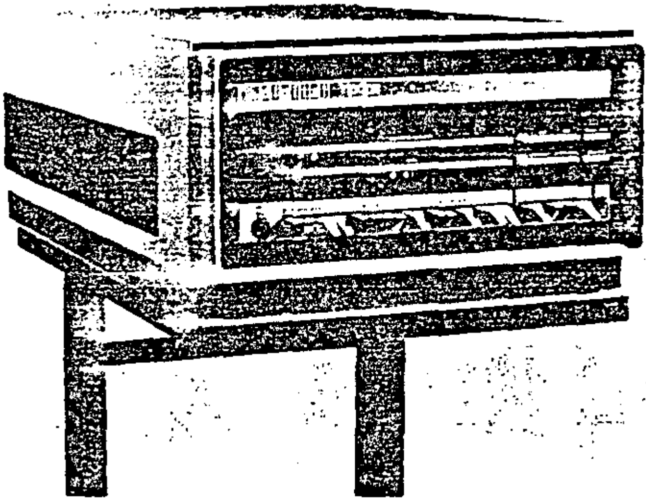
1984

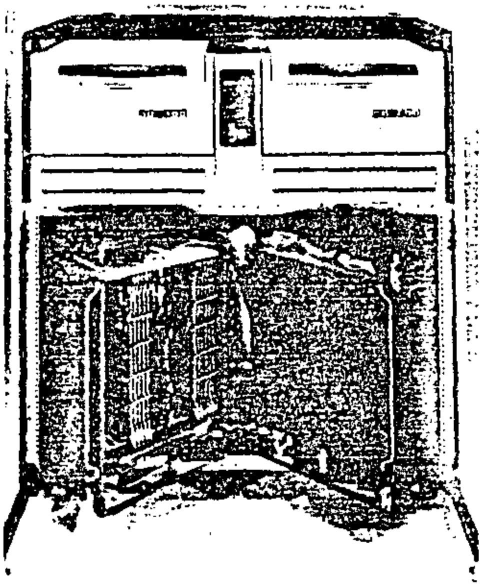
1

THE PDP-11 FAMILY









[illegible] 2

The PDP-11 Family

The PDP-11 has evolved quite differently from the other computers discussed in this book and, as a result, provides an independent and interesting story. Like the other computers, the factors that have created the various PDP-11 machines have been market and technology based, but they have generated a large number of implementations (ten) over a relatively short (eight-year) lifetime. Because there are multiple implementations spanning a performance range at the same time, the PDP-11 provides problems and insight which did not occur in the evolutions of the traditional mini (PDP-8 Family), the optimal price/performance machines (18-bit), and the high performance timesharing machines (the DECsystem 10). The PDP-11 designs cover a range of 500:1 in system price ($500 to $250,000) and 500:1 in memory size (4 Kwords to 2 Mwords).

Rather than attempt to summarize the goals of designers, sentiments of users, or the thoughts of researchers, the discussion of the PDP-11 is divided into chapters which, in most cases, consist of papers written contemporaneously with various important PDP-11 developments. The chapters are arranged in five categories: introduction to the PDP-11, conceptual basis for PDP-11 models, implementations of the PDP-11, evaluation of the PDP-11, and the virtual address extension of the PDP-11.

INTRODUCTION TO THE PDP-11

Chapter 9, first published when the PDP-11 was announced, introduces the PDP-11 architecture, gives its goals, and predicts how it might evolve. The concept of a family of machines is quite strong, but the actual development of that family has differed a good deal from the projections in this chapter. The major reasons (discussed in Chapter 16) for the disparity between the predicted and actual evolution are:

1. The notion of designing with improved technology, especially for a family of machines, was not understood in 1970. This understanding came later and was presented in a paper in 1972 [Bell *et al.*, 1972b].
2. The Unibus proved unacceptable for intercommunications at the very high and low-end designs. Although Chapter 9 suggests a multiprocessor and multiple Unibuses for high-end designs, such a structure did not evolve as a standard.
3. The address space for both physical and virtual memory was too small.

3

4. Several data-type extensions were not predicted. Although floating-point arithmetic was envisioned, the character string and decimal operations were not envisioned, or at least were not described. These data-types evolved in response to market needs that did not exist in 1970.

CONCEPTUAL BASIS FOR THE PDP-11 MODELS

Chapters 10 and 11 consist of two papers that form some of the conceptual basis for the various PDP-11 models. Chapter 10 by Strecker is an exposition of cache memory structure and its design parameters. The cache memory concept is the basis of three PDP-11 models, the PDP-11/34A, the PDP-11/60, and the PDP-11/70, in addition to the cache-8 (Chapter 7) and the KL10 processor for the PDP-10 (Chapter 21).

Strecker gives the performance evaluation in terms of cache miss ratios, whereas the reader is probably interested in performance or speedup. These two measures, shown in Figure 1, are related [Lee, 1969] in the following way (assuming an infinitely fast processor):

p = Total number of memory accesses by the processor Pc
m = Number of memory accesses that are missed by the cache and have to be referred to the primary memory Mp
$t.c$ = Cycle time of cache memory Mc
$t.p$ = Cycle time of primary memory Mp
R = $t.p/t.c$ (ratio of memory speeds), where R is typically 3 to 10

The relative execution speeds are:

$$t\,(no\ cache) = pR$$
$$t\,(to\ cache) = p + mR$$
$$speedup = pR/(p + mR) = R/(1 + (m/p)\,R)$$
$$a = miss\ ratio = m/p$$

Therefore:

$$speedup = R/(1 + aR) = 1/(a + 1/R)$$

Note that:

If a = 0 (100% hit), the speedup is R
If a = 1 (100% miss), the speedup is $R/(1 + R)$, i.e., the speedup is less than 1 (i.e., time to reference both memories)

Chapter 11 contains a unique discussion of buses – the communications link between two or more computer system components. Although buses are a standard of interconnection, they are the least understood element of computer design

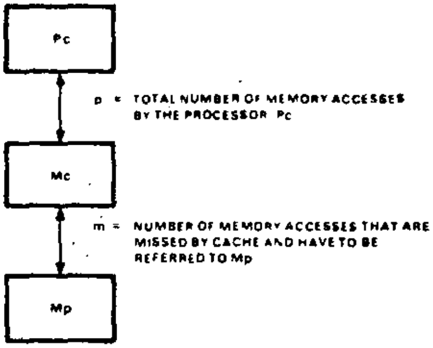

Pc
p = TOTAL NUMBER OF MEMORY ACCESSES BY THE PROCESSOR Pc
Mc
m = NUMBER OF MEMORY ACCESSES THAT ARE MISSED BY CACHE AND HAVE TO BE REFERRED TO Mp
Mp

Figure 1. The structure of Pc, Mcache, and Mp of cached computer.

because their implementation is distributed in various components. Their behavior is difficult to express in a state diagram or other conventional representation (except a timing diagram) because the operation of buses is inherently pipelined; hence, design principles and understanding are minimal.

In Chapter 11, Levy first characterizes the intercommunication problem into the constituent dialogues that must take place between pairs of components. After giving a general model of interconnection, Levy provides examples of PDP-11 buses that characterize the general design space. Finally, he discusses the various intercommunications (model) aspects: arbitration (deciding which components can intercommunicate), data transmission, and error control.

IMPLEMENTATIONS OF THE PDP-11

Chapter 12 is a descriptive narrative about the design of the LSI-11 at the chip, board, and backplane levels. Since it was written from the viewpoint of a knowledgeable user, it lacks some of the detail that the designers at Western Digital (Roberts, Soha, Pohlman) or at DEC (Dickhut, Dickman, Olsen, Titelbaum) might have provided. A detailed account of the chip-level design is available, however [Soha and Pohlman, 1974].

Two design levels are described: the three chip set microprogrammed computer used to interpret the PDP-11 instruction set, and the particular PMS-level components that are integrated into a backplane to form a hardware system. Chapter 12 also provides a discussion of the microprogramming tradeoff that took place between the chip and module levels. This tradeoff was necessary to carry out the clock, console, refresh, and power-fail functions which are normally in hardware.

Since the time that the Sebern paper (Chapter 12) was written, packaging for LSI-11 systems has moved in two directions: toward the single board microcomputer and toward modularity. The single board microcomputer concept is the highest performance machine of the family, and thus has to have the right balance of features, price, and performance against criteria that are usually vague.

Four interesting aspects of computer engineering are shown in the PDP-11/60: the cache to reduce Unibus traffic; trace-driven design of floating-point arithmetic processors; writable control store; and special features for reliability, availability, and maintainability.

The Unibus was found to be inadequate for handling all the data traffic in high performance systems, but by using a cache, most processor references do not use the Unibus and so leave it free for I/O traffic. In the PDP-11/60 work described in this chapter, Mudge uses Strecker's (Chapter 10) program traces and methodology. The cache design process is implicit in the way in which the work is carried out to determine the structure parameters. Sensitivity plots are used to determine the effects of varying each parameter of the design. The time between changes of context is an important parameter because all real-time and multiprogrammed systems have many context switches. The study leading to the determination of block size is also given.

Microprogramming is used to provide both increased user-level capability and increased reliability, availability, and maintainability. The writable control store option is described together with its novel use for data storage. This option has been recently used for emulating the PDP-8 at the OS/8 operating system level.

Chapter 14 presents a comprehensive comparison of the eight processor implementations used in the ten PDP-11 models. The work was carried out to investigate various design styles for a given problem, namely, the interpretation of the PDP-11 instruction set. The tables provide valuable insight into processor implementations, and the data is particularly useful because it comes from Snow and Siewiorek, non-DEC observers examining the PDP-11 machines.

The tables include:

1. A set of instruction frequencies, by Strecker, for a set of ten different applications. (The frequencies do not reflect all uses, e.g., there are no floating-point instructions, nor has operating system code been analyzed.)
2. Implementation cost (modules, integrated circuits, control store widths) and performance (micro- and macroinstruction times) for each model.
3. A canonical data path for all PDP-11 implementations against which each processor is compared.

With this background data, a top-down model is built which explains the performance (macroinstruction time) of the various implementations in terms of the microinstruction execution and primary memory cycle time. Because these two parameters do not fully explain (model) performance, a bottom-up approach is also used, including various design techniques and the degree of processor overlap. This analysis of a constrained problem should provide useful insight to both computer and general digital systems designers.

exemplified by the bounded system shown in Figure 2. This integrated system contains an LSI-11 chip set, 32 Kwords of memory, connectors for six communication line interfaces, and a controller for two floppy disk drives. It uses 175 circuits (to implement the same functionality using standard LSI-11 modules would require 375 integrated circuits). The modularity direction is exemplified by the LSI-11/2, for which typical option modules are shown in Figure 3.

Unlike the reports from an architect's or reporter's viewpoint, Chapter 13 is a direct account of the design process from the project viewpoint. A mid-range machine is an inherently difficult design because it is neither the lowest cost nor

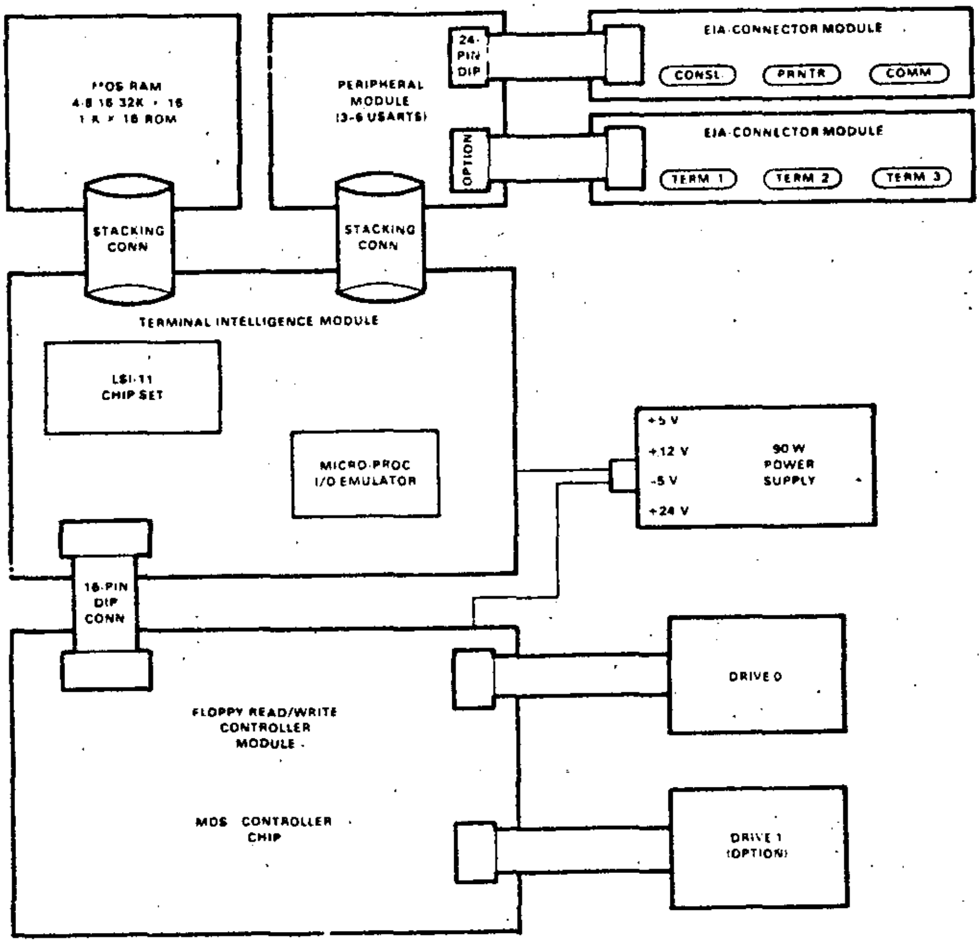
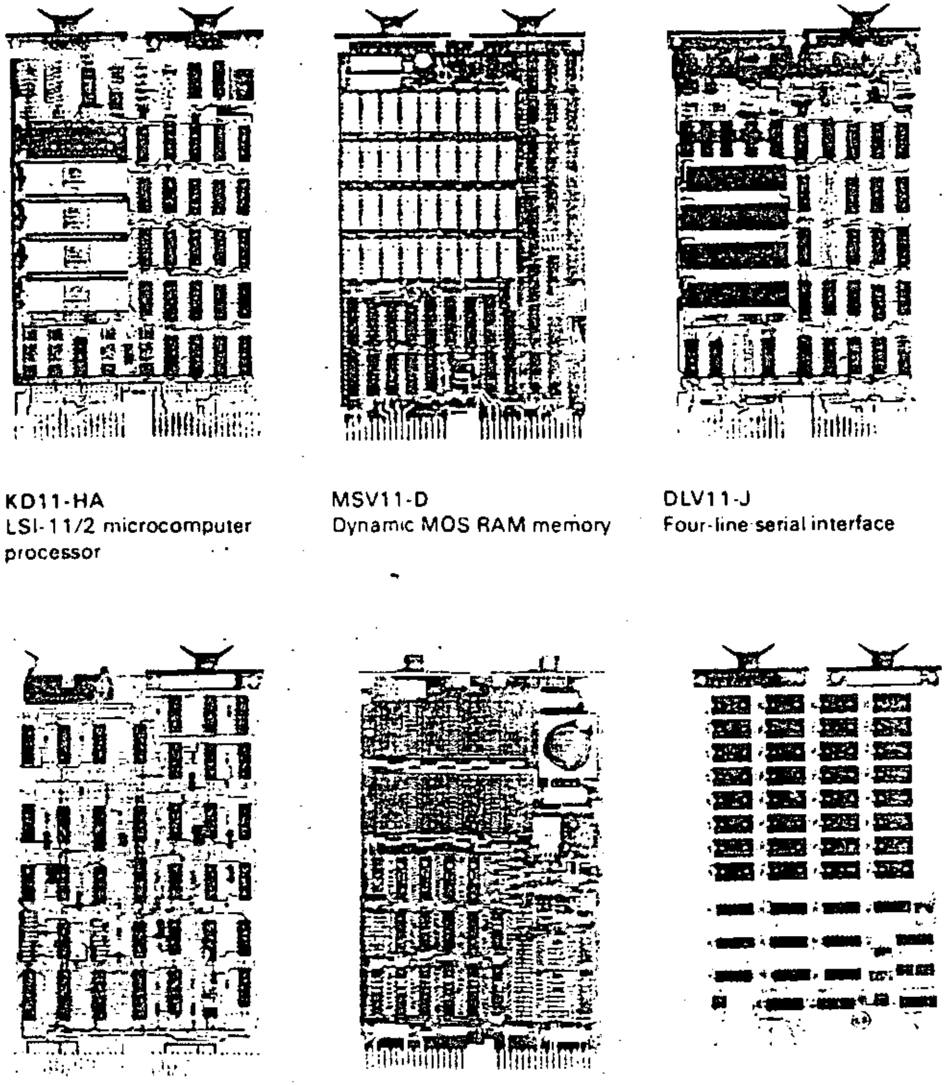


Figure 2. A bounded LSI-11 based system.

7



KD11-HA
LSI-11/2 microcomputer processor

MSV11-D
Dynamic MOS RAM memory

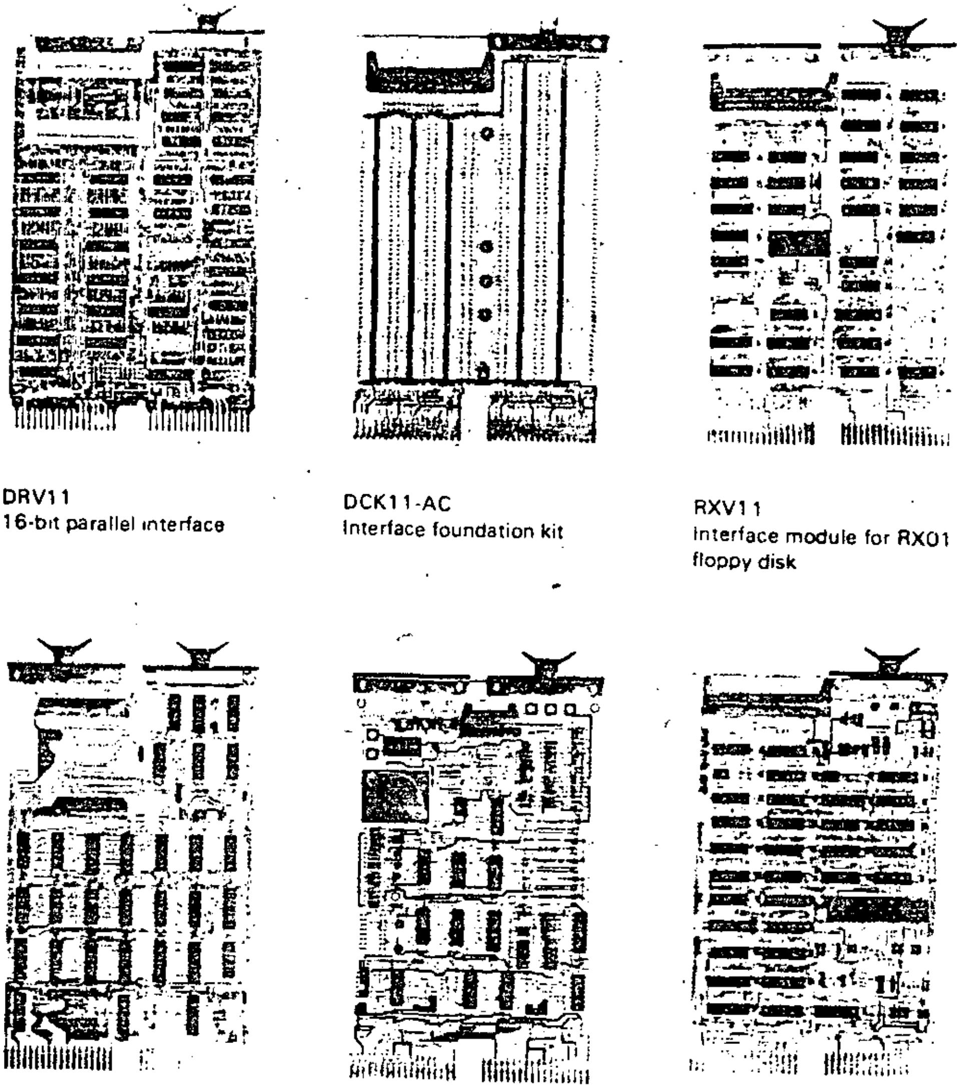
DLV11-J
Four-line serial interface

IBV11-A
IEEE instrument bus interface

MRV11-BA
4K UV PROM board with 256-word RAM

MRV11-AA
4K PROM board

Figure 3. The double-height modules forming the LSI-11/2 (part 1 of 2).



DRV11
16-bit parallel interface

DCK11-AC
Interface foundation kit

RXV11
Interface module for RX01
floppy disk

REV11-A
Refresh/ bootstrap/
diagnostic/ terminator
module

KPV11-A
Power sequencer/ line clock
module

DLV11
Single-line serial interface

Figure 3. The double-height modules forming the LSI-11/2 (part 2 of 2).

EVALUATION OF THE PDP-11

Chapter 15 evaluates the PDP-11 as a machine for executing FORTRAN. Because FORTRAN is the most often executed language for the PDP-11, it is important to observe the PDP-11 architecture as seen by the language processor – its user. The first FORTRAN compiler and object (run) time system are described, together with the evolutionary extensions to improve performance. The FORTRAN IV-PLUS (optimizing) compiler is only briefly discussed because its improvements, largely due to compiler optimization technology, are less relevant to the PDP-11 architecture.

The chapter title, "Turning Cousins into Sisters," overstates the compatibility problem since the five variations of the PDP-11 instruction set for floating-point arithmetic are made compatible by essentially providing five separate object (run) time systems and a single compiler. This transparency is provided quite easily by "threaded code," a concept discussed in the chapter.

The first version of the FORTRAN machine was a simple stack machine. As such, the execution times turned out to be quite long. In the second version, the recognition of the special high-frequency-of-use cases (e.g., $A \leftarrow 0$, $A \leftarrow A + 1$) and the improved conventions for three-address operations (to and from the stack) allowed speedup factors of 1.3 and 2.0 for floating-point and integers.

It is interesting to compare Brender's idealized FORTRAN IV-PLUS machine with the Floating-Point Processors (on the PDP-11/34, 11/45, 11/55, 11/60, and 11/70). If the FORTRAN machine described in the paper is implemented in microcode and made to operate at Floating-Point Processor speeds, the resulting machines operate at roughly the same speed and programs occupy roughly the same program space.

The basis for Chapter 16, "What Have We Learned From the PDP-11?" [Bell and Strecker, 1976] was written to critique the original expository paper on the PDP-11 (Chapter 9) and to compare the actual with the predicted evolution. Four critical technological evolutions – bus bandwidth, PMS structure, address space, and data-type – are examined, along with various human organizational aspects of the design.

The first section of Chapter 16 compares the original goals of the PDP-11 (Chapter 9) with the goals of possible future models from the original design documents. Next, the ISP and PMS evolutions, including the VAX extension, are described. The Unibus characteristics are especially interesting as the bus turns out to be more cost-effective over a wider range than would be expected.

The section of the chapter which deals with multiprocessors and multicomputers gives the rationale behind the slow evolution of these structures. Because a number of these computer structures have been built (especially at Carnegie-Mellon University), they are described in detail.

The final section of the chapter interrelates technology with the various implementations (including VAX-11/780) that have occurred. Table 6 gives the performance characteristics for the various models with the relevant technology, contributions, and implementation techniques required to span the range.

VIRTUAL ADDRESS EXTENSION OF THE PDP-11

The latest member of the PDP-11 family, the Virtual Address Extension 11 or VAX-11, is described in Chapter 17. This paper, by the architect of VAX-11, discusses the new architecture and its first implementation, the VAX-11/780.

VAX-11 extends the PDP-11 to provide a large, 32-bit virtual address for each user process. The architecture includes a compatibility mode that allows PDP-11 programs written for the RSX-11M program environment to run unchanged. In this way, PDP-11 programs can be moved among VAX and PDP-11 computers, depending on the user's address size and computational and generality needs.

Chapter 17 provides a clean, somewhat terse, yet comprehensive description of the VAX-11 architecture. Because the VAX part of the architecture is so complete in terms of data-types, operators, addressing and memory management, it can also serve as a textbook model and case study for architecture in general. Goals, constraints, and various design choices are given, although explanations of what was traded away in the design choices are not detailed.

11

9

A New Architecture for Minicomputers —The DEC PDP-11

C. GORDON BELL, ROGER CADY, HAROLD McFARLAND, BRUCE A. DELAGI, JAMES F. O'LOUGHLIN, RONALD NOONAN, and WILLIAM A. WULF

INTRODUCTION

The minicomputer* has a wide variety of uses: communications controller, instrument controller, large-system preprocessor, real-time data acquisition systems, . . . desk calculator. Historically, Digital Equipment Corporation's (DEC) PDP-8 family, with 6000 installations has been the archetype of these minicomputers.

In some applications current minicomputers have limitations. These limitations show up when the scope of their initial task is increased (e.g., using a higher level language, or processing more variables). Increasing the scope of the task generally requires the use of more comprehensive executives and system control programs, hence larger memories and more processing. This larger system tends to be at the limit of current minicomputer capability, thus the user receives diminishing returns with respect to memory, speed efficiency, and program development time. This limitation is not surprising since the basic architectural concepts for current minicomputers were formed in the early 1960s. First, the design was constrained by cost, resulting in rather simple processor logic and

* The PDP-11 design is predicated on being a member of one (or more) of the micro, midi, mini, . . . maxi (computer name) markets. We will define these names as belonging to computers of the third generation (integrated circuit to medium-scale integrated circuit technology), having a core memory with cycle time of 0.5 ~ 2 μs, a clock rate of 5 ~ 10 MHz . . . a single processor with interrupts and usually applied to doing a particular task (e.g., controlling a memory or communications lines, preprocessing for a larger system, process control). The specialized names are defined as follows.

	Maximum Addressable Primary Memory (Words)	Processor and Memory Cost (1970 Kilodollars)	Word Length (Bits)	Processor State (Words)	Data-Types
Micro	8 K	~5	8 ~ 12	2	Integers, words, Boolean vectors
Mini	32 K	5 ~ 10	12 ~ 16	2-4	Vectors (i.e., indexing)
Midi	65 K ~ 128 K	10 ~ 20	16 ~ 24	4-16	Double length floating point (occasionally)

register configurations. Second, application experience was not available. For example, the early constraints often created computing designs with what we now consider weaknesses:

1. Limited addressing capability, particularly of larger core sizes.
2. Few registers, general registers, accumulators, index registers, base registers.
3. No hardware stack facilities.
4. Limited priority interrupt structures, and thus slow context switching among multiple programs (tasks).
5. No byte string handling.
6. No read-only memory (ROM) facilities.
7. Very elementary I/O processing.
8. No larger model computer, once a user outgrows a particular model.
9. High programming costs because users program in machine language.

In developing a new computer, the architecture should at least solve the above problems. Fortunately, in the late 1960s, integrated circuit semiconductor technology became available so that newer computers could be designed that solve these problems at low cost. Also, by 1970, application experience was available to influence the design. The new architecture should thus lower programming cost while maintaining the low hardware cost of minicomputers.

The DEC PDP-11 Model 20 is the first computer of a computer family designed to span a range of functions and performance. The Model 20 is specifically discussed, although design guidelines are presented for other members of the family. The Model 20 would nominally be classified as a third generation (integrated circuits), 16-bit word, one central processor with eight 16-bit general registers, using two's complement arithmetic and addressing up to 2^{16} 8-bit bytes of primary memory (core). Though classified as a general register processor, the operand accessing mechanism allows it to perform equally well as a 0- (stack), 1- (general register), and 2- (memory-to-memory) address computer. The computer's components (processor, memories, controls, terminals) are connected via a single switch, called the Unibus.

The machine is described using the processor-memory-switch (PMS) notation of Bell and Newell [1971] at different levels. The following descriptive sections correspond to the levels: external design constraints level; the PMS level – the way components are interconnected and allow information to flow; the program level – the abstract machine that interprets programs; and finally, the logical design level. (We omit a discussion of the circuit level, the PDP-11 being constructed from TTL integrated circuits.)

DESIGN CONSTRAINTS

The principal design objective is yet to be tested; namely, do users like the machine? This will be tested both in the marketplace and by the features that are emulated in newer machines; it will be tested indirectly by the life span of the PDP-11 and any offspring.

Word Length

The most critical constraint, word length (defined by IBM), was chosen to be a multiple of 8 bits. The memory word length for the Model 20 is 16 bits, although there are 32- and 48-bit instructions and 8- and 16-bit data. Other members of the family might have up to 80-bit instructions with 8-, 16-, 32- and 48-bit data. The internal, and preferred external character set, was chosen to be 8-bit ASCII.

Range and Performance

Performance and function range (extendability) were the main design constraints; in fact, they were the main reasons to build a new computer. DEC already has four computer families that span a range* but are incompatible. In addition to the range, the initial machine was constrained to fall within the small-computer product line, which means to have about the same performance as a PDP-8. The initial machine outperforms the PDP-5, LINC, and PDP-4 based families. Performance, of course, is both a function of the instruction set and the technology. Here, we are fundamentally only concerned with the instruction set performance because faster hardware will always increase performance for any family. Unlike the earlier DEC families, the PDP-11 had to be designed so that new models with significantly more performance can be added to the family.

A rather obvious goal is maximum performance for a given model. Designs were programmed using benchmarks, and the results were compared with both DEC and potentially competitive machines. Although the selling price was constrained to lie in the $5,000 to $10,000 range, it was realized that the decreasing cost of logic would allow a more complex organization than that of earlier DEC computers. A design that could take advantage of medium- and eventually large-scale integration was an important consideration. First, it could make the computer perform well; second, it would extend the computer family's life. For these reasons, a general register organization was chosen.

Interrupt Response. Since the PDP-11 will be used for real-time control applications, it is important that devices can communicate with one another quickly (i.e., the response time of a request should be short). A multiple priority level, nested interrupt mechanism was selected; additional priority levels are provided by the physical position of a device on the Unibus. Software polling is unnecessary because each device interrupt corresponds to a unique address.

Software

The total system including software is, of course, the main objective of the design. Two techniques were used to aid programmability. First, benchmarks gave a continuous indication as to how well the machine interpreted programs; second, systems programmers continually evaluated the design. Their evaluation considered: what code the compiler would produce; how would the loader work; ease of program relocatability; the use of a debugging program; how the compiler, assembler, and editor would be coded – in effect, other benchmarks; how real-time monitors would be written to use the various facilities and present a clean interface to the users; finally, the ease of coding a program.

Modularity

Structural flexibility (sometimes called modularity) for a particular model was desired. A flexible and straightforward method for interconnecting components had to be used because of varying user needs (among user classes and over time). Users should have the ability to configure an optimum system based on cost, performance, and reliability, both by interconnection and, when necessary, constructing new components. Since users build special hardware, a computer should be interfaced easily. As a by-product of modularity, computer components can be produced and stocked, rather than tailor-made on order. The physical structure is almost identical to the PMS structure discussed in the following section; thus,

* PDP-4, 7, 9, 15 family; PDP-5, 8, 8/S, 8/I, 8/L family; LINC, PDP-8/LINC, PDP-12 family; and PDP-6, 10 family. The initial PDP-1 did not achieve family status.

reasonably large building blocks are available to the user.

Microprogramming

A note on microprogramming is in order because of current interest in the "firmware" concept. We believe microprogramming, as we understand it [Wilkes and Stringer, 1953], can be a worthwhile technique as it applies to processor design. For example, microprogramming can probably be used in larger computers when floating-point data operators are needed. The IBM System 360 has made use of the technique for defining processors that interpret both the System 360 instruction set and earlier family instruction sets (e.g., 1401, 1620, 7090). In the PDP-11, the basic instruction set is quite straightforward and does not necessitate microprogrammed interpretation. The processor-memory connection is asynchronous: therefore, memory of any speed can be connected. The instruction set encourages the user to write reentrant programs. Thus, read-only memory can be used as part of primary memory to gain the permanency and performance normally attributed to microprogramming. In fact, the Model 10 computer, which will not be further discussed, has a 1024-word read-only memory, and a 128-word read-write memory.

Understandability

Understandability was perhaps the most fundamental constraint (or goal) although it is now somewhat less important to have a machine that can be understood quickly by a novice computer user than it was a few years ago. DEC's early success has been predicated on selling to an intelligent but inexperienced user. Understandability, though hard to measure, is an important goal because all (potential) users must understand the computer. A straightforward design should simplify the systems programming task; in the case of a compiler, it should make translation (particularly code generation) easier.

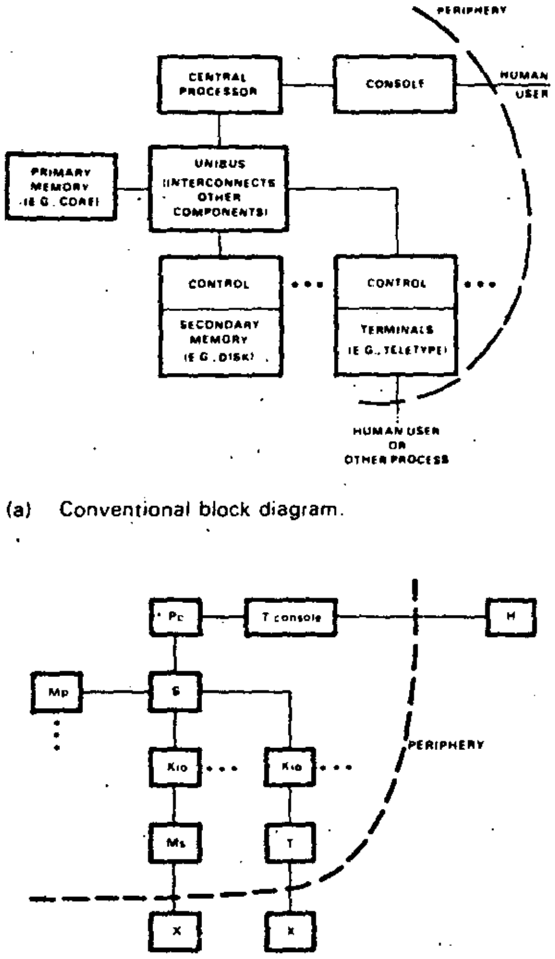
PDP-11 STRUCTURE AT THE PMS LEVEL*

Introduction

PDP-11 has the same organizational structure as nearly all present-day computers (Figure 1). The primitive PMS components are: the primary memory Mp which holds the programs while the central processor Pc interprets them; I/O controls Kio which manage data transfers between terminals T or secondary memories Ms to primary memory Mp; the components outside the computer at periphery X either humans H or some external process (e.g., another computer); the processor console (T.console) by which humans communicate with the computer and observe its behavior and affect changes in its state; and a switch S with its control K which allows all the other components to communicate with one another. In the case of PDP-11, the central logical switch structure is implemented using a bus or chained switch S called the Unibus, as shown in Figure 2. Each physical component has a switch for placing messages on the bus or taking messages off the bus. The central control decides the next component to use the bus for a message (call). The S (Unibus) differs from most switches because any component can communicate with any other component.

The types of messages in the PDP-11 are along the lines of the hierarchical structure common to present-day computers. The single

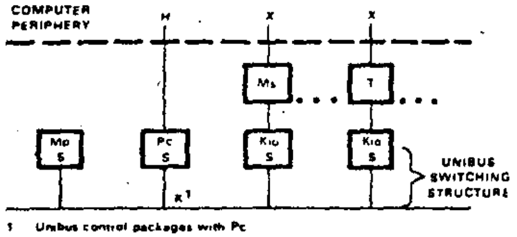
*A descriptive (block-diagram) level [Bell and Newell, 1970] to describe the relationship of the computer components: processors, memories, switches, controls, links, terminals, and data operators. PMS is described in Appendix 2.


PERIPHERY
CENTRAL PROCESSOR
CONSOLE
HUMAN USER
PRIMARY MEMORY (E.G. CORE)
UNIBUS (INTERCONNECTS OTHER COMPONENTS)
CONTROL
SECONDARY MEMORY (E.G. DISK)
CONTROL
TERMINALS (E.G., TELETYPE)
HUMAN USER OR OTHER PROCESS
Pc
T console
H
Mp
S
Kio
Kio
PERIPHERY
Ms
T
X
X

(a) Conventional block diagram.

(b) PMS diagram (see Appendix 2).

Figure 1. Conventional block diagram and PMS diagram of PDP-11.



1 Unibus control packages with Pc

Figure 2. PDP-11 physical structure PMS diagram.

bus makes conventional and other structures possible. The message processes in the structure that utilize S (Unibus) are:

1. The central processor Pc requests that data be read or written from or to primary memory Mp for instructions and data. The processor calls a particular memory module by concurrently specifying the module's address, and the address within the modules. Depending on whether the processor requests reading or writing, data is transmitted either from the memory to the processor or vice versa.
2. The central processor Pc controls the initialization of secondary memory Ms and terminal T activity. The processor sets status bits in the control associated with a particular Ms or T, and the device proceeds with the specified action (e.g., reading a card or punching a character into paper tape). Since some devices transfer data vectors directly to primary memory, the vector control information (i.e., the memory location and length) is given as initialization information.
3. Controls request the processor's attention in the form of interrupts. An interrupt request to the processor has the effect of changing the state of the processor; thus, the processor begins executing a program associated with the interrupting process. Note that the interrupt process is only a signaling method, and when the processor interrupt occurs, the interrupter specifies a unique address value to the processor. The address is a starting address for a program.
4. The central processor can control the transmission of data between a control (for T or Ms) and either the processor or a primary memory for program controlled data transfers. The device signals for attention using the interrupt dialogue

and the central processor responds by managing the data transmission in a fashion similar to transmitting initialization information.

5. Some device controls (for T or Ms) transfer data directly to/from primary memory without central processor intervention. In this mode the device behaves similarly to a processor; a memory address is specified, and the data is transmitted between the device and primary memory.
6. The transfer of data between two controls, e.g., a secondary memory (disk) and say a terminal/T. display is not precluded, provided the two use compatible message formats.

As we show more detail in the structure there are, of course, more messages (and more simultaneous activity). The above does not describe the shared control and its associated switching which is typical of a magnetic tape and magnetic disk secondary memory systems. A control for a DECtape memory (Figure 3) has an S ('DECtape bus) for transmitting data between a single tape unit and the DECtape transport. The existence of this kind of structure is based on the relatively high cost of the control relative to the cost of the tape and the value of being able to run concurrently with other tapes. There is also a dialogue at the periphery between X-T and X-Ms that does not use the Unibus. (For example, the removal of a magnetic tape reel from a tape unit or a human user H striking a typewriter key are typical dialogues.)

All of these dialogues lead to the hierarchy of present computers (Figure 4). In this hierarchy we can see the paths by which the above messages are passed: Pc-Mp; Pc-K; K-Pc; Kio-T and Kio-Ms; and Kio-Mp; and, at the periphery, T-X and T-Ms; and T. console-H.

Model 20 Implementation

Figure 5 shows the detailed structure of a uniprocessor Model 20 PDP-11 with its various components (options). In Figure 5, the Unibus characteristics are suppressed. (The detailed properties of the switch are described in the logical design section.)

Extensions to Increase Performance

The reader should note (Figure 5) that the important limitations of the bus are: a concurrency of one, namely, only one dialogue can occur at a given time, and a maximum transfer rate of one 16-bit word per 0.75 microsecond, giving a transfer rate of 21.3 megabits/second. While the bus is not a limit for a uniprocessor structure, it is a limit for multiprocessor structures. The bus also imposes an artificial limit on the system performance when high-speed devices (e.g., TV cameras, disks) are transferring

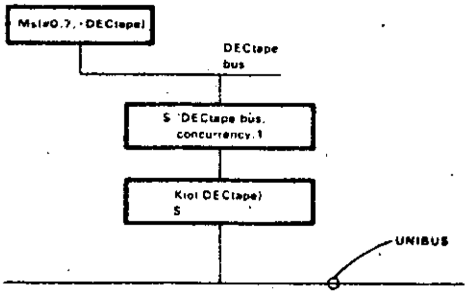


Figure 3. DECtape control switching PMS diagram.

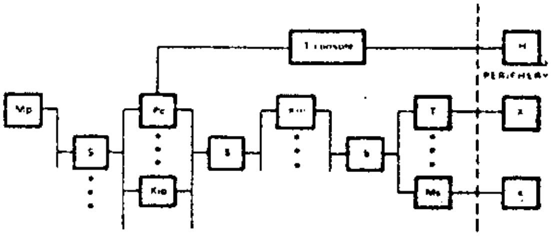
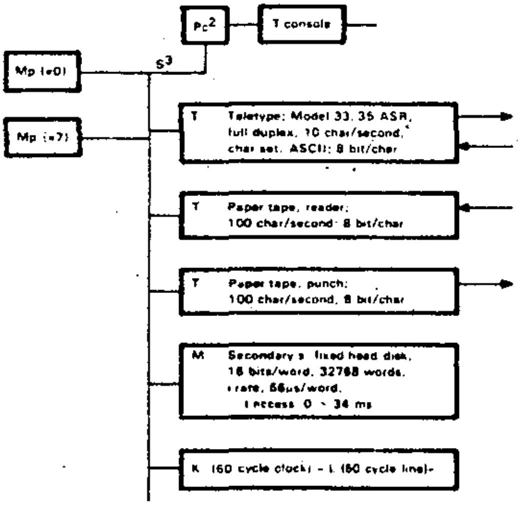


Figure 4. Conventional hierarchy computer structure

data to multiple primary memories. On a larger system with multiple independent memories, the supply of memory cycles is 17 megabits/second times the number of modules. Since there is such a large supply of memory cycles per second and since the central processor can absorb only approximately 16 megabits/second, the simple one-Unibus structure must be modified to make the memory cycles available. Two changes are necessary. First, each of the memory modules has to be changed so that multiple units can access each module on an independent basis. Second, there must be independent control accessing mechanisms. Figure 6 shows how a single memory is modified to have more access ports (i.e., connect to four Unibuses).


Pc²
T console
Mp (#0)
S³
Mp (#7)
T Teletype: Model 33, 35 ASR, full duplex, 10 char/second, char set, ASCII; 8 bit/char
T Paper tape, reader; 100 char/second· 8 bit/char
T Paper tape, punch; 100 char/second, 8 bit/char
M Secondary s fixed head disk, 16 bits/word, 32768 words, t rate, 66μs/word, t access 0 ~ 34 ms
K (60 cycle clock) – L (60 cycle line)-

NOTES

1 Mp (technology: core, 4096 words; t cycle 1 2 μs; t access 0 6 μs, 16 bits/word)

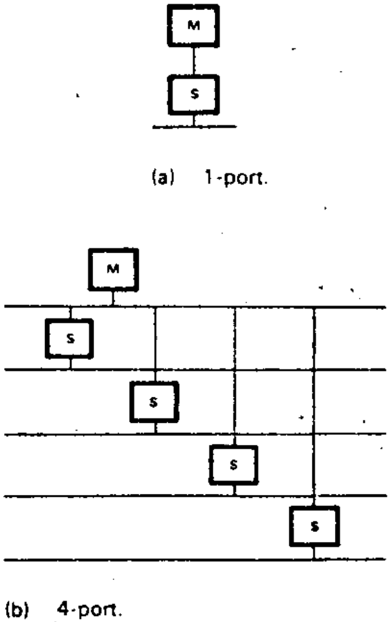
2. P(central c, Model 30, integrated circuit, general registers, 2 addresses/instruction, *addresses are register stack, Mp,* data-types bits, bytes, words, word integers, byte integers, Boolean vectors, 8 bits/byte, 16 bits/word, operations (+, −, / optional), × (optional) /2, ×2,¬, − (negate); [illegible] Mprocessor state, (general registers; 8 + 1 word; integrated circuit))

3 S (Unibus non-hierarchy, bus; concurrency 1, 1 word/0 75 μs)

Figure 5. PDP-11 structure and characteristics PMS diagram.

Figure 7 shows a system with three independent memory modules that are accessed by two independent Unibuses. Note that two of the secondary memories and one of the transducers are connected to both Unibuses. It should be noted that devices that can potentially interfere with Pc-Mp accesses are constructed with two ports; for simple systems, both ports are connected to the same bus, but for systems with more buses, the second connection is to an independent bus.

Figure 8 shows a multiprocessor system with two central pı ocessors and three Uniouses. Two of the Unibus controls are included within the two processors, and the third bus is controlled by an independent control unit. The structure also has a second switch to allow either of two processors (Unibuses) to access common shared devices. The interrupt mechanism allows either



(a) 1-port.

(b) 4-port.

Figure 6 1- and 4-port memory modules PMS diagram.

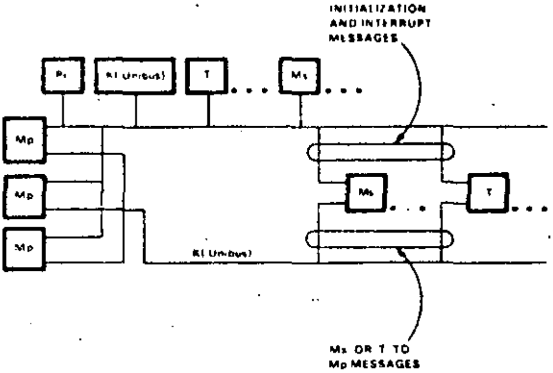
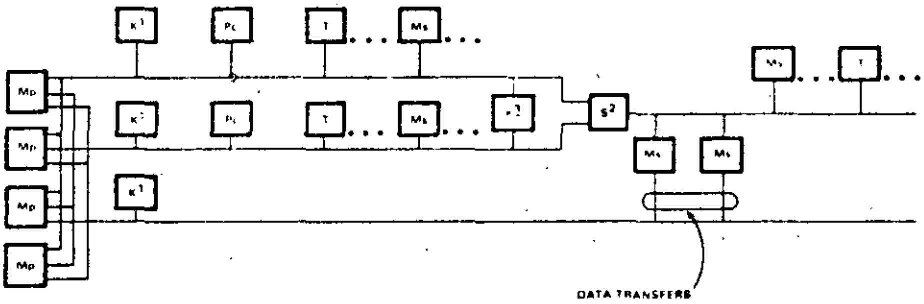


Figure 7. Three Mp, two S ('Unibus) structure PMS diagram.



1. K('Unibus)
2. K('Unibus multiple bus to single bus coupler; from 2 Unibus, to: 1 Unibus)
3. K('Processor-to-processor coupler)
4. Ms(duplex)

Figure 8. Dual Pc multiprocessor system PMS diagram.

processor to respond to an interrupt, and similarly either processor may issue initialization information on an anonymous basis. A control unit is needed so that two processors can communicate with one another: shared primary memory is normally used to carry the body of the message. A control connected to two Pc's (Figure 8) can be used for reliability: either processor or Unibus could fail, and the shared Ms would still be accessible.

Higher Performance Processors

Increasing the bus width has the greatest effect on performance. A single bus limits data transmission to 21.4 megabits/second, and though Model 20 memories are 16 megabits/second, faster (or wider) data path width modules will be limited by the bus. The Model 20 is not restricted, but for higher performance processors operating on double-word (fixed-point) or triple-word (floating-point) data, two or three accesses are required for a single datatype. The direct method to improve the performance is to double or triple the primary memory and central processor data path widths. Thus, the bus data rate is automatically doubled or tripled.

For 32- or 48-bit memories, a coupling control unit is needed so that devices of either width appear isomorphic to one another. The coupler maps a data request of a given width into a higher- or lower-width request for the bus being coupled to, as shown in Figure 9. (The bus is limited to a fixed number of devices for electrical reasons; thus, to extend the bus, a bus-repeating unit is needed. The bus-repeating control unit is almost identical to the bus coupler.) A computer with a 48-bit primary memory and processor and 16-bit secondary memory and terminals (transducers) is shown in Figure 9.

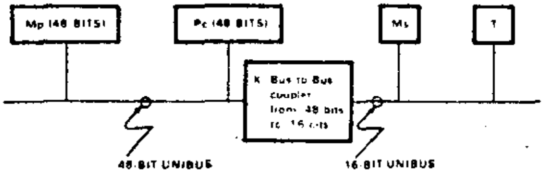


Figure 9. Computer with 48-bit Pc Mp with 16-bit Ms. T.PMS diagram.

In summary, the design goal was to have a modular structure providing the final user with freedom and flexibility to match his needs. A secondary goal of the Unibus is open-endedness by providing multiple buses and defining wider path buses. Finally, and most important, the Unibus is straightforward.

THE INSTRUCTION SET PROCESSOR (ISP) LEVEL-ARCHITECTURE[*]

Introduction, Background, and Design Constraints

The Instruction Set Processor (ISP) is the machne defined by the hardware and/or software that interprets programs. As such, an ISP is independent of technology and specific implementations.

The instruction set is one of the least understood aspects of computer design; currently, it is an art. There is currently no theory of instruction sets, although there have been attempts to construct them [Maurer, 1966], and there has also been an attempt to have a computer program design an instruction set [Haney, 1968]. We have used the conventional approach in this design. First, a basic ISP was adopted and then incremental design modifications were made (based on the results of the benchmarks).[†]

Although the approach to the design was conventional, the resulting machine is not. A common classification of processors is as 0-, 1-, 2-, 3-, or 3-plus-1-address machines. This scheme has the form:

op *l*1, *l*2, *l*3, *l*4

[*] The word "architecture" has been operationally defined [Amdahl *et al.*, 1964] as "the attributes of a system as seen by a programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

[†] A predecessor multiregister computer was proposed that used a similar design process. Benchmark programs were coded on each of ten "competitive" machines, and the object of the design was to get a machine that gave the best score on the benchmarks. This approach had several fallacies: The machine had no basic character of its own; the machine was difficult to program since the multiple registers were assigned to specific functions and had inherent idiosyncrasies to score well on the benchmarks; the machine did not perform well for programs other than those used in the benchmark test; and finally, compilers that took advantage of the machine appeared to be difficult to write. Since all "competitive machines" had been hand-coded from a common flowchart rather than separate flowcharts for each machine, the apparent high performance may have been due to the flowchart organization.

where *l*1 specifies the location (address) in which to store the result of the binary operation (op) of the contents of operand locations *l*2 and *l*3, and *l*4 specifies the location of the next instruction.

The action of the instruction is of the form:

*l*1 ← *l*2 op *l*3; goto *l*4

The other addressing schemes assume specific values for one or more of these locations. Thus, the one-address von Neumann [Burks *et al.*, 1962] machines assume *l*1 = *l*2 = the accumulator and *l*4 is the location following that of the current instruction. The two-address machine assumes *l*1 = *l*2: *l*4 is the next address.

Historically, the trend in machine design has been to move from a 1- or 2-word accumulator structure as in the von Neumann machine toward a machine with accumulator and index register(s).* As the number of registers is increased, the assignment of the registers to specific functions becomes more undesirable and inflexible; thus, the general register concept has developed. The use of an array of general registers in the processor was apparently first used in the first generation, vacuum-tube machine, PEGASUS [Elliott *et al.*, 1956] and appears to be an outgrowth of both 1- and 2-address structures. (Two alternative structures – the early 2- and 3-address-per-instruction computers may be disregarded, since they tend to always access primary memory for results as well as temporary storage and thus are wasteful of time and memory cycles and require a long instruction.) The stack concept (0-address) provides the most efficient access method for specifying algorithms, since very little space, only the access addresses and the operators, needs to be given. In this scheme the operands of an operator are always assumed to be on the "top of the stack." The stack has the additional advantage that arithmetic expression evaluation and compiler statement parsing have been developed to use a stack effectively. The disadvantage of the stack is due, in part, to the nature of current memory technology. That is, stack memories have to be simulated with random-access memories: multiple stacks are usually required; and even though small stack memories exist, as the stack overflows, the primary memory (core) has to be used.

Even though the trend has been toward the general register concept (which, of course, is similar to a 2-address scheme in which one of the addresses is limited to small values). it is important to recognize that any design is a compromise. There are situations for which any of these schemes can be shown to be "best." The IBM System 360 series uses a general register structure, and their designers [Amdahl *et al.*, 1964] claim the following advantages for the scheme.

1. Registers can be assigned to various functions: base addressing, address calculation, fixed-point arithmetic, and indexing.
2. Availability of technology makes the general register structure attractive.

The System 360 designers also claim that a stack organized machine such as the English Electric KDF 9 [Allmark and Lucking, 1962] or the Burroughs B5000 [Lonergan and King, 1961] has the following disadvantages.

1. Performance is derived from fast registers, not the way they are used.
2. Stack organization is too limiting and requires many copy and swap operations.
3. The overall storage of general registers and stack machines are the same, considering point 2.

*Due, in part, to needs, but mainly to technology that dictates how large the structure can be.

4. Ti
pl
lc
5. S
al
6. V
st

We g
Point 5
is irrele
have th
scheme
with a
structio
eral re
general
tional
general
indexin
or pro
eral re
memo
side w
was of
operat
(accou
ecut
the co
The
11 ap
using
dressi
sider
addr
ory-t
to us
of sc
An
struc
in th
plete
tern
with
oper

4. The stack has a bottom, and when placed in slower memory, there is a performance loss.
5. Subroutine transparency is not easily realized with one stack.
6. Variable length data is awkward with a stack.

We generally concur with points 1, 2, and 4. Point 5 is an erroneous conclusion, and point 6 is irrelevant (that is, general register machines have the same problem). The general register scheme also allows processor implementations with a high degree of parallelism since all instructions of a local block can operate on several registers concurrently. A set of truly general purpose registers should also have additional uses. For example, in the DEC PDP-10, general registers are used for address integers, indexing, floating point, Boolean vectors (bits), or program flags and stack pointers. The general registers are also addressable as primary memory, and thus, short program loops can reside within them and be interpreted faster. It vas observed in operation that PDP-10 stack operations were very powerful and often used (accounting for as many as 20 percent of the executed instructions in some programs, e.g., the compilers).

The basic design decision that sets the PDP-11 apart was based on the observation that by using *truly* general registers and by suitable addressing mechanisms, it was possible to consider the machine as a 0-address (stack), 1-address (general register), or 2-address (memory-to-memory) computer. Thus, it is possible to use whichever addressing scheme, or mixture of schemes, is most appropriate.

Another important design decision for the instruction set was to have only a few data-types in the basic machine, and to have a rather complete set of operations for each data-type. (Alternative designs might have more data-types with few operations, or few data-types with few operations.) In part, this was dictated by the machine size. The conversion between data-types must be accomplished easily either automatically or with one or two instructions. The data-types should also be sufficiently primitive to allow other data-types to be defined by software (and by hardware in more powerful versions of the machine). The basic data-type of the machine is the 16-bit integer which uses the two's complement convention for sign. This data-type is also identical to an address.

PDP-11 Model 20 Instruction Set (Basic Instruction Set)

A formal description of the basic instruction set is given in the original paper [Bell *et al.*, 1970] using the ISPL notation [Bell and Newell, 1970]. The remainder of this section will discuss the machine in a conventional manner.

Primary Memory. The primary memory (core) is addressed as either 2^{16} bytes or 2^{15} words using a 16-bit number. The linear address space is also used to access the input/output devices. The device state, data and control registers are read or written like normal memory locations.

General Register. The general registers are named: R[0:7]<15:0>; that is, there are eight registers each with 16 bits. The naming is done starting at the left with bit 15 (the sign bit) to the least significant bit 0. There are synonyms for R[6] and R[7]:

1. Stack Pointer\SP<15:0>
   := R[6]<@15:0>
   Used to access a special stack that is used to store the state of interrupts, traps, and subroutine calls.
2. Program Counter\PC<15:0>
   := R[7]<@15:0>
   Points to the current instruction being interpreted. It will be seen that the fact that PC is one of the general registers is crucial to the design.

Any general register, R[0:7], can be used as a stack pointer. The special Stack Pointer SP has additional properties that force it to be used for changing processor state interrupts, traps, and subroutine calls. (It also can be used to control dynamic temporary storage subroutines.)

In addition to the above registers there are 8 bits used (from a possible 16) for processor status, called PS<15:0> register. Four bits are the Condition Codes\CC associated with arithmetic results; the T-bit controls tracing; and 3 bits control the priority of running programs Priority <2:0>. Individual bits are mapped in PS as shown in the appendix.

Data-Types and Primitive Operations. There are two data lengths in the basic machine: bytes and words, which are 8 and 16 bits, respectively. The nontrivial data-types are word-length integers (w.i.); byte-length integers (by.i); word-length Boolean vectors (w.bv); i.e., 16 independent bits (Booleans) in a 1-dimensional array; and byte-length Boolean vectors (by.bv). The operations on byte and word Boolean vectors are identical. Since a common use of a byte is to hold several flag bits (Booleans), the operations can be combined to form the complete set of 16 operations. The logical operations are: "clear," "complement," "inclusive or," and "implication" ($x \supset y$ or $\neg x \vee y$).

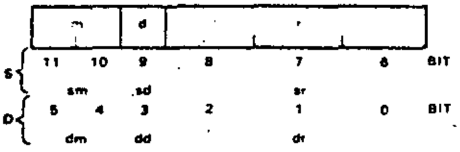
There is a complete set of arithmetic operations for the word integers in the basic instruction set. The arithmetic operations are: "add," "subtract," "multiply" (optional), "divide" (optional), "compare," "add one," "subtract one," "clear," "negate," and "multiply and divide" by powers of two (shift). Since the address integer size is 16 bits, these data-types are most important. Byte-length integers are operated on as words by moving them to the general registers where they take on the value of word integers. Word-length-integer operations are carried out and the results are returned to memory (truncated).

The floating-point instructions defined by software (not part of the basic instruction set) require the definition of two additional data-types (of length two and three), i.e., double words (d.w.) and triple words (t.w.). Two additional data-types, double integer (d.i.) and triple floating-point (t.f. or f) are provided for arithmetic. These data-types imply certain additional operations and the conversion to the more primitive data-types.

Address (Operand) Calculation. The general methods provided for accessing operands are the most interesting (perhaps unique) part of the machine's structure. By defining several access methods to a set of general registers, to memory, or to a stack (controlled by a general register), the computer is able to be a 0-, 1-, and 2-address machine. The encoding of the instruction source (S) fields and destination (D) fields are given in Figure 10 together with a list of the various access modes that are possible. (The appendix gives a formal description of the effective address calculation process.)

It should be noted from Figure 10 that all the common access modes are included (direct, indirect, immediate, relative, indexed, and indexed indirect) plus several relatively uncommon ones. Relative (to PC) access is used to simplify program loading, while immediate mode speeds up execution. The relatively uncommon access modes, auto-increment and auto-decrement, are used for two purposes: access to a stack under control of the registers* and access to bytes or words organized as strings or vectors. The indirect access mode allows a stack to hold addresses of data (instead of data). This mode is desirable when manipulating longer and variable-length data-types (e.g., strings, double fixed, and triple floating-

*Note that, by convention, a stack builds toward register 0, and when the stack crosses 400_8, a stack overflow occurs.



r = REGISTER SPECIFICATION R[r]
d = DEFER (INDIRECT) ADDRESS BIT
m = MODE (00 = R[r], 01 = R[r]; NEXT R[r] + ai; [1]
10 = R[r], R[r] −ai, NEXT R[2]
11 = INDEXED WITH NEXT WORD)

The following access modes can be specified

0 Direct to a register R[r].

1 Indirect to a register, R[r] for address of data.

2 Auto increment via register (pop) – use register as address, then increment register.

3 Auto increment via register (pop) – defer.

4 Auto decrement via register (push) – decrement register, then use register as address

5 Auto decrement indirect – decrement register, then use register as the address of the address of data

2 Immediate data – next full word is the data (r = PC).

3 Direct data – next full word is the address of data (r = PC)

6 Direct indexed – use next full word indexed with R[r] as address of data.

7 Direct indexed – indirect – use next full word indexed with R[r] as the address of the address of data

6 Relative access – next full word plus PC is the address (R = PC)

7 Relative indirect access – next full word plus PC is the address of the address of data (r = PC).

[1] Address increment/ai value is 1 or 2.

Figure 10. Address calculation formats.

point). The register auto-increment mode may be used to access a byte string; thus, for example, after each access, the register can be made to point to the next data item. This is used for moving data blocks, searching for particular elements of a vector, and byte-string operations (e.g., movement, comparisons, editing).

This addressing structure provides flexibility while retaining the same, or better, coding efficiency than classical machines. As an example of the flexibility possible, consider the variations possible with the most trivial word instruction MOVE (Table 1). The MOVE instruction is coded in conventional 2-address, 1-address (general register) and 0-address (stack) computers. The 2-address format is particularly nice for MOVE, because it provides an efficient encoding for the common operation: A ← B (note that the stack and general registers are not involved). The vector moves A[I] ← B(I) is also efficiently encoded. For the general register (and 1-address format), there are about 13 MOVE operations that are commonly used. Six moves can be encoded for the stack (about the same number found in stack machines).

Instruction Formats. There are several instruction decoding formats depending on whether zero, one, or two operands have to be explicitly referenced. When two operands are required, they are identified as source S and destination D and the result is placed at destination D. For single operand instructions (unary operators), the instruction action is D ← u D; and for two operand instructions (binary operators), the action is D ← D b S (where u and b are unary and binary operators, e.g., ¬, – and +, –, ×, /, respectively. Instructions are specified by a 16-bit word. The most common binary operator format (that for operations requiring two addresses) uses bits 15:12 to specify the operation code, bits 11:6 to specify the destination D, and bits 5:0 to specify the source S. The other instruction formats are given in Figure 11.

Instruction Interpretation Process. The instruction interpretation process is given in Figure 12, and follows the common fetch-execute cycle. There are three major states: (1) interrupting – the PC and PS are placed on the stack accessed by the Stack Pointer/SP, and the new state is taken from an address specified by the source requesting the trap or interrupt: (2) trace (controlled by T-bit) – essentially one instruction at a time is executed as a trace trap occurs after each instruction, and (3) normal instruction interpretation. The five (lower) states in the diagram are concerned with instruction fetching, operand fetching, executing the operation specified by the instruction and storing the result. The nontrivial details for fetching and storing the operands are not shown in the diagram but can be constructed from the effective address calculation process (appendix). The

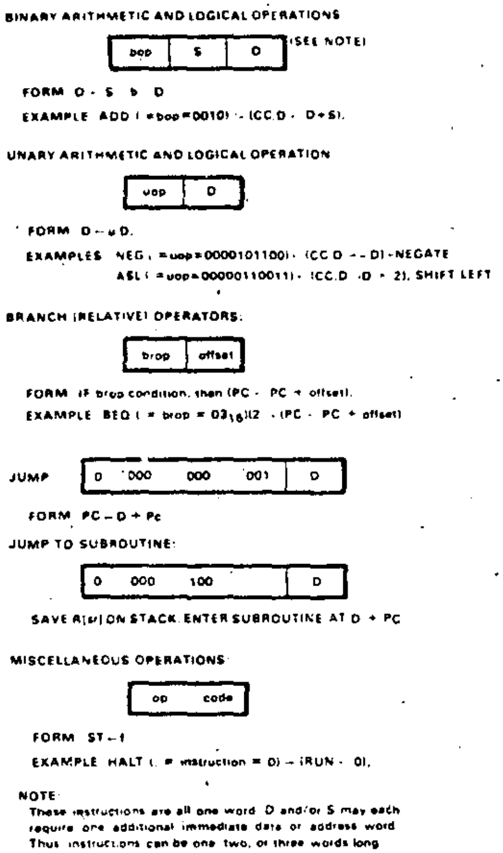

BINARY ARITHMETIC AND LOGICAL OPERATIONS
bop
S
D
(SEE NOTE)
FORM D ← S b D
EXAMPLE ADD (:=bop=0010) → (CC.D ← D+S).
UNARY ARITHMETIC AND LOGICAL OPERATION
uop
D
FORM D ← u D.
EXAMPLES NEG (:=uop=0000101100) → (CC.D ← – D) – NEGATE
ASL (:=uop=0000110011) → (CC.D ← D × 2). SHIFT LEFT
BRANCH (RELATIVE) OPERATORS:
brop
offset
FORM IF brop condition, then (PC ← PC + offset).
EXAMPLE BEQ (:= brop = 03₁₆)(Z → (PC ← PC + offset)
JUMP
0 000 000 001
D
FORM PC ← D + Pc
JUMP TO SUBROUTINE:
0 000 100
D
SAVE R[r] ON STACK. ENTER SUBROUTINE AT D + PC
MISCELLANEOUS OPERATIONS
op code
FORM ST ← f
EXAMPLE HALT (:= instruction = 0) → (RUN ← 0).
NOTE:
These instructions are all one word. D and/or S may each require one additional immediate data or address word. Thus instructions can be one, two, or three words long.

Figure 11. PDP-11 instruction formats (simplified).

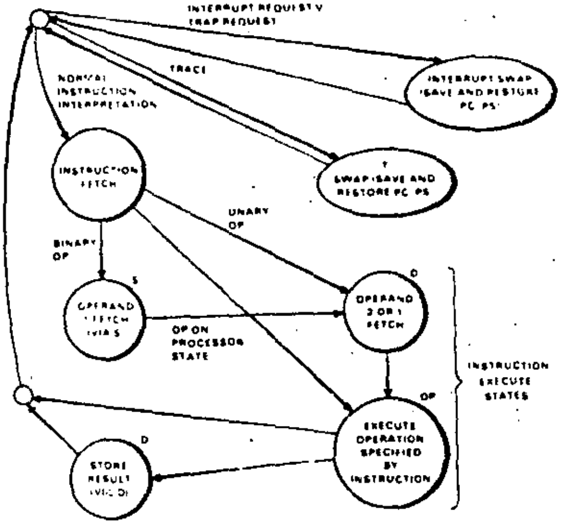


Figure 12 PDP-11 instruction interpretation process state diagram.

state diagram, though simplified, is similar to 2- and 3-address computers, but is distinctly different than a 1-address (1-accumulator) computer.

The ISP description (appendix) gives the operation of each of the instructions, and the more conventional diagram (Figure 11) shows the decoding of instruction classes. The ISP description is somewhat incomplete; for example, the add instruction is defined as:

$$\text{ADD } (:= \text{bop} = 0010_2) \Rightarrow (\text{CC.D} \leftarrow \text{D} + \text{S})$$

Addition does not exactly describe the changes to the Condition Codes CC (which means whenever a binary opcode [bop] of 0010_2 occurs the ADD instruction is executed with the above effect). In general, the CC are based on the result, that is, Z is set if the result is zero, N if negative, C if a carry occurs, and V if an overflow was detected as a result of the operation. Conditional branch instructions may thus follow the arithmetic instruction to test the results of the CC bits.

Examples of Addressing Schemes

Use as a Stack (Zero-Address) Machine. Table 2 lists typical 0-address machine instructions together with the PDP-11 instructions that perform the same function. It should be noted that translation (compilation) from normal infix expressions to reverse Polish is a comparatively trivial task. Thus, one of the primary reasons for using stacks is for the evaluation of expressions in reverse Polish form.

Consider an assignment statement of the form:

$$D \leftarrow A + B/C$$

Table 1. Coding for the MOVE Instruction To Compare with Conventional Machines

Assembler Format	Effect	Description
2-Address Machine Format		
MOVE.B. A*	A ← B	Replace A with contents of B
MOVE #N. A	A ← N	Replace A with number B
MOVE B(RZ). A(RZ)	A[I] ← B[I]	Replace element of a connector

MOVE (R3)+. (R4)+	A[I] ← B[I]:
	I ← I + 1
		Replace element of a vector. move to next element

General-Register Machine Format		
MOVE A. R1	R1 ← A	Load register
MOVE R1. A	A ← R1	Store register
MOVE @A. R1	R1 ← M[A]	Load or store indirec' via element A
MOVE R1. R3	R1 ← R3	Register-to-register transfer
MOVE R1. A(R1)	A[I] ← R1	Store indexed (load indexed) (or store)
MOVE @A(RO). R1	R1 ← M[A[I]]	Load (or store) indexed indirect
MOVE (R1). R3	R1 ← M[R2]	Load indirect via register
MOVE (R1)+. R3	R3 ←M[I]	Load (or store) element indirect via register. move to next element
Stack Machine Format		
MOVE #N. –(RO)	S ← N	Load stack with literal
MOVE A. –(RO)	S ← A	Load stack with contents of A
MOVE @(RO)+. –(RO)	S ← M[S]	Load stack with memory specified by top of stack
MOVE (RO)+. A	A ← S	Store stack in A
MOVE (RO)+. @(RO)+	M[S_2] ← S_1	Store stack top in memory addressed by stack top -1
MOVE (RO). –(RO)	S ← S	Duplicate top of stack

*Assembler Format
( ) Denotes contents of memory addressed by
- Decrement register first
+ Increment register after
@ Indirect
# Literal

which has the reverse Polish form:

DABC/ + ←

and would normally be encoded on a stack machine as follows:

Load stack address of D
Load stack A
Load stack B
Load stack C
/
+
Store.

However, with the PDP-11, there is an address method for improving the program encoding and run time, while not losing the stack concept. An encoding improvement is made by doing an operation to the top of the stack from a direct-memory location (while loading). Thus, the previous example could be coded as:

Load stack B
Divide stack by C
Add A to stack
Store stack D

Use as a 1-Address (General Register) Machine. The PDP-11 is a general register computer and should be judged on that basis. Benchmarks have been coded to compare the transactions operate independently of the bus length and response time of the master and slave. Since the bus is bidirectional and is used by all devices, any device can communicate with any other device. The controlling device is the master, and the device to which the master is communicating is the slave. For example, a data transfer from processor (master) to memory (always a slave) uses the Data Out dialogue facility for writing and a transfer from memory to processor uses the Data In dialogue facility for reading.

Bus Control. Most of the time the processor is bus master fetching instructions and operands from memory and storing results in memory. Bus mastership is determined by the current processor priority and the priority line upon which a bus request is made and the physical placement of a requesting device on the linked bus. The assignment of bus mastership is done concurrent with normal communication (dialogues).

Unibus Dialogues

Three types of dialogues use the Unibus. All the dialogues have a common protocol that first consists of obtaining the bus mastership (which is done concurrent with a previous transaction) followed by a data exchange with the requested device. The dialogues are: Interrupt; Data In and Data In Pause; and Data Out and Data Out Byte.

Interrupt. Interrupt can be initiated by a master immediately after receiving bus mastership. An address is transmitted from the master to the slave on Interrupt. Normally, subordinate control devices use this method to transmit an interrupt signal to the processor.

Data In and Data In Pause. These two bus operations transmit slave's data (whose address is specified by the master) to the master. For the Data In Pause operation, data is read into the master and the master responds with data which is to be rewritten in the slave.

Data Out and Data Out Byte. These two operations transfer data from the master to the slave at the address specified by the master. For Data Out, a word at the address specified by the address lines is transferred from master to slave. Data Out Byte allows a single data byte to be transmitted.

Processor Logical Design

The Pc is designed using TTL logical design components and occupies approximately eight 8 inch × 12 inch printed circuit boards. The Pc is physically connected to two other components, the console and the Unibus. The control for the Unibus is housed in the Pc and occupies one of the printed circuit boards. The most regular part of the Pc is the arithmetic and state section. The 16-word scratchpad memory and combinational logic data operators, D (shift) and D (adder, logical ops), form the most regular part of the processor's structure. The 16-word memory holds most of the 8-word processor state found in the ISP, and the 8 bits that form the Status word are stored in an 8-bit register. The input to the adder-shift network has two latches which are either memories or gates. The output of the adder-shift network can be read to either the data or address parts of the Unibus, or back to the scratchpad array.

The instruction decoding and arithmetic control are less regular than the above data and state and these are shown in the lower part of the figure. There are two major sections: the instruction fetching and decoding control and the instruction set interpreter (which, in effect, defines the ISP). The later control section operates on, hence controls, the arithmetic and state parts of the Pc. A final control is concerned with the interface to the Unibus (distinct from the Unibus control that is housed in the Pc).

CONCLUSIONS

In this paper we have endeavored to give a complete description of the PDP-11 Model 20 computer at four descriptive levels. These present an unambiguous specification at two levels (the PMS structure and the ISP), and, in addition, specify the constraints for the design at the top level, and give the reader some idea of the implementation at the bottom level logical design. We have also presented guidelines for forming additional models that would belong to the same family.

ACKNOWLEDGEMENTS

The authors are grateful to Mr. Nigberg of the technical publication department at DEC and to the reviewers for their helpful criticism. We are especially grateful to Mrs. Dorothy Josephson at Carnegie-Mellon University for typing the notation-laden manuscript.

APPENDIX. DEC PDP-11 INSTRUCTION SET PROCESSOR DESCRIPTION (IN ISPL)

The following description gives a cursory description of the instructions in the ISPL, the initial notation of Bell and Newell [1971]. Only the processor state and a brief description of the instructions are given.

Primary Memory State

M\Mp\Memory $[0:2^{16}-1]<7:0>$	Byte memory
Mw$[0:2^{15}-1]<15:0> := M[0:2^{16}-1]<7:0>$	Word memory mapping

Processor State (9 words)

R\Registers [0:7]<15:0>	Word general registers
SP<15:0> := R[6]<15:0>	Stack pointer
PC<15:0> := R[7]<15:0>	Program counter
PS<15:0>	Processor state register
Priority\P<2:0> := PS<7:5>	Under program control; priority level of the process currently being interpreted; a higher level process may interrupt or trap this process.
CC\Condition-Codes<3:0> := PS<3:0>	
Carry\C := CC<0>	A result condition code indicating an arithmetic carry from bit 15 of the last operation.
Negative\N := CC<3>	A result condition code indicating last result was negative.
Zero\Z := CC<2>	A result condition code indicating last result was zero.

Overflow\V := CC<1>	A result condition code indicating an arithmetic overflow of the last operation.
Trace\T := ST<4>	Denotes whether instruction trace trap is to occur after each instruction is executed.
Undefined<7:0> := PS<15:8>	Unused
Run	Denotes normal execution.
Wait	Denotes waiting for an interrupt.

Instruction Set

The following instruction set will be defined briefly and is incomplete. It is intended to give the reader a simple understanding of the machine operation.

MOV (:= bop = 0001) → (CC,D ← S);	Move word
MOVB (:= bop = 1001) → (CC,Db ← Sb);	Move byte

Binary Arithmetic: D ← D b S;

ADD (:= bop = 0110) → (CC,D ← D + S);	Add
SUB (:= bop = 1110) → (CC,D ← D − S);	Subtract
CMP (:= bop = 0010) → (CC ← D − S);	Word compare
CMPB (:= bop = 1010) → (CC ← Db − Sb);	Byte compare
MUL (:= bop = 0111) → (CC, D ← D × S)	Multiply, if D is a register then a double length operator
DIV (:= bop = 1111) → (CC, D ← D/S);	Divide, if D is a register, then a remainder is saved

Unary Arithmetic: D ← uS;

CLR (:= uop = 050_8) → (CC,D ← 0);	Clear word
CLRB (:= uop = 1050_8) → (CC,Db ← 0);	Clear byte
COM (:= uop = 051_8) → (CC,D ← ¬D);	Complement word
COMB (:= uop = 1051_8) → (CC,Db ← ¬Db);	Complement byte
INC (:= uop = 052_8) → (CC,D ← D + 1);	Increment word
INCB (:= uop = 1052_8) → (CC,Db ← Db + 1);	Increment byte
DEC (:= uop = 053_8) → (CC,D ← D − 1);	Decrement word
DECB (:= uop = 1053_8) → (CC,Db ← Db − 1);	Decrement byte
NEG (:= uop = 054_8) → (CC,D ← − D);	Negate
NEGB (:= uop = 1054_8) → (CC,Db ← − Db)	Negate byte
ADC (:= uop = 055_8) → (CC,D ← D + C);	Add the carry
ADCB (:= uop = 1055_8) → (CC,Db ← Db + C);	Add to byte the carry
SBC (:= uop = 056_8) → (CC,D ← D − C);	Subtract the carry

SBCB (:= uop = 1056_8) → (CC,Db ← Db − C);	Subtract from byte the carry
TST (:= uop = 057_8) → (CC ← D);	Test
TST (:= uop = 1057_8) → (CC ← Db);	Test byte

Shift Operations: $D \leftarrow D \times 2^n$;

ROR (:= sop = 060_8) → (C □ D ← C □ D/2{rotate});	Rotate right
RORB (:= sop = 1060_8) → (C □ Db ← C □ Db/2{rotate});	Byte rotate right
ROL (:= sop = 061_8) → (C □ D ← C □ D × 2 {rotate});	Rotate left
ROLB (:= sop = 1061_8) → (C □ Db ← C □ Db × 2 {rotate});	Byte rotate left
ASR (:= sop = 062_8) → (CC,D ← D × 2);	Arithmetic shift right
ASRB (:= sop = 1062_8) → (CC,Db ← Db/2);	Byte arithmetic shift right
ASL (:= sop = 063_8) → (CC,D ← D × 2);	Arithmetic shift left
ASLB (:= sop = 1063_8) → (CC,Db ← Db × 2);	Byte arithmetic shift left
ROT (:= sop = 064_8) → (C □ D ← D × 2^S);	Rotate
ROTB (:= sop = 1064_8) → (C □ Db ← D × 2^S);	Byte rotate
LSH (:= sop = 065_8) → (CC,D ← D × 2^S{logical});	Logical shift
LSHB (:= sop = 1065_8) → (CC,Db ← Db × 2^S{logical});	Byte logical shift
ASH (:= sop = 066_8) → (CC,D ← D × 2^S);	Arithmetic shift
ASHB (:= sop = 1066_8) → (CC,Db ← Db × 2^S);	Byte arithmetic shift

NOR (:= sop = 067_8 → (CC,D ← normalize (D));
(R[r'] → normalize␣exponent (D));
	Normalize

NORD (:= sop = 1067_8 → (Db ←normalize (Dd));
(R[r'] ← normalize␣exponent (D));
	Normalize double

SWAB (:= sop = 3) → (CC,D ← D<7:0, 15:8>)	Swap bytes

Logical Operations

BIC (:= bop = 0100) → (CC,D ← D ← D ∧ ¬S);	Bit clear
BICB (:= bop = 1100) → (CC,Db ← Db ∨ ¬Sb);	Byte bit clear
BIS (:= bop = 0101) → (CC,D ← D ∨ S);	Bit set
BISB (:= bop = 1101 → (CC,Db ← Db ∨ Sb);	Byte bit set
BIT (:= bop = 0011) → (CC ← D ∧ S);	Bit test under mask
BITB (:= bop = 1011) → (CC ← Db ∧ Sb);	Byte bit test under mask

Branches and Subroutines Calling: PC ← f;

JMP (:= sop = 0001_8) → (PC ← D');	Jump unconditional
BR (:= brop = 01_{16}) → (PC ← PC + offset);	Branch unconditional
BEQ (:= brop = 03_{16}) → (Z → (PC ← PC + offset));	Equal to zero
BNE (:= brop = 02_{16}) → (¬Z → (PC ← PC + offset));	Not equal to zero
BLT (:= brop = 05_{16}) → (N ⊕ V → (PC ← PC + offset));	Less than (zero)
BGE (:= brop = 04_{16}) → (N ≡ V → (PC ← PC + offset));	Greater than or equal (zero)
BLE (:= brop = 07_{16}) → (Z ∨ (N ⊕ V) → (PC ← PC + offset));	Less than or equal (zero)

BGT (:= brop = 06_{16}) → (¬(Z ∨ (N ⊕ V)) → (PC ← PC + offset));	Less greater than (zero)
BCS/BHIS (:= brop = 87_{16}) → (C → (PC ← PC + offset));	Carry set: higher or same (unsigned)
BCC/BLO (:= brop = 86_{16}) → (¬C → (PC ← PC + offset));	Carry clear; lower (unsigned)
BLOS (:= brop = 83_{16}) → (C ∧ Z → (PC ← PC + offset));	Lower or same (unsigned)
BHI (:= brop = 82_{16}) → ((¬C ∨ Z) → (PC ← PC + offset));	Higher than (unsigned)
BVS (:= brop = 85_{16}) → (V → (PC ← PC + offset));	Overflow
BVC (:= brop = 84_{16}) → (¬V → (PC ← PC + offset));	No overflow
BMT (:= brop = 81_{16}) → (N → (PC ← PC + offset));	Minus
BPL (:= brop = 80_{16}) → (¬N → (PC ← PC + offset));	Plus
JSR (:= sop = 0040_8) →	Jump to subroutine by putting
(SP ← SP − 2; next	R[sr], PC on stack and loading
M[SP] ← R[sr];	R[sr] with PC, and going to
R[sr] ← PC; PC ← D);	subroutine at D)
RTS(: = i = 000200_8) → (PC ← R[dr];	Return from subroutine
R[dr] ← M[SP]; SP ← SP + 2);	

Miscellaneous Processor State Modification:

RTI (: = i = 2_8) → (PC ← M[SP];	Return from interrupt
SP ← SP + 2; next	
PS ← M[SP];	
SP ← SP + 2);	
HALT (: = i = 0) → (Run ← 0);	
WAIT (: = i = 1) → (Wait ← 1);	
TRAP (: = i = 3) → (SP ← SP + 2; next	Trap to M[34_8] store status and PC
M[SP] ← PS;	
SP ← SP + 2; next	
M[SP] ← PC;	
PC ← M[34_8];	Enter new process
PS ← M[12]);	Emulator trap
EMT (: = brop − 82_{16}) → (SP ← SP + 2; next	
M[SP] ← PS;	
SP ← SP + 2; next	
M[SP] ← PC;	
PC ← M[30_8];	
PS ← M[32_8]);	
IOT (: = i = 4) → (see TRAP)	I/O trap to M[20_8]
RESET (: = i = 5) → (not described)	Reset to external devices
OPERATE(: = i<5:15> = 5) →	Condition code operate
(i<4> → (CC ← CC ∨ i<3:0>);	Set codes
¬i<4> → (CC ← CC ∧ ¬i<3:0>));	Clear codes

end Instruction⎵execution

16

The Evolution of the PDP-11

C. GORDON BELL and J. CRAIG MUDGE

A computer is not solely determined by its architecture; it reflects the technological, economic, and organizational aspects of the environment in which it was designed and built. In the introductory chapters the nonarchitectural design factors were discussed: the availability and price of the basic electronic technology, the various government and industry rules and standards, the current and future market conditions, and the manufacturing process.

In this chapter one can see the result of the interaction of these various forces in the evolution of the PDP-11. Twelve distinct models (LSI-11, PDP-11/04, 11/05, 11/20, 11/34, 11/34C, 11/40, 11/45, 11/55, 11/60, 11/70, and VAX-11/780) exist in 1978.

The PDP-11 has been successful in the marketplace: over 50,000 were sold in the first eight years that it was on the market (1970–1977). It is not clear how rigorous a test (aside from the marketplace) the design has been given, since a large and aggressive marketing organization, armed with software to correct architectural inconsistencies and omissions, can save almost any design.

Many ideas from the PDP-11 have migrated to other computers with newer designs. Although some of the features of the PDP-11 are patented, machines have been made with similar bus and instruction set processor structures. Many computer designers have adopted a unified data and address bus similar to the Unibus as their fundamental architectural component. Many microprocessor designs incorporate the PDP-11 Unibus notion of mapping I/O and control registers into the memory address space, eliminating the need for I/O instructions without complicating the I/O control logic.

It is the nature of computer engineering to be goal-oriented, with pressure to produce deliverable products. It is therefore difficult to plan for an extensive lifetime. Nevertheless, the PDP-11 evolved rapidly over a much wider range than expected. An outline of a family plan was set forth in a memo on April 3, 1969, by Roger Cady, head of the PDP-11 engineering group at the time (Table 1). The actual evolution is shown in tree form in Figure 1 and is mapped onto a cost/performance representation in Figure 2.

Table 1. PDP-11 Family Projection as of April 3, 1969

Model	Processor	Logic Power	Arithmetic Power	Speed (μs)	Price ($K)	Configuration	Software Paper Tape	Disk
11/10	–	0.7	0.7	2-3	4	Technologically cost reduced 11/20 with Mos		
11/20	KA11	1	1	2.2	5.2	Pc, 1-Kbyte ROM, 128 byte R/W turnkey console		
11/30	KA11	1	1	2.2	9.3	Pc, 8-Kbyte core, console, TTY	Assembler, editor, math utility FOCAL, BASIC, ASA BASIC FORTRAN)‡	8-like monitor (system builder w/ODT, DDT, PIP)†
11/40	KB11	2*	10-20	1.2	13	Adds *, /, normalize, etc. possible microprogrammed processor, no EAE saves $1,000	Possible 16-Kbyte FORTRAN IV improved assembler	FORTRAN IV
11/45	KB11	2*	10-20	1.2	15 + disk	11/45 with memory protect/relocate maximum core 262 Kbyte, maximum physical memory (using disk) 2^{22} bytes	–	Super monitor** 65-Kbyte virtual memory/user for either small or large disk
11/50	KC11	2*	50-100	1.2	25	Adds hardware floating point 32-bit processor, 16-bit memory (16 Kbyte)	–	–
11/55	KC11	2*	50-100	1.2	27 + disk	With memory protect/relocate		
11/65	KD11	4	100-200	1.2 32-bit	45 + disk	32-bit separate memory bus, 32-bit processor		

NOTES:

*If microprogrammed, then logical power could be tailored to user and go to 20-50, 40-100 for 11/65.

†Business language system under consideration.

‡Possible by-product of FOCAL.

**Super monitor for 11/45, 11/55, 11/65 is priority multi-user real-time system.

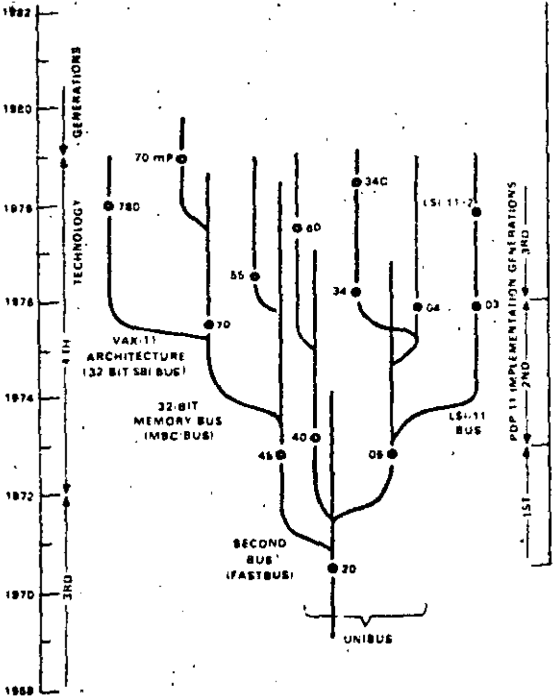


Figure 1. The PDP-11 Family tree.

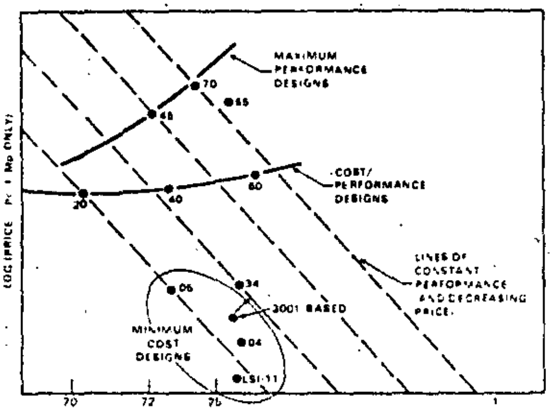


Figure 2. PDP-11 models price versus time with lines of constant performance.

EVALUATION AGAINST THE ORIGINAL GOALS

In the original 1970 PDP-11 paper (Chapter 9), a set of design goals and constraints were given, beginning with a discussion of the weaknesses frequently found in minicomputers. The designers of the PDP-11 faced each of these known minicomputer weaknesses; and their goals included a solution to each one. This section reviews the original goals, commenting on the success or failure of the PDP-11 in meeting each of them.

The weaknesses of prior designs that were noted were limited addressability, a small number of registers, absence of hardware stack facilities, limited interrupt structures, absence of byte string handling and read-only memory facilities, elementary I/O processing, absence of growth-path family members, and high programming costs.

The first weakness of minicomputers was their limited addressing capability. The biggest (and most common) mistake that can be made in a computer design is that of not providing enough address bits for memory addressing and management. The PDP-11 followed this hallowed tradition of skimping on address bits, but it was saved by the principle that a good design can evolve through at least one major change.

For the PDP-11, the limited address problem was solved for the short run, but not with enough finesse to support a large family of minicomputers. That was indeed a costly oversight, resulting in both redundant development and lost sales. It is extremely embarassing that the PDP-11 had to be redesigned with memory management* only two years after writing the paper that outlined the goal of providing increased address space. All earlier DEC designs suffered from the same problem, and only the

* The memory management served two other functions besides expanding the 16-bit processor-generated addresses into 18-bit Unibus addresses: program relocation and protection.

PDP-10 evolved over a long period (15 years) before a change occurred to increase its address space. In retrospect, it is clear that another address bit is required every two or three years, since memory prices decline about 30 percent yearly, and users tend to buy constant price successor systems.

A second weakness of minicomputers was their tendency to skimp on registers. This was corrected for the PDP-11 by providing eight 16-bit registers. Later, six 64-bit registers were added as the accumulators for floating-point arithmetic. This number seems to be adequate: there are enough registers to allocate two or three registers (beyond those already dedicated to program counter and stack pointer) for program global purposes and still have registers for local statement computation.* More registers would increase the context switch time and worsen the register allocation problem for the user.

A third weakness of minicomputers was their lack of hardware stack capability. In the PDP-11, this was solved with the autoincrement/autodecrement addressing mechanism. This solution is unique to the PDP-11, has proved to be exceptionally useful, and has been copied by other designers. The stack limit check, however, has not been widely used by DEC operating systems.

A fourth weakness, limited interrupt capability and slow context switching, was essentially solved by the Unibus interrupt vector design. The basic mechanism is very fast, requiring only four memory cycles from the time an interrupt request is issued until the first instruction of the interrupt routine begins execution. Implementations could go further and save the general registers, for example, in memory or in special registers. This was not specified in the architecture and has not been done in any of the implementations to date. VAX-11 provides explicit load and save process context instructions.

A fifth weakness of earlier minicomputers, inadequate character handling capability, was met in the PDP-11 by providing direct byte addressing capability. String instructions were not provided in the hardware, but the common string operations (move, compare, concatenate) could be programmed with very short loops. Early benchmarks showed that this mechanism was adequate. However, as COBOL compilers have improved and as more understanding of operating systems string handling has been obtained, a need for a string instruction set was felt, and in 1977 such a set was added.

A sixth weakness, the inability to use read-only memories as primary memory, was avoided in the PDP-11. Most code written for the PDP-11 tends to be reentrant without special effort by the programmer, allowing a read-only memory (ROM) to be used directly. Read-only memories are used extensively for bootstrap loaders, program debuggers, and for simple functions. Because large read-only memories were not available at the time of the original design, there are no architectural components designed specifically with large ROMs in mind.

A seventh weakness, one common to many minicomputers, was primitive I/O capabilities. The PDP-11 answers this to a certain extent with its improved interrupt structure, but the completely general solution of I/O computers has not yet been implemented. The I/O processor concept is used extensively in display processors, in communication processors, and in signal processing. Having a single machine instruction that transmits a block of data at the interrupt level would decrease the central processor overhead per character by a factor of 3; it

* Since dedicated registers are used for each Commercial Instruction Set (CIS) instruction, this was no longer true when CIS was added.

should have been added to the PDP-11 instruction set for implementation on all machines. Provision was made in the 11/60 for invocation of a micro-level interrupt service routine in writable control store (WCS), but the family architecture is yet to be extended in this direction.

Another common minicomputer weakness was the lack of system range. If a user had a system running on a minicomputer and wanted to expand it or produce a cheaper turnkey version, he frequently had no recourse, since there were often no larger and smaller models with the same architecture. The PDP-11 has been very successful in meeting this goal.

A ninth weakness of minicomputers was the high cost of programming caused by programming in lower level languages. Many users programmed in assembly language, without the comfortable environment of high-level languages, editors, file systems, and debuggers available on bigger systems. The PDP-11 does not seem to have overcome this weakness, although it appears that more complex systems are being successfully built with the PDP-11 than with its predecessors, the PDP-8 and the PDP-15. Some systems programming is done using higher level languages; however, the optimizing compiler for BLISS-11 at first ran only on the PDP-10. The use of BLISS has been slowly gaining acceptance. It was first used in implementing the FORTRAN-IV PLUS (optimizing) compiler. Its use in PDP-10 and VAX-11 systems programming has been more widespread.

One design constraint that turned out to be expensive, but worth it in the long run, was the necessity for the word length to be a multiple of eight bits. Previous DEC designs were oriented toward 6-bit characters, and DEC had a large investment in 12-, 18-, and 36-bit systems, as described in Parts II and V.

Microprogrammability was not an explicit design goal, partially because fast, large, and inexpensive read-only memories were not available at the time of the first implementation. All subsequent machines have been microprogrammed, but with some difficulty because some parts of the instruction set processor, such as condition code setting and instruction register decoding, are not ideally matched to microprogrammed control.

The design goal of understandability seems to have received little attention. The PDP-11 was initially a hard machine to understand and was marketable only to those with extensive computer experience. The first programmers' handbook was not very helpful. It is still unclear whether a user without programming experience can learn the machine solely from the handbook. Fortunately, several computer science textbooks [Gear, 1974; Eckhouse, 1975; Stone and Siewiorek, 1975] and other training books have been written based on the PDP-11.

Structural flexibility (modularity) for hardware configurations was an important goal. This succeeded beyond expectations and is discussed extensively in the Unibus Cost and Performance section.

EVOLUTION OF THE INSTRUCTION SET PROCESSOR

Designing the instruction set processor level of a machine – that collection of characteristics such as the set of data operators, addressing modes, trap and interrupt sequences, register organization, and other features visible to a programmer of the bare machine – is an extremely difficult problem. One has to consider the performance (and price) ranges of the machine family as well as the intended applications, and difficult tradeoffs must be made. For example, a wide performance range argues for different encodings over the range: for small systems a byte-oriented approach with small addresses is optimal, whereas larger systems require more operation codes, more registers, and larger addresses. Thus, for larger machines, instruction coding efficiency can be traded for performance.

The PDP-11 was originally conceived as a small machine, but over time its range was gradually extended so that there is now a factor of 500 in price ($500 to $250,000) and memory size (8 Kbytes to 4 Mbytes*) between the smallest and largest models. This range compares favorably with the range of the IBM System 360 family (16 Kbytes to 4 Mbytes). Needless to say, a number of problems have arisen as the basic design was extended.

Chronology of the Extensions

A chronology of the extensions is given in Table 2. Two major extensions, the memory management and the floating point, occurred with the 11/45. The most recent extension is the Commercial Instruction Set, which was defined to enhance performance for the character string and decimal arithmetic data-types of the commercial languages (e.g., COBOL). It introduced the following to the PDP-11 architecture:

1. Data-types representing character sets, character strings, packed decimal strings, and zoned decimal strings.
2. Strings of variable length up to 65 Kcharacters.
3. Instructions for processing character strings in each data-type (move, add, subtract, multiply, divide).
4. Instructions for converting among binary integers, packed decimal strings, and zoned decimal strings.
5. Instructions to move the descriptors for the variable length strings.

The initial design did not have enough operation code space to accommodate instructions for new data-types. Ideally, the complete set of operation codes should have been specified at initial design time so that extensions would fit. With this approach, the uninterpreted operation codes could have been used to call the various operation functions, such as a floating-point addition. This would have avoided the proliferation of run-time support systems for the various hardware/software floating-point arithmetic methods (Extended Arithmetic Element, Extended Instruction Set, Floating Instruction Set, Floating-Point Processor). The extracode technique was used in the Atlas and Scientific Data Systems (SDS) designs, but these techniques are overlooked by most computer designers. Because the complete instruction set processor (or at least an extension framework) was unspecified in the initial design, completeness and orthogonality have been sacrificed.

At the time the PDP-11/45 was designed, several operation code extension schemes were examined: an escape mode to add the floating-point operations, bringing the PDP-11 back to being a more conventional general register machine by reducing the number of addressing modes, and finally, typing the data by adding a global mode that could be switched to select floating point instead of byte operations for the same operation codes. The floating-point instruction set, introduced with the 11/45, is a version of the second alternative.

It also became necessary to do something about the small address space of the processor. The Unibus limits the physical memory to the 262,144 bytes addressable by 18-bits. In the PDP-11/70, the physical address was extended to 4 Mbytes by providing a Unibus map so that devices in a 256 Kbyte Unibus space could transfer into the 4-Mbyte space via mapping registers. While the physical address limits are acceptable for both the Unibus and larger systems, the address for a single program is still confined to an instantaneous space of 16 bits, the user virtual address. The main method of

*Although 22 bits are used, only 2 megabytes can be utilized in the 11/70.

Table 2. Chronology of PDP-11 Instruction Set Processor (ISP) Evolution

Model(s)	Evolution
11/20	Base ISP (16-bit virtual address) and PMS (16-bit processor physical memory address) Unibus with 18-bit addressing
11/20	Extended Arithmetic Element (hardware multiply/divide)
11/45 (11/55,11/70, 11/60,11/34)	Floating-point instruction set with 6 additional registers (46 instructions) in the Floating-Point Processor
11/45 (11/55,11/70)	Memory management (KT11C), 3 modes of protection (Kernel, Supervisor, User); 18-bit processor physical addressing; 16-bit virtual addressing in 8 segments for both instruction and data spaces
11/45 (11/55,11/70)	Extensions for second set of general registers and program interrupt request
11/40 (11/03)	Extended Instruction Set for multiply/divide; floating-point instruction set (4 instructions)
11/40 (11/34,11/60)	Memory Management (KT11D), 2 modes of protection (Kernel, User); 18-bit processor physical addressing; 16-bit virtual addressing in 8 segments
11/70	22-bit processor physical addressing; Unibus map for peripheral controller 22-bit addressing
11/70 (11/60)	Error register accessibility for on-line diagnosis and retry (e.g., cache parity error)
11/03 (11/04,11/34)	Program access to processor status register via explicit instruction (versus Unibus address)
11/03	One level program interrupt
11/60	Extended Function Code for invocation of user-written microcode
VAX-11/780	VAX architectural extensions for 32-bit virtual addressing; VAX ISP
11/03	Commercial Instruction Set (CIS)
11/70mP	Interprocessor Interrupt and System Timers for multiprocessor

dealing with relatively small addresses is via process-oriented operating systems that handle many small tasks. This is a trend in operating systems, especially for process control and transaction processing. It does, however, enforce a structuring discipline in (user) program organization. The RSX-11 series of operating systems for the PDP-11 are organized this way, and the need for large addresses is lessened.

The initial memory management proposal to extend the virtual memory was predicated on dynamic, rather than static, assignment of memory segment registers. In the current memory management scheme, the address registers are usually considered to be static for a task (although some operating systems provide functions to get additional segments dynamically).

With dynamic assignment, a user can address a number of segment names, via a table, and directly load the appropriate segment registers. The segment registers act to concatenate additional address bits in a base address fashion. There have been other schemes proposed that extend the addresses by extending the length of the general registers – of course, extended addresses propagate throughout the design and include double length address variables. In effect, the extended part is loaded with a base address.

With larger machines and process-oriented operating systems, the context switching time becomes an important performance factor. By providing additional registers for more processes, the time (overhead) to switch context from one process (task) to another can be reduced. This option has not been used in the operating system implementations of the PDP-11s to date, although the 11/45 extensions included a second set of general registers. Various alternatives have been suggested, and to accomplish this effectively requires additional operators to handle the many aspects of process scheduling. This extension appears to be relatively unimportant since the range of computers coupled with networks tends to alleviate the need by increasing the real parallelism (as opposed to the apparent parallelism) by having various independent processors work on the separate processes in parallel. The extensions of the PDP-11 for better control of I/O devices is clearly more important in terms of improved performance.

Architecture Management

In retrospect, many of the problems associated with PDP-11 evolution were due to the lack of an ongoing architecture management function. As can be seen from Table 1, the notion of planned evolution was very strong at the beginning. However, a formal architecture control function was not set up until early in 1974. In some sense this was already too late – the four PDP-11 models designed by that date (11/20, 11/05, 11/40, 11/45) had incompatibilities between them. The architecture control function since then has ensured that no further divergence (except in the LSI-11) took place in subsequent models, and in fact resulted in some convergence. At the time the Commercial Instruction Set was added, an architecture extension framework was adopted. Insufficient encodings existed to provide a large number of additional instructions using the same encoding style (in the same space) as the basic PDP-11, i.e., the operation code and operand specifier addressing mode specifiers within a single 16-bit word. An instruction extension framework was adopted which utilized a full word as the opcode, with operand addressing mode specifiers in succeeding instruction stream words along the lines of VAX-11. This architectural extension permits 512 additional opcodes, and instructions may have an unlimited number of operand addressing mode specifiers. The architecture control function also had to deal with the Unibus address space problem.

With VAX-11, architecture management has been in place since the beginning. A definition of the architecture was placed under formal change control well before the VAX-11/780 was built, and both hardware and software engineering groups worked with the same document. Another significant difference is that an extension framework was defined in the original architecture.

An Evaluation

The criteria used to decide whether or not to include a particular capability in an instruction set are highly variable and border on the artistic.* Critics ask that the machine appear elegant, where elegance is a combined quality of instruction formats relating to mnemonic significance, operator/data-type completeness and orthogonality, and addressing consistency. Having completely general facilities (e.g., registers) which are not context dependent assists in minimizing the number of instruction types and in increasing understandability (and usefulness). The authors feel that the PDP-11 has provided this.

At the time the Unibus was designed, it was felt that allowing 4 Kbytes of the address space for I/O control registers was more than enough. However, so many different devices have been interfaced to the bus over the years that it is no longer possible to assign unique addresses to every device. The architectural group has thus been saddled with the chore of device address bookkeeping. Many solutions have been proposed, but none was soon enough; as a result, they are all so costly that it is cheaper just to live with the problem and the attendant inconvenience.

Techniques for generating code by the human and compiler vary widely and thus affect instruction set processor design. The PDP-11 provides more addressing modes than nearly any other computer. The eight modes for source

*Today one would use the *S*, *M*, and *R* measures and methodology defined in Appendix 3.

and destination with dyadic operators provide what amounts to 64 possible ADD instructions. By associating the Program Counter and Stack Pointer registers with the modes, even more data accessing methods are provided. For example, 18 varieties of the MOVE instruction can be distinguished as the machine is used in two-address, general register, and stack machine program forms. (There is a price for this generality – namely, fewer bits could have been used to encode the address modes that are actually used most of the time.)

How the PDP-11 Is Used

In general, the PDP-11 has been used mostly as a general register (i.e., memory to registers) machine. This can be seen by observing the use frequency from Strecker's data (Chapter 14). In one case, it was observed that a user who previously used a one-accumulator computer (e.g., PDP-8), continued to do so. A general register machine provides the greatest performance, and the cost (in terms of bits) is the same as when used as a stack machine. Some compilers, particularly the early ones, are stack oriented since the code production is easier. In principle, and with much care, a fast stack machine could be constructed. However, since most stack machines use primary memory for the stack, there is a loss of performance even if the top of the stack is cached. While a stack is the natural (and necessary) structure to interpret the nested block structure languages, it does not necessarily follow that the interpretation of all statements should occur in the context of the stack. In particular, the predominance of register transfer statements are of the simple 2- and 3-address forms:

$$D \leftarrow S$$

and

$$D1(\text{index } 1) \leftarrow f(S2(\text{index } 2), S3(\text{index } 3)).$$

These do not require the stack organization. In effect, appropriate assignment allows a general register machine to be used as a stack machine for most cases of expression evaluation. This has the advantage of providing temporary, random access to common subexpressions, a capability that is usually hard to exploit in stack architectures.

THE EVOLUTION OF THE PMS (MODULAR) STRUCTURE

The end product of the PDP-11 design is the computer itse'f, and in the evolution of the architecture one can see images of the evolution of ideas. In this section, the architectural evolution is outlined, with a special emphasis on the Unibus.

The Unibus is the architectural component that connects together all of the other major components. It is the vehicle over which data flow between pairs of components takes place. Its structure is described in Chapter 11.

In general, the Unibus has met all expectations. Several hundred types of memories and peripherals have been interfaced to it; it has become a standard architectural component of systems in the $3K to $100K price range (1975). The Unibus does limit the performance of the fastest machines and penalizes the lower performance machines with a higher cost. Recently it has become clear that the Unibus is adequate for large, high performance systems when a cache structure is used because the cache reduces the traffic between primary memory and the central processor since about one-tenth of the memory references are outside the cache. For still larger systems, supplementary buses were added for central processor to primary memory and primary memory to secondary nemory traffic. For very small systems like the LSI-11, a narrower bus was designed.

The Unibus, as a standard, has provided an architectural component for easily configuring

40

systems. Any company, not just DEC, can easily build components that interface to the bus. Good buses make good engineering neighbors, since people can concentrate on structured design. Indeed, the Unibus has created a secondary industry providing alternative sources of supply for memories and peripherals. With the exception of the IBM 360 Multiplexer/Selector Bus, the Unibus is the most widely used computer interconnection standard.

The Unibus has also turned out to be invaluable as an "umbilical cord" for factory diagnostic and checkout procedures. Although such a capability was not part of the original design, the Unibus is almost capable of controlling the system components (e.g., processor and memory) during factory checkout. Ideally, the scheme would let all registers be accessed during full operation. This is possible for all devices except the processor. By having all central processor registers available for reading and writing in the same way that they are available from the console switches, a second system can fully monitor the computer under test.

In most recent PDP-11 models, a serial communications line, called the ASCII Console, is connected to the console, so that a program may remotely examine or change any information that a human operator could examine or change from the front panel, even when the system is not running. In this way computers can be diagnosed from a remote site.

Difficulties with the Design

The Unibus design is not without problems. Although two of the bus bits were set aside in the original design as parity bits, they have not been widely used as such. Memory parity was implemented directly in the memory; this phenomenon is a good example of the sorts of problems encountered in engineering optimization. The trading of bus parity for memory parity exchanged higher hardware cost and decreased performance for decreased service cost and better data integrity. Because engineers are usually judged on how well they achieve production cost goals, parity transmission is an obvious choice to pare from a design, since it increases the cost and decreases the performance. As logic costs decrease and pressure to include warranty costs as part of the product design cost increases, the decision to transmit parity may be reconsidered.

Early attempts to build tightly coupled multiprocessor or multicomputer structures (by mapping the address space of one Unibus onto the memory of another), called Unibus windows, were beset with a logic deadlock problem. The Unibus design does not allow more than one master at a time. Successful multiprocessors required much more sophisticated sharing mechanisms such as shared primary memory.

Unibus Cost and Performance

Although performance is always a design goal, so is low cost; the two goals conflict directly. The Unibus has turned out to be nearly optimum over a wide range of products. It served as an adequate memory-processor interconnect for six of the ten models. However, in the smallest system, DEC introduced the LSI-11 Bus, which uses about half the number of conductors. For the largest systems, a separate 32-bit data path is used between processor and memory, although the Unibus is still used for communication with the majority of the I/O controllers (the slower ones). Figure 1 summarizes the evolution of memory-processor interconnections in the LSI-11 Family. Levy (Chapter 11) discusses the evolution in more detail.

The bandwidth of the Unibus is approximately 1.7 megabytes per second or 850 K transfers/second. Only for the largest configurations, using many I/O devices with very high data rates, is this capacity exceeded. For most configurations, the demand put on an I/O bus is limited by the rotational delay and head positioning of disks and the rate at which programs (user and system) issue I/O requests.

An experiment to further the understanding of Unibus capacity and the demand placed against it was carried out. The experiment used a synthetic workload; like all synthetic workloads, it can be challenged as not being representative. However, it was generally agreed that it was a heavy I/O load. The load simulated transaction processing, swapping, and background computing in the configuration shown in Figure 3. The load was run on five systems, each placing a different demand on the Unibus.

Each run produced two numbers: (1) the time to complete 2,000 transactions, and (2) the number of iterations of a program called HANOI that were completed.

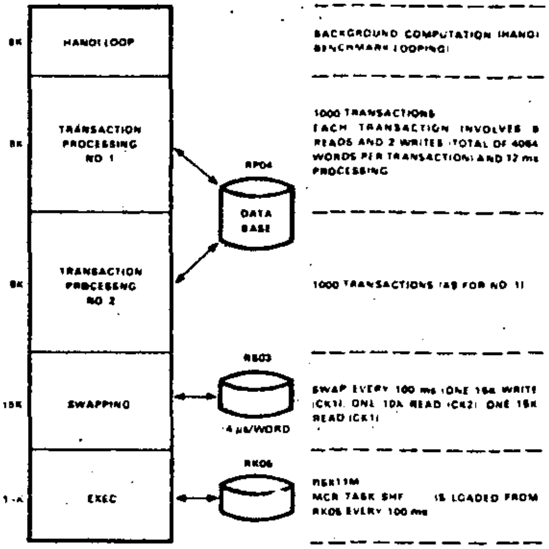

8K
HANOI LOOP
8K
TRANSACTION PROCESSING NO 1
8K
TRANSACTION PROCESSING NO 2
15K
SWAPPING
1-K
EXEC
RP04
DATA BASE
RS03
4 µs/WORD
RK06
BACKGROUND COMPUTATION (HANOI BENCHMARK LOOPING)
1000 TRANSACTIONS
EACH TRANSACTION INVOLVES 8 READS AND 2 WRITES (TOTAL OF 4064 WORDS PER TRANSACTION) AND 12 ms PROCESSING
1000 TRANSACTIONS (AS FOR NO 1)
SWAP EVERY 100 ms (ONE 16K WRITE (CK1), ONE 10K READ (CK2), ONE 15K READ (CK1)
RSX11M
MCR TASK SHF IS LOADED FROM RK06 EVERY 100 ms

Figure 3. The synthetic workload used to measure Unibus capacity.

System	Benchmark Time (minutes)*	Number of HANOI Iterations
11/60 cache on	15	12
11/60 cache off	15	2
11/40	15	3
11/70 MBCBUS	15	23
11/70 Unibus	26	38

*2,000 transactions plus swapping plus HANOI.

The results were interpreted as follows:

1. **I/O throughput.** For this workload the Unibus bandwidth was adequate. For systems 1 through 4 the I/O activity took the same amount of time.
2. **11/70 Unibus.** The run on this system (no use was made of the 32-bit wide processor/memory bus) took longer because of the retries caused by data lates (approximately 19,000) on the moving head disk (RP04). The extra time taken for the benchmark allowed more iterations of HANOI to occur. The PDP-11/70 Unibus had a bandwidth of about 1 megabyte. It was less than the usual Unibus (about 1.7 megabyte) because of the map delay (100 nanoseconds), the cache cycle (240 nanoseconds), and the main memory bus redriving and synchronization.
3. **11/60 Cache.** Systems 1 and 2 clearly show the effectiveness of a cache. Most memory references for HANOI were to the cache and did not involve the Unibus, which was the PDP-11/60s I/O Bus. Systems 2 and 3 were essentially equivalent, as expected. There are two reasons for the 11/40 having slightly more compute bandwidth than an 11/60 with its cache off. First, the 11/40 memory is faster than the 11/60 backing store, and second, the 11/40 processor relinquishes the Unibus for a direct memory access cycle; the 11/60 processor must request the Unibus for a processor cycle.

There are several attributes of a bus that affect its cost and performance. One factor affecting performance is simply the data rate of a single conductor. There is a direct tradeoff involving cost, performance, and reliability. Shannon [1948] gives a relationship between the fundamental signal bandwidth of a link and the error rate (signal-to-noise ratio) and data rate. The performance and cost of a bus are also affected by its length. Longer cables cost proportionately more, since they require more complex circuitry to drive the bus.

Since a single-conductor link has a fixed data rate, the number of conductors affects the net speed of a bus. However, the cost of a bus is directly proportional to the number of conductors. For a given number of wires, time domain multiplexing and data encoding can be used to trade performance and logic complexity. Since logic technology is advancing faster than wiring technology, it seems likely that fewer conductors will be used in all future systems, except where the performance penalty of time domain multiplexing is unacceptably great.

If, during the original design of the Unibus, DEC designers could have foreseen the wide range of applications to which it would be applied, its design would have been different. Individual controllers might have been reduced in complexity by more central control. For the largest and smallest systems, it would have been useful to have a bus that could be contracted or expanded by multiplexing or expanding the number of conductors.

The cost-effectiveness of the Unibus is due in large part to the high correlation between memory size, number of address bits, I/O traffic, and processor speed. Gene Amdahl's rule of thumb for IBM computers is that 1 byte of memory and 1 byte/sec of I/O are required for each instruction/sec. For traditional DEC applications, with emphasis in the scientific and control applications, there is more computation required per memory word. Further, the PDP-11 instruction sets do not contain the extensive commercial instructions (character strings) typical of IBM computers, so a larger number of instructions must be executed to accomplish the same task. Hence, for DEC computers, it is better to assume 1 byte of memory for each 2 instructions/sec, and that 1 byte/sec of I/O occurs for each instruction/sec.

In the PDP-11, an average instruction accesses 3–5 bytes of memory, so assuming 1 byte of I/O for each instruction/sec, there are 4–6 bytes of memory accessed on the average for each instruction/sec. Therefore, a bus that can support 2 megabytes/sec of traffic permits instruction execution rates of 0.33–0.5 mega-instructions/sec. This implies memory sizes of 0.16–0.25 megabytes, which matches well with the maximum allowable memory of 0.064–0.256 megabytes. By using a cache memory on the processor, the effective memory processor rate can be increased to balance the system further. If fast floating-point instructions were added to the instruction set, the balance might approach that used by IBM and thereby require more memory (an effect seen in the PDP-11/70).

The task of I/O is to provide for the transf of data from peripheral to primary memory where it can be operated on by a program in a processor. The peripherals are generally slow, inherently asynchronous, and more error-prone than the processors to which they are attached.

Historically, I/O transfer mechanisms have evolved through the following four stages:

1. **Direct sequential I/O under central processor control.** An instruction in the processor causes a data transfer to take place with a device. The processor does not resume operation until the transfer is complete. Typically, the device control may share the logic of the processor: The first input/output transfer (IOT) instruction in the PDP-1 is an example: the IOT effects transfer between the Accumulator and a selected device. Direct I/O simplifies programming because every operation is sequential.

2. **Fixed buffer, 1-instruction controllers.** An instruction in the central processor causes a data transfer (of a word or vector), but in this case, it is to a buffer of the simple controller and thus at a speed matching that of the processor. After the high speed transfer has occurred, the processor continues while an asynchronous, slower transfer occurs between the buffer and the device. Communication back to the processor is via the program interrupt mechanism. A single instruction to a simple controller can also cause a complete block (vector) of data to be transmitted between memory and the peripheral. In this case, the transfer takes place via the direct memory access (DMA) link.
3. **Separate I/O processors – the channel.** An independent I/O processor with a unique ISP controls the flow of data between primary memory and the peripheral. The structure is that of the multiprocessor, and the I/O control program for the device is held in primary memory. The central processor informs the I/O processor about the I/O program location.
4. **I/O computer.** This mechanism is also asynchronous with the central processor, but the I/O computer has a private memory which holds the I/O program. Recently, DEC communications options have been built with embedded control programs. The first example of an I/O computer was in the CDC 6600 (1964).

The authors believe that the single-instruction controller is superior to the I/O processor as embodied in the IBM Channel mainly because the latter concept has not gone far enough. Channels are costly to implement, sufficiently complex to require their own programming environment, and yet not quite powerful enough to assume the processing, such as file management, that one would like to offload from the processor. Although the I/O traffic does require central processor resources, the addition of a second, general purpose central processor is more cost-effective than using a central processor-I/O processor or central processor-multiple I/O processor structure. Future I/O systems will be message-oriented, and the various I/O control functions (including diagnostics and file management) will migrate to the subsystem. When the I/O computer is an exact duplicate of the central processor, not only is there an economy from the reduced number of part types but also the same programming environment can be used for I/O software development and main program development. Notice that the I/O computer must implement precisely the same set of functions as the processor doing direct I/O.*

MULTIPROCESSORS

It is not surprising that multiprocessors are used only in highly specialized applications such as those requiring high reliability or high availability. One way to extend the range of a family and also provide more performance alternatives with fewer basic components is to build multiprocessors. In this section some factors affecting the design and implementation of multiprocessors, and their effect on the PDP-11, are examined.

It is the nature of engineering to be conservative. Given that there are already a number of risks involved in bringing a product to the market, it is not clear why one should build a higher risk structure that may require a new way of programming. What has resulted is a sort of deadlock situation: people cannot learn how to program multiprocessors and employ them in a

*The I/O computer is yet another example of the wheel of reincarnation of display processors (see Chapter 7).

single task until such machines exist, but manufacturers will not build the machine until they are sure that there will be a demand for it, i.e., that the programs will be ready.

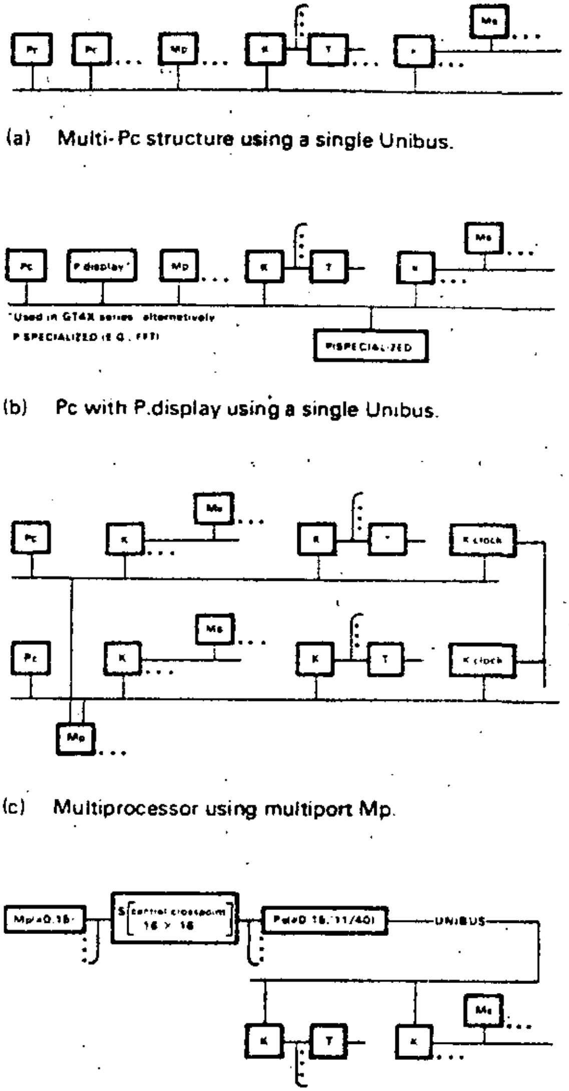
There is little or no market for multiprocessors even though there is a need for increased reliability and availability of machines. IBM has not promoted multiprocessors in the marketplace, and hence the market has lagged.

One reason that there is so little demand for multiprocessors is the widespread acceptance of the philosophy that a better single-processor system can always be built. This approach achieves performance at the considerable expense of spare parts, training, reliability, and flexibility. Although a multiprocessor architecture provides a measure of reliability, backup, and system tunability unreachable on a conventional system, the biggest and fastest machines are uniprocessors – except in the case of the Bell Laboratories Safeguard Computer [Bell Laboratories, 1975].

Multiprocessor systems have been built out of PDP-11s. Figure 4 summarizes the design and performance of some of these machines. The topmost structure was built using 11/05 processors, but because of inadequate arbitration techniques in the processor, the expected performance did not materialize. Table 3 shows the expected results for multiple 11/05 processors sharing a single Unibus and compares them with the PDP-11/40.

From the results of Table 3 one would expect to use as many as three 11/05 processors to achieve the performance of a model 11/40. More than three processors will increase the performance at the expense of the cost-effectiveness. This basic structure has been applied on a production basis in the GT40 series of graphics processors for the PDP-11. In this scheme, a second display processor is added to the Unibus for display picture maintenance. A similar structure is used for connecting special signal-processing computers to the Unibus although these structures are technically coupled computers rather than multiprocessors.

As an independent check on the validity of this approach, a multiprocessor system has


Pc
Mp
K
T
Ms
P display*
*Used in GT4X series alternatively
P SPECIALIZED (E.G., FFT)
P(SPECIALIZED
K clock
Mp(#0:15)
S[central crosspoint 16 × 16]
Pc(#0:15; 11/40)
UNIBUS

(a) Multi-Pc structure using a single Unibus.

(b) Pc with P.display using a single Unibus.

(c) Multiprocessor using multiport Mp.

(d) C.mmp CMU multi-miniprocessor computer structure.

Figure 4. PDP-11 multiprocessor PMS structures.

Table 3. Multiple PDP-11/05 Processors Sharing a Single Unibus

Number and Processor Model	Processor Performance (Relative)	Processor Price	Price*/Performance	System Price	Price†/Performance
1–11/05	1.00	1.00	1.00	3.00	1.00
2–11/05	1.85	1.23	0.66	3.23	0.58
3–11/05	2.4	1.47	0.61	3.47	0.48
1–11/40	2.25	1.35	0.60	3.35	0.49

*Processor cost only.

†Total system cost assuming one-third of system is processor cost.

been built, based on the Lockheed SUE [Ornstein *et al.*, 1972]. This machine, used as a high speed communications processor, is a hybrid design: it has seven dual-processor computers with each pair sharing a common bus as outlined above. The seven pairs share two multiport memories.

The second type of structure given in Figure 4 is a conventional, tightly coupled multiprocessor using multiple-port memories. A number of these systems have been installed, and they operate quite effectively. However, they have only been used for specialized applications because there has been no operating system support for the structure.

PDP-11 Based Multiprocessor: Carnegie-Mellon University Research Computers

The PDP-11 architecture has been employed to pioneer new ideas in the area of multiprocessors. The three multiprocessors built at Carnegie-Mellon University (CMU) are discussed: C.mmp [Wulf and Bell, 1972], a 16-processor multiprocessor: C.vmp [Siewiorek *et al.*, 1976], a triplicated, voting multiprocessor computer for high reliability; and Cm* (Chapter 20), a set of computer modules based on LSI-11.

The three CMU multiprocessors are good examples of multiprocessor development directions because it is quite likely that technology will force the evolution of computing structures to converge into three styles of multiprocessor computers: (1) C.mmp style, for high performance, incremental performance, and availability (maintainability); (2) C.vmp style for very high availability motivated by increasing maintenance costs, and (3) loosely coupled computers like Cm* to handle specialized processing, e.g., front end, file, and signal processing. This argument is based on history, present technology, and resulting price extrapolations:

1. MOS technology appears to be increasing in both speed and density faster than the technology (such as ECL) from which high performance machines are usually built.
2. Standards in the semiconductor industry tend to form more quickly for high volume products. For example, in the 8-bit microcomputer market, one type supplies about 50 percent of the market and three types supply over 90 percent.
3. The price per chip of the single MOS chip processors decreases at a substantially greater rate than for the low volume, high performance special designs. Chips in both designs have high design costs, but the single-MOS-chip processors have a much higher volume.

4. Several 16-bit processor-on-a-chip processors, with an address space matching and appropriate data-types matching the performance, exist in 1978. Such a commodity can form the basis for nearly all future computer designs.
5. The performance (instructions per second) per chip, which is already greater for MOS processor chips than for any other kind, is improving more rapidly than for large scale computers. This will pull usage more rapidly into large arrays of processors because of the essentially "free cost" of processors (especially relative to large, low volume custom-built machines).

Therefore, most subsequent computers will be based on standard, high volume parts. For high performance machines, since processing power is available at essentially zero cost from processor-on-a-chip-based processors, large scale computing will come from arrays of processors, just as memory subsystems are built from arrays of 64 Kbit integrated circuits.

The multiprocessor research projects at CMU have emphasized synthesis and measurement. Operating systems have been built for them, and the executions of user programs have been carefully analyzed. All the multiprocessor interferences, overheads, and synchronization problems have been faced for several applications; the resultant performance helps to put their actual costs in perspective. Figure 5 shows the HARPY speech recognition program and compares the performance of C.mmp and Cm* with three DEC uniprocessors (PDP-10 with KA10 processor, PDP-10 with KL10 processor, and PDP-11/40).

C.mmp

C.mmp (Figure 6) a 16 processor (11/40s and 11/20s) system has 2.5 million words of shared primary memory. It was built to investigate the programming (and resulting performance) questions associated with having a large number of processors. Since the time that the first paper [Wulf and Bell, 1972] was written, C.mmp has been the object of some interestin studies, the results of which are summarized below.

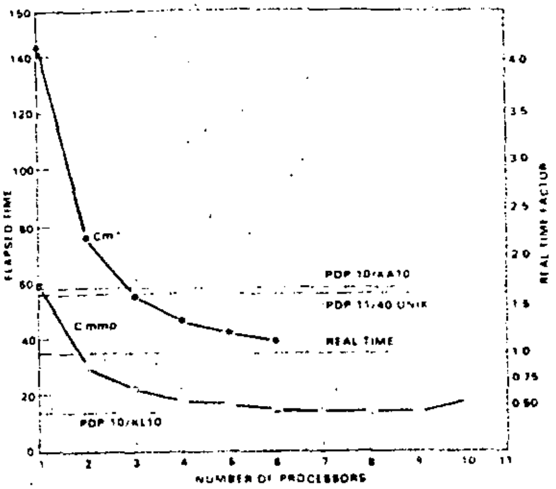
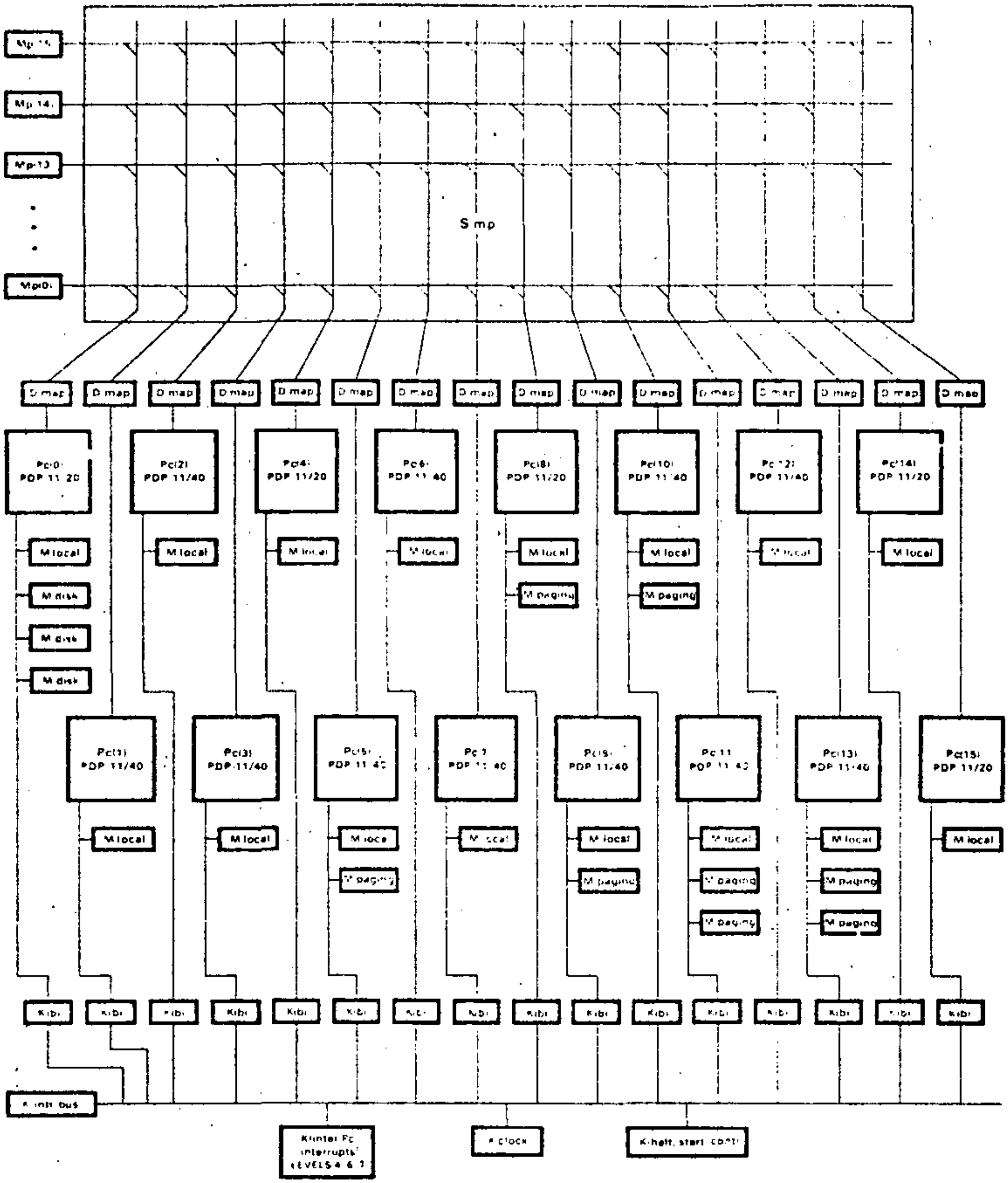


Figure 5 A performance comparison of two multiprocessors, C.mmp and Cm*, with three uniprocessors at Carnegie-Mellon University. The application used is HARPY, a speech recognition program. This graph is based on work done by Peter Oleinick [1978] and Peter Feiler at CMU.

C.mmp was motivated by the need for more computing power to solve speech recognition and signal processing problems and to understand the multiprocessor software problem. Until C.mmp, only one large, tightly coupled multiprocessor had been built – the Bell Laboratories Safeguard Computer [Bell Laboratories, 1975].

The original paper [Wulf and Bell, 1972] describes the economic and technical factors influencing multiprocessor feasibility and argues for the timeliness of the research. Various problems to be researched and a discussion of particular design aspects are given. For example, since C.mmp is predicated on a common operating systems, there are two sources of degradation: memory contention and lock contention.



NOTE
Kibi stands for K(inter-bus interface)

Figure 6 A PMS diagram of C.mmp (from [Oleinick,1978]).

The machine's theoretical performance as a function of memory-processor interference is based on Strecker's [1970] work. In practice, because the memory was not built with low-order address interleaving, memory interference was greater than expected. This problem was solved by having several copies of the program segments.

As the number of memory modules and processors becomes very large, the theoretical performance (as measured by the number of accesses to the memory by the processors) approaches half the memory bandwidth (i.e., the number of memory modules memory cycle time) [Baskett and Smith, 1976]. Thus, with infinite processors, there is no maximum limit on performance, provided all processors are not contending for the same memory.

Although there is a discussion in the original paper outlining the design direction of the operating system, HYDRA, later descriptions should be read [Wulf *et al.*, 1975]. Since the small address of the PDP-11 necessitated frequent map changes, PDP-11/40s with writable control stores were used to implement the operating systems calls which change the segment base registers.

There are three basic approaches to the effective application of multiprocessors:

1. System level workload decomposition. If a workload contains a lot of inherently independent activities, e.g., compilation, editing, file processing, and numerical computation, it will naturally decompose.
2. Program decomposition by a programmer. Intimate knowledge of the application is required for this time-consuming approach.
3. Program decomposition by the compiler. This is the ideal approach. However, results to date have not been especially noteworthy.

C.mmp was predicated on the first two approaches. ALGOL 68, a language with facilities for expressing parallelism in programs, has since been implemented. It has assisted greatly with program decomposition and looks like a promising general approach. It is imperative, however, to extend the standard languages to handle vectors and arrays.

The contention for shared resources in a multiprocessor system occurs at several levels. At the lowest level, processors contend at the cross-point switch level for memory. On a higher level there is contention for shared data in the operating system kernel; processes contend for I/O devices and for software processes, e.g., for memory management. At the user level shared data implies further contention. Table 4 points to models on experimental data at these different levels.

Marathe's data show that the shared data of HYDRA is organized into enough separate objects so that a very small degradation (less than 1 percent) results from contention for these objects. He also built a queueing model which projected that the contention level would be about 5 percent in a 48 processor system.

Oleinick [1978] has used C.mmp to conduct an experimental, as opposed to theoretical, study of the implementation of parallel algorithms on a multiprocessor. He studied the operation of Rootfinder, a program that is an

Table 4. References for Experimental Data on Contention at Each of Three Levels in the C.mmp System

Contention Level	Reference
User-program	Oleinick [1978] Fuller and Oleinick [1976]
HYDRA kernel objects	Marathe and Fuller [1977]
Cross-point switch	Baskett and Smith [1976] Fuller [1976] Strecker [1970] Wulf and Bell [1972]

extension of the bisection method for finding the roots of an equation.

A natural decomposition of the binary search for a root into n parallel processes is to evaluate the function simultaneously at n points. Under ideal conditions, all processes would finish the function evaluation (required at each step) at the same time, and then some brief bookkeeping would take place to determine the next subinterval for the n processes to work on. However, because the time to evaluate the function is data dependent, some processes are completed before others. Moreover, if the bookkeeping task is time consuming relative to the time to evaluate the function, the speedup ratio will suffer. Oleinick systematically studied each source of fluctuation in performance and found the dominant one to be the mechanism used for process synchronization.

Four different locks for process synchronization, called: (1) spin lock, (2) kernel semaphore, (3) PM0, and (4) PM1, are available to the C.mmp user. The spin lock, the most rudimentary, does not cause an entry to the HYDRA operating system. It is a short sequence of instructions which continually test a semaphore until it can be set successfully. The process of testing for the availability of a resource, and seizing the resource if available, could be called TEST-AND-LOCK. When the resource is no longer needed, it is released by an UNLOCK process. These two processes are called the *P* operation and the *V* operation respectively, as originally named by Edgar Dijkstra. The *P* and *V* operations in the C.mmp spin lock are in fact the following PDP-11 code sequences:

```
P:  CMP SEMAPHORE,
    #1                  ;SEMAPHORE=1?
    BNE P               ;loop until it is 1
    DEC SEMAPHORE       ;Decrement SEMAPHORE
    BNE P               ;If not equal 0 go to P
V:  MOV #1, SEMAPHORE   ;Reset SEMAPHORE to 1
```

Although this repeating polling is extremely fast, it has two major drawbacks: first, the processor is not free to do useful work; second, the polling process consumes memory cycles of the memory bank that contains the semaphore.

The kernel semaphore, implemented in HYDRA, is the low level synchronization mechanism intended to be used by system processes. When a process blocks or wakes up, a state change for that process is made inside the kernel of HYDRA. If a process blocks (fails to obtain a needed resource) while trying to P (test and lock) a semaphore, the kernel swaps the process from the processor, and the pages belonging to that process are kept in primary memory. The other semaphore mechanisms (PM0 and PM1) take proportionately more time (>1 millisecond).

C.vmp

C.vmp, is a triplicated, voting multiprocessor designed to understand the difficulty (or ease) of using standard, off-the-shelf LSI-11s to provide greatly increased reliability. There is concern for increased reliability because systems are becoming more complex, are used for more critical applications, and because maintenance costs for all systems are increasing. Because the designers themselves carry out and analyze the work, this section provides first-hand insight into high reliability designs and the design process – especially its evaluation.

Several design goals were set and the work has been carried out. The C.vmp system has operated since late 1977, when the first phase of work was completed.

The goal of software and hardware transparency turned out to be easier to attain than expected, because of an idiosyncrasy of the floppy disk controller. Because the controller effects a word-at-a-time bus transfer from a one-sector buffer, voting can be carried out at a very low level. It is unclear how the system would have been designed without this type of controller; at a minimum, some part of the software transparency goal would not have been met, and a significant controller modification would have been necessary.

A number of models are given by which the design is evaluated. From the discussion of component reliabilities the reader should get some insight into the factors contributing to reliability. It should be noted that a custom-designed LSI voter is needed to get a sufficiently low cost for a marketable C.vmp. While the intent of C.vmp development was not a product, it does provide much of the insight for such a product.

Cm*

Cm* is described in Chapter 20; however, because it is one of the three CMU machines pointing to future technology-driven trends in multiprocessor use of LSI-11 architecture, it is given some mention here. The Cm* work, sponsored by the National Science Foundation (NSF) and the Advanced Research Projects Agency (ARPA), is an extension of earlier NSF-sponsored research [Bell *et al.*, 1973] on register transfer level modules. As large-scale integration and very large-scale integration enable construction of the processor-on-a-chip, it is apparent that low level register transfer modules are obsolete for the construction of all but low volume computers. Although the research is predicated on structures employing a hundred or so processors, Chapter 20 describes the culmination of the first (10-processor) phase.

In Chapter 20 the authors base their work on diseconomy-of-scale arguments. To provide additional context for their research, computer modules (Cm*), multiprocessors (C.mmp), and computer networks are described in terms of performance and problem suitability. They give a description of the modules structure, together with its associated limitations and potential research problems.

The grouping of processor and memory into modules and the hierarchy of bus structures – LSI-11 Bus, Map Bus, and Intercluster bus, radical departures from conventional computer systems – is given. The final, most important part of the chapter evaluates the performance of Cm* for five different problems.

Since the time that Chapter 20 was written, construction of a 50 computer modules Cm* has begun and will be operational by the end of 1978 for evaluation in 1979. The extension of Cm* is known as Cm*/50 and is shown in Figure 7. It will be used to test parallel processing methods, fault tolerance, modularity, and the extensibility of the Cm* structure.

The PDP-11/70mP Experimental Multiprocessor Computer

The PDP-11/70mP aims to extend the reliability, availability, maintainability and performance range of the PDP-11 Family. It uses 11/70 processor hardware and the RSX-11M software as basic building blocks.

The systems can have up to four processors which have access to common central memories as shown in Figure 8. Each MOS primary memory contains 256 Kbyte to 1 Mbyte and a port (switch) by which up to four processors may access it. A failed memory may be isolated for repair. Usually two processors share (have access to) each of the I/O devices through a Unibus switch or dual ported disk memories.

Failure of a high speed mass storage bus controller, a processor, or one port of a device will not preclude use of that device through the other port. These devices can also be isolated from their respective buses so that failure of a device will not preclude access to other devices.

Each of the processor units has a write-through cache memory. Through normal system operation, data within these local caches may become inconsistent with data elsewhere in the system. To eliminate this problem, the operating system and the hardware components have been modified. The RSX-11M system either clears the cache of inconsistent data or avoids using the cache for specific situations.

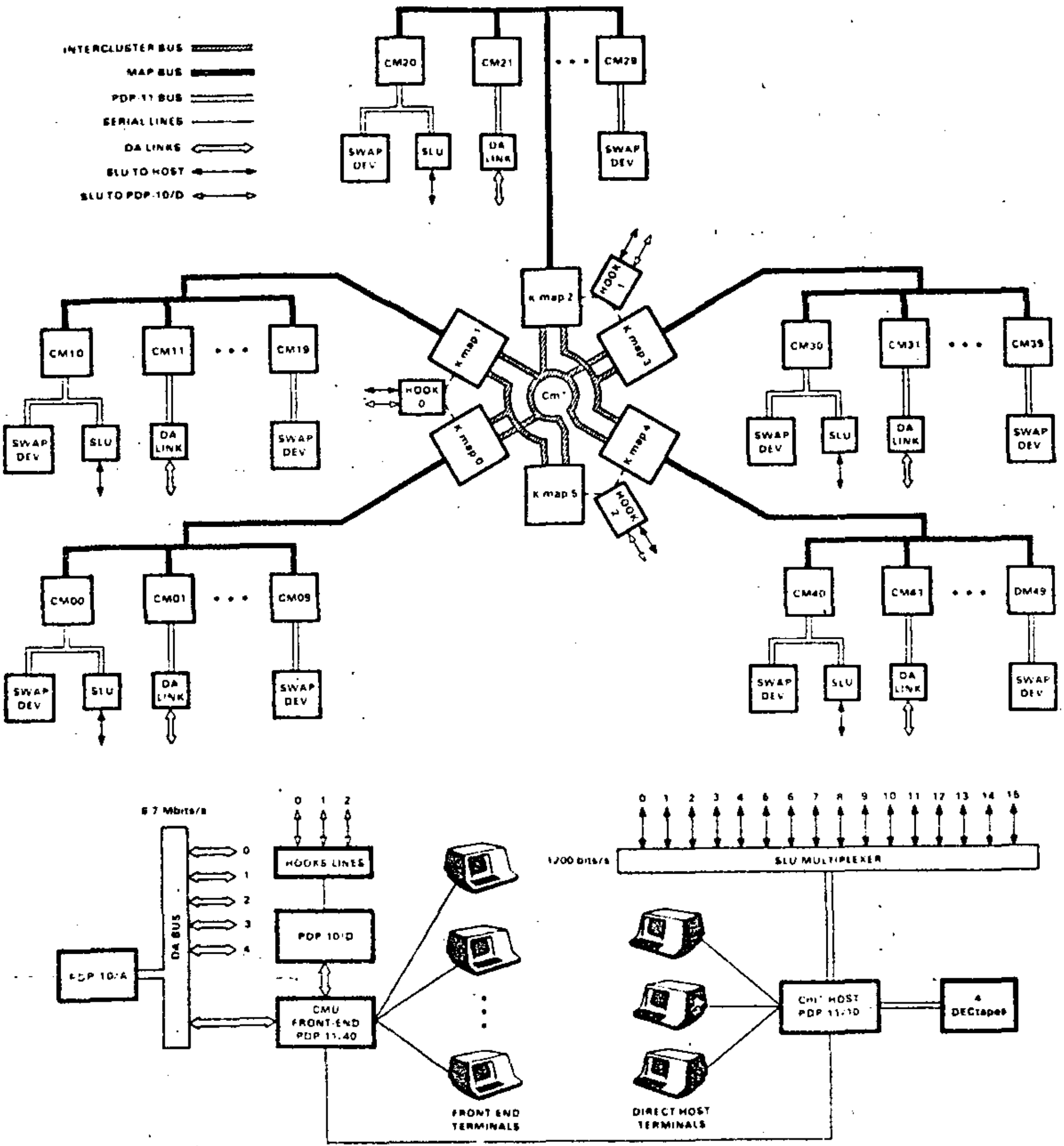


Figure 7 Details of the Cm*/50 system.

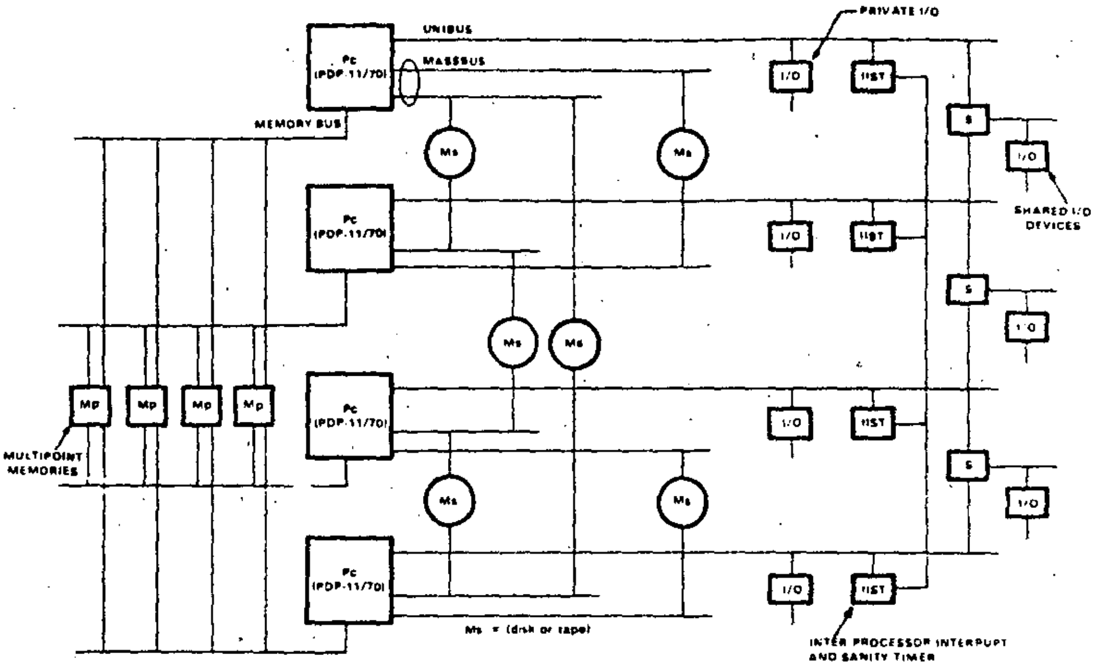


Figure 8. Four-processor multiprocessor based on PDP-11/70 processors.

The software to manipulate the cache is contained in the executive and is transparent to user programs.

An Interprocessor Interrupt and Sanity Timer (IIST) provides the executive software with a mechanism to interrupt processors for rescheduling. The IIST includes a timer for each processor which is periodically refreshed by software after execution of diagnostic check routines. If the refresh commands do not occur within a prescribed interval, the IIST will issue an interprocessor interrupt to inform the other processors of faulty operation. The IIST also contains a mechanism for initially loading the multiprocessor system.

The system design results in an extension to the PDP-11 that is transparent to user programs and yields increases in performance over a single processor 11/70 system. This performance increase is due to the symmetry, such that nearly any resource can be accessed by any process with minimum overhead. Also, unlike multiple computer systems that communicate via high speed links, the large primary memory can be combined and used by a single process. Moreover, dynamic assignment of processes to specific computer systems (Figure 9) can be made.

The system has been designed to increase the availability by reducing the impact of failures on system performance through the use of multiple redundant components. In this way, failed elements can be isolated for repair. The design is such that the system may be easily reconfigured so that system operation can be resumed and the failed component repaired off-line.

Extensions to the diagnostic software and hardware error detection mechanisms facilitate quick location of faults. User-mode diagnostics are run concurrently with the application software: this permits maintenance of the disk and tape units to be done on-line.

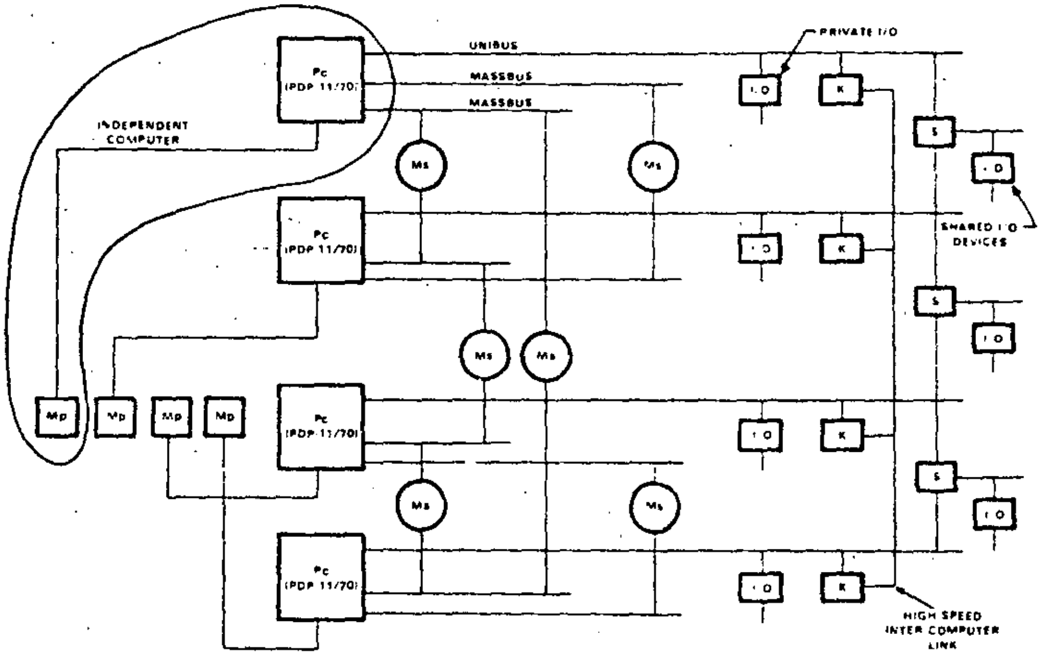


Figure 9. Four-processor multicomputer system based on PDP-11/70 processors.

Now that the 11/70mP has implemented its IIST and defined an architectural extension for multiprocessing, another roadblock to the use of multiprocessors has been passed; namely, an extension for interprocessor signaling has been defined. This might have been defined much earlier in the life of the PDP-11. In the IBM computers the SIGP instruction was not available on 360s until the 370 extensions.

PULSAR: A Performance Range mP System

PULSAR is a 16 LSI-11 multiprocessor computer for investigating the cost-effectiveness of multiple microprocessors. It covers a performance range of approximately a single LSI-11 to better than a PDP-11/70 for simple instructions.

The breadboard system (Figure 10) is based on the PDP-11/70 processor-memory-switch structure, including multiple interrupt levels and 22-bit physical addressing. However, it does not implement instruction (I) and data (D) space or Supervisor mode, and it lacks the Floating-Point Processors.

The processors (P-Boards) communicate with each other, the Unibus Interface (UBI), and a Common Cache and Control via a high-bandwidth, synchronous bus.

The Common Cache and Control contains a large (8 Kword), direct-mapping, shared cache with a 2-word block size, interfacing to the 2- or 4-way interleaved 11/70 Memory Bus. This prevents the memory subsystem from becoming a bottleneck, in spite of the large reduction in bandwidth demand provided by the cache. The control provides all the mapping functions for both Unibus and processor accesses to memory. The Unibus map registers and the process map registers for each processor are held in a single bipolar memory.

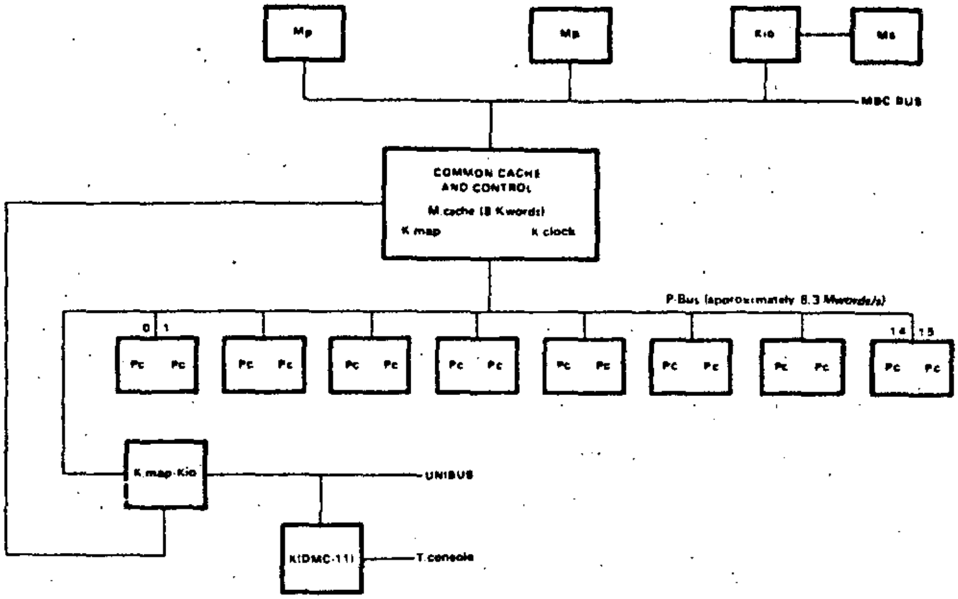


Figure 10. PMS diagram of the breadboard version of the DEC PULSAR.

The Unibus Interface provides the Unibus control functions of a conventional PDP-11. Interrupts are fielded by the first enabled processor with preferential treatment for any processor in WAIT state.

Each processor board contains two independent microprocessor chip sets with modified microcode. Internal contention for the adapter is eliminated by running the two processors out of phase with each other. Such contention as does exist is resolved by the mechanism for arbitration of the processor bus itself. The PULSAR has a serial line (ASCII) console interfacing via a microcode driven communications controller, equipped with modified microcode. In addition, a debugging panel has displays for every stage of the processor bus and controller pipeline.

Console operations are effected by the Unibus Interface interrogating or changing a save area for each processor, physically held in the mapping array, in response to ASCII console messages over the Unibus. Each processor places all appropriate status in the save area on every HALT, and restores from the save area prior to acting upon every CONTINUE or START.

The PULSAR system is pipeline oriented with specific time slots for each processor. This permits a single simple arbitration mechanism, rather than separate complex ones for each resource.

Once the pipeline is assigned to a transaction, the successive intervals of time are assigned to the following resources in order:

1. The mapping array.
2. The address translation logic.
3. The cache.
4. The address validation logic.
5. The data lines of the P-Bus.

The memory subsystem, which is not a part of this resource pipeline, has an independent arbitration mechanism. Interfacing between these independent mechanisms is by means of queues.

There are some operations that require more than one access to the same resource in the pipeline. These operations are effectively handled as two transactions. Examples of such operations are, memory writes and internal I/O page (memory-management register) accesses. A memory write may need a second access to the cache for update, while the Internal I/O Page may need another access to the map array.

There are other operations in which the timing does not permit the use of a particular resource in the specific interval that is allocated to that transaction. This happens, for instance, when a read operation results in a cache miss. The data is not available in time. In this case a second transaction takes place, initiated when backing store data becomes available.

Cost projections indicate that a multiprocessor will have an increase in parts count over each possible equivalent performance uniprocessor in the range. This will range from a 20 percent increase for a two-processor, multiprocessor system to 0 percent at the top of the range. The 20 percent premium can be reduced if no provision is made for expansibility over the entire range. Clearly, a separate single processor structure can be cost-effective (since this is the LSI-11). The premium is based on parts count only and excludes considerations of cost benefits due to production learning, common spares and manuals, lower engineering costs, etc.

A number of computer systems have been built based on multiple processors in systems ranging from independent computers (with no interconnection) through tightly coupled computer networks which communicate by passing messages, to multiprocessor computers with shared memory. Table 5 gives a comparison of the various computers. Although n independent computers is a highly reliable structure, it is hard to give an example where there is no interconnection among the computers. The standard computer network interconnected via standard communications links is not given.

It is interesting to compare the multiprocessor and the tightly coupled multicomputer configurations (Figure 8 and 9) where the configurations are drawn in exactly the same way and with the same peripherals. In this way, columns 2 and 6 of Table 5 can be more easily compared. The tradeoff between the two structures is between lower cost and potentially higher performance for the multiprocessor (unless tasks can be statically assigned to the various computers in the network) versus somewhat higher reliability, availability, and maintainability for the network computer (because there is more independence among software and hardware). Varying the degree of coupling in the processors through the amount of shared memory determines which structure will result. The cost and the resultant reliability differentials for the two systems are determined by the size and the reliability of the software.

TECHNOLOGY: COMPONENTS OF THE DESIGN

In Chapter 2, it was noted that computers are strongly influenced by the basic electronic technology of their components. The PDP-11 Family provides an extensive example of designing with improved technologies. Because design resources have been available to do concurrent implementations spanning a cost/performance range, PDP-11s offer a rich source of examples of the three different design styles: constant cost with increasing functionality, constant functionality with decreasing cost, and growth path.

Memory technology has had a much greater impact on PDP-11 evolution than logic technology. Except for the LSI-11, the one logic family (7400 series TTL) has dominated PDP-11 implementations since the beginning. Except for a small increase after the PDP-11/20, gate density has not improved markedly. Speed improvement has taken place in the Schottky

Table 5. Characteristics of Various PDP-11 Based Multiprocessor and Multicomputers

	C.mmp	11/70mP	Pulsar	Cm*	C.vmp	11/70mC	n Computers
Coupling	Multiprocessor	Multiprocessor	Multiprocessor	Tightly coupled network	Triple modular redundant voting computer	Tightly coupled computer network	Independent
Page/figure	395/6	400/8	402/10	399/7	Not shown	401/9	Not shown
Processor type	20, 40	70	LSI-11	LSI-11	LSI-11	70	70
Reliability,	Medium	High	Medium	Medium	Very high	High	High
Availability,	Medium	High	Low	Medium	Very high	High	High
Maintainability	Medium	High	Low	Low	Very high	High	High
Performance range (times base processor)	1 - 16	1 - 4	2 - 16	1 - 100	1	1 - 4	1 - 12
Advantages	All resources can operate on any task(s); large processes occupying all Mp can be run			Range	Very high R, A, M	Backup of tasks to alt. computer; fast inter-C transfers	Complete Independence
Disadvantage	Single switch		Single memory and peripherals	Static assignment of tasks	1 Pc performance	Static assignment of tasks to computers	

TTL, and a speed/power improvement has occurred in the low power Schottky (LS) series. Departures from medium-scale integrated transistor-transistor logic, in terms of gate density, have been few, but effective. Examples are the bit-slice in the PDP-11/34 Floating-Point Processor, the use of programmable logic arrays in the PDP-11/04 and PDP-11/34 control units, and the use of emitter-coupled logic in some clock circuitry.

Memory densities and costs have improved rapidly since 1969 and have thus had the most impact. Read-write memory chips have gone from 16 bits to 4,096 bits in density and read-only memories from 16 bits to the 8 or 16 Kbits widely available in 1978. Various semiconductor memory size availabilities are given in Chapter 2 using the model of semiconductor density doubling each year since 1962.

The memory technology of 1969 imposed several constraints. First, core memory was cost-effective for the primary (program) memory, but a clear trend toward semiconductor primary memory was visible. Second, since the largest high speed read-write memories available were just 16 words, the number of processor registers had to be kept small. Third, there were no large high speed read-only memories that would have permitted a microprogrammed approach to the processor design.

These constraints established four design attitudes toward the PDP-11's architecture. First, it should be asynchronous, and thereby capable of accepting different configurations of memory that operate at different speeds. Second, it should be expandable to take eventual advantage of a larger number of registers, both user registers for new data-types and internal registers for improved context switching, memory mapping, and protected multiprogramming. Third, it could be relatively complex, so that a microcode approach could eventually be used to advantage: new data-types could be added to the instruction set to increase performance, even though they might add complexity. Fourth, the Unibus width should be relatively large, to get as much performance as possible, since the amount of computation possible per memory cycle was relatively small.

As semiconductor memory of varying price and performance became available, it was used to trade cost for performance across a reasonably wide range of PDP-11 models. Different techniques were used on different models to provide the range. These techniques include: microprogramming for all models except the 11/20 to lower cost and enhance performance with more data-types (for example, faster floating point); use of faster program memories for brute-force speed improvements (e.g., 11/45 with MOS primary memory, 11/55 with bipolar primary memory, and the 11/60 with a large writable control store); use of caches (11/70, 11/60, and 11/34C); and expanded use of fast registers inside the processor (the 11/45 and above). The use of semiconductors versus cores for primary memory is a purely economic consideration, as discussed in Chapter 2.

Table 6 shows characteristics of each of the PDP-11 models along with the techniques used to span a cost and performance range. Snow and Siewiorek (Chapter 14) give a detailed comparison of the processors.

VAX-11

Enlarging the virtual address space of an architecture has far more implications than enlarging the physical address space. The simple device of relocating program-generated addresses can solve the latter problem. The physical address space, the amount of physical memory that can be addressed, has been increased in two steps in the PDP-11 Family (Table 2).

The virtual address space, or name space, is a much more fundamental part of an architecture. Such addresses are programmer generated: to name data objects, their aggregates (whether they be vectors, matrices, lists, or

Table 6. Characteristics of PDP-11 Models with Techniques Used to Span Cost and Performance Range

		Performance			Range-Spanning Techniques		
Model	First Shipment	Basic Instructions Per Second (relative to PDP-11/03)	Floating-Point Arithmetic (whetstone instructions per second)	Memory Range (Kbytes)	For High Performance	For Low Cost	Notable Attributes
11/03 (LSI-11)	6/75	1	26	8–56		8 bit wide datapath; LSI-11 Bus; tailored PLA control	LSI–4 chips; ODT; Floating-Point (FIS), CIS, WCS mid-life kickers
11/04	9/75	2.8	18	8–56		Standard package; ROM; PLA	Backplane compatible with 11/34 for field upgrade; built-in ASCII console; self-diagnosis
11/05	6/72	2.5	13	8–56		Microprogrammed; ROM	Minimal 11 (2 boards)
11/20	6/70	3.1	20	8–56			ISP; Unibus
11/34	3/76	3.5	204	16–256		Shared use of ALU; PLA; ROM; microprogrammed	Cost-performance balance; 11/34C mid-life kicker; Bit-slice FPP
11/34C	5/78	7.3	262	32–256			Classic use of cache
11/40	1/73	3.6	57	16–256	Variable cycle length	Microprogrammed	FIS extension
11/60	6/77	27	592	32–256	Fetch overlap; dual scratch-pads; TTL/S	Heavily microprogrammed	Integral floating-point; WCS for local storage; RAMP
11/45	6/72	Core: 13 MOS: 23 Bipolar: 41	~260 ~335 ~362	8–256	Instruction prefetch; dual scratchpads; Fastbus; autonomous FPP; TTL/S		Pc speed to match 300 ns bipolar, high speed minicomputer FPP; memory management
11/55	6/76	41	725	16–64 (0–192 core)	All bipolar memory		
11/70	3/75	36	671	64-2048	32-bit-wide DMA bus; large memory		Cache; multiple buses, RAMP, FP11-C mid-life kicker; remote diagnosis
70mP							Multiprocessor architectural extensions; on-line maintainability; performance; availability
		range: 41–1	range: 56–1	range: 256–1			

shared data segments) and instructions (subroutine addresses, for example). Names seen by an individual program are part of a larger name space – that managed by an operating system and its associated language translators and object-time systems. An operating system provides program sharing and protection among programs using the name space of the architecture.

As the PDP-11/70 design progressed, it was realized that for some large applications there would soon be a bad mismatch between the 64-Kbyte name space and 4-Mbyte memory space. Two trends could be clearly seen: (1) minicomputer users would be processing large arrays of data, particularly in FORTRAN programs (only 8,096 double precision floating-point numbers are needed to fill a 16-bit name space), and (2) applications programs were growing rapidly in size, particularly large COBOL programs. Moreover, anticipated memory price declines made the problem worse. The need for a 32-bit integer data-type was felt, but this was far less important than the need for 32-bit addressing of a name space.

Thus, in 1974, architectural work began on extending the virtual address space of the PDP-11. Several proposals were made. The principal goal was compatibility with the PDP-11. In the final proposed architecture each of the eight general registers was extended to 32 bits. The addressing modes (hence, address arithmetic) inherent in the PDP-11 allowed this to be a natural, easy extension.

The design of the structure to be placed on a 32-bit virtual address presented the most difficulty. The most PDP-11 compatible structure would view a 32-bit address as 2^{16} 16-bit PDP-11 segments, each having the substructure of the memory management architecture presently being used. This segmented address space, although PDP-11 compatible, was ill-suited to FORTRAN and most other languages, which expect a linear address space.

A severe design constraint was that existing PDP-11 subroutines must be callable from programs which ran in the Extended Address mode. The main problem areas were in establishing a protocol for communicating addresses (between programs between the operating systems and programs on the occurrence of interrupts). Saving state (the program counter and its extension) on the stack was straightforward. However, the accessing of linkage addresses on the stack after a subroutine call instruction or interrupt event was not straightforward. Complicated sequences were necessary to ensure that the correct number of bytes (representing a 32-bit or 16-bit address) were popped from the stack.

The solution was hampered by the fact that DEC customers programmed the PDP-11 at all levels – there was no clear user level, below which DEC had complete control, as is the case with the IBM System 360 or the PDP-10 using the TOPS-10 or TOPS-20 monitors.

The proposed architecture was the result of work by engineers, architects, operating system designers and compiler designers. Moreover, it was subjected to close scrutiny by a wider group of engineers and programmers. Much was learned about the consequences of strict PDP-11 compatibility, the notions of degree of compatibility, and the software costs which would be incurred by an extended PDP-11 architecture.

Fortunately, the project was discontinued. There were many reservations about its viability. It was felt that the PDP-11 compatibility constraint caused too much compromise. Any new architecture would require a large software investment; a quantum jump over the PDP-11 was needed to justify the effort.

In April 1975, work on a 32-bit architecture was started on VAX-11, with the goal of building a machine which was culturally compatible with PDP-11. The initial group, called VAXA, consisted of Gordon Bell; Peter Conklin, Dave Cutler, Bill Demmer, Tom Hastings, Richy Lary, Dave Rodgers, Steve Rothman, and Bill Strecker as the principal architect. As a result of the experience with the extended PDP-11 designs, it was decided to drop the constraint of the PDP-11 instruction format in designing the extended virtual address space, or Native mode, of the VAX-11 architecture. However, in order to run existing PDP-11 programs, VAX-11 includes PDP-11 Compatibility mode. This mode provides the basic PDP-11 instruction set without privileged instructions (as defined by the RSX-11M operating system) and floating-point instructions. Nor is the former memory management architecture (KT-11) preserved in this mode.

Preserving the existing PDP-11 instruction formats with VAX-11 would have required too high a price in dynamic bit efficiency. Whereas the PDP-11 has a high level of efficiency in this area, adding the new operation codes for the anticipated data-types, access modes, and different length addresses would have lowered the instruction stream bit efficiency. An operation code extension field would have been required. It was also felt that data stream bit efficiency could be improved. For example, measurements showed that 98 percent of all literals were 6 bits or less in length.

Besides the desire to add the data-types for string, 32- and 64-bit integers, and decimal arithmetic, there were many other extensions proposed. These included a common procedure CALL instruction, demand paging, true indexing, context-sensitive indexing, and more I/O addressing.

Along the way, some major perturbations to the PDP-11 style were considered and rejected, often because they violated the notion of compatibility with PDP-11. Typed data and descriptor addressing were rejected on the grounds of dynamic bit efficiency. Although system software costs may be lower with such architectures, it was not possible to quantify the gain convincingly. Also, such an architecture destroyed any compatibility, cultural or otherwise, with PDP-11.

The experience with PDP-11 (floating point, in particular) led the VAX designers to reject a soft-machine architecture, i.e., one with an instruction set (and highly microprogrammed implementations) for general purpose emulation. Their PDP-11 experience showed that embedding a data-type (once it is understood) in the architecture gives a higher performance gain than embedding the higher level language control constructs. There was also a general objection to soft machines: the problem of controlling a proliferation of instruction sets invented by many small software groups was felt to be unmanageable. Moreover, higher level instruction sets jeopardize the ability to communicate between programs that are written in different languages. This compatibility is a major goal of VAX.

A capabilities-based architecture was rejected because it was not fully understood and because there was no performance or reliability data available from the few experimental machines which had been built.

ACKNOWLEDGEMENTS

We gratefully acknowledge the suggestions of Roger Cady, Dick Clayton, and Bruce Delagi who were eminently qualified and intimately involved in the PDP-11's evolution.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

ELEMENTO DE UNA COMPUTADORA

1984

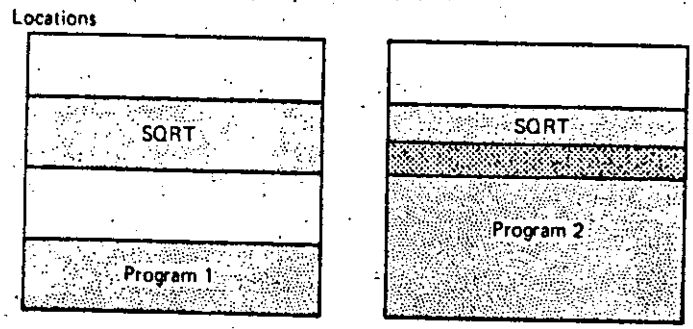


FIGURE 1.3 Example core allocation for absolute loading

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 number of students participating it is very difficult to personally grade every program run on the machine problems. So for the very simple problems – certainly the first problem which may be to count the number of A's in a register and leave the answer in another register – we have written a grading program that is included as part of the operating system. The grading program calls the student's program and transfers control to it. In this simple problem the student's program processes the contents of the register, leaves his answer in another register, and returns to the grading program. The latter checks to find out if the correct number has been left in the answer register. Afterwards, the grading program prints out a listing of all the students in the class and their grades. For example:

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 course.)

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

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

1.5 OPERATING SYSTEM USER VIEWPOINT: BATCH CONTROL LANGUAGE

Many users view an operating system only through the batch system control cards by which they must preface their programs. In this section we will discuss a simple monitor system and the control cards associated with it. Other more 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.

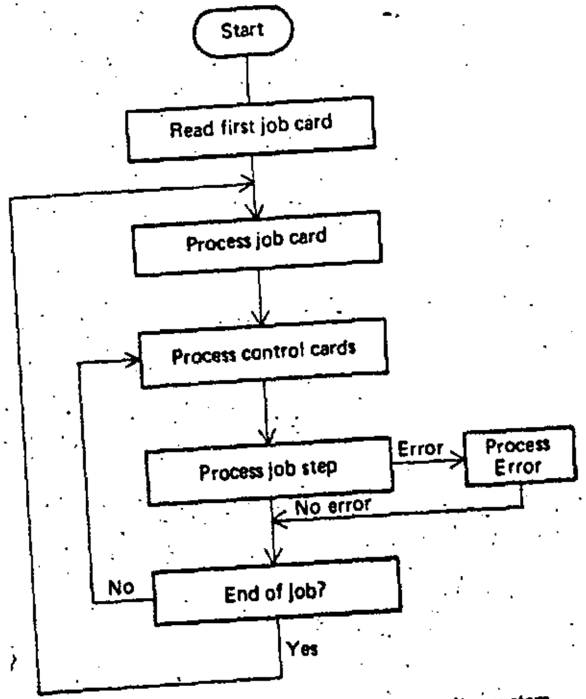


FIGURE 1.4 Main loop of a simple batch monitor system

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 ha[illegible]one 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

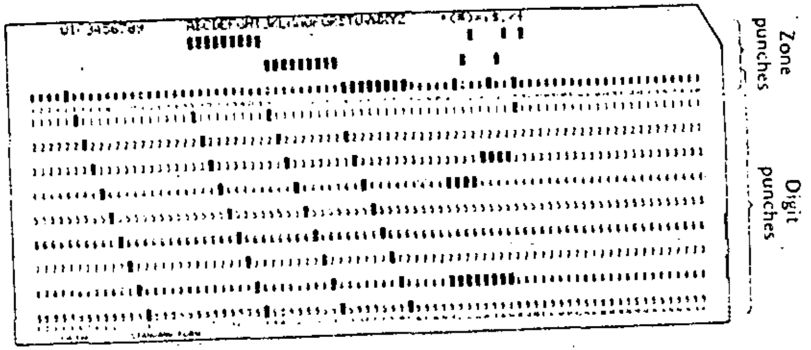


Fig. 10-1. Standard punched card using Hollerith code.

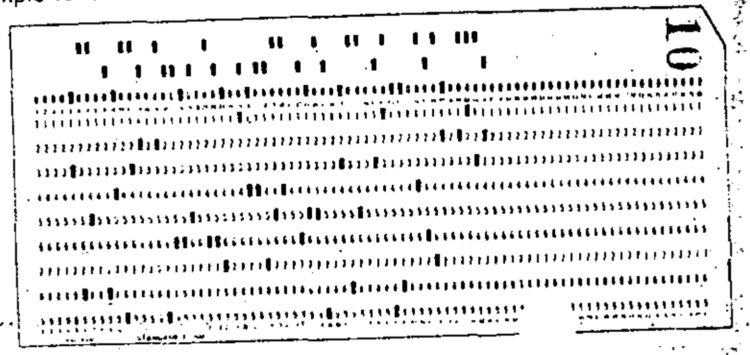
therefore the letter H. Continuing in this fashion, you should see that the complete message reads, "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOGS BACK."

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.

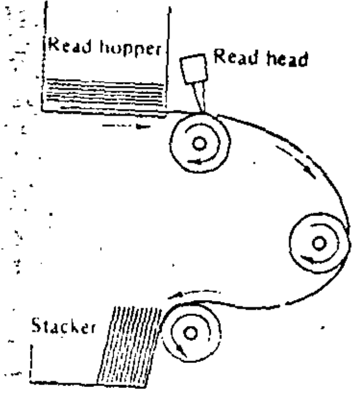


Fig. 10-3. Card-reading operation.

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

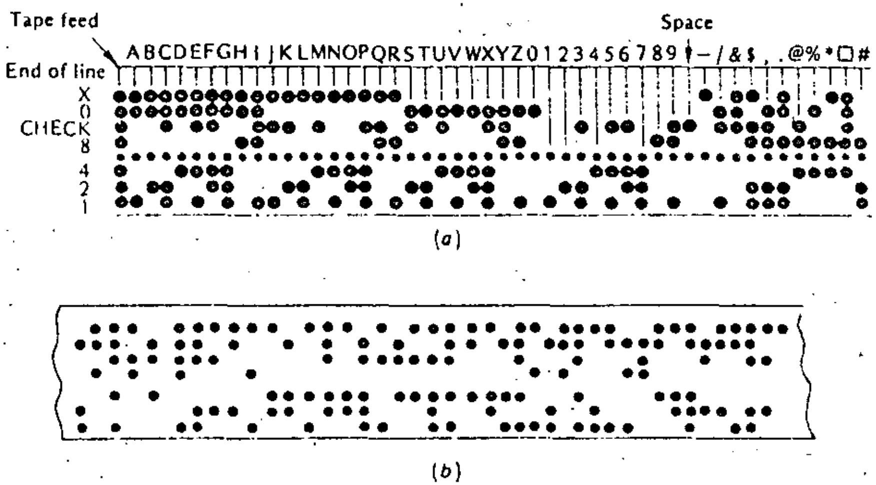
Since each card contains 24 words, the data rate is 24 × 600 = 14,400 words per minute. This is equivalent to 36 × 14,400 = 518,400 bits per minute, or 518,400/60 = 8,640 bits per second.

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.


Tape feed
Space
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 - / & $, . @ % * □ #
End of line
X
0
CHECK
8
4
2
1

(a)

(b)

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

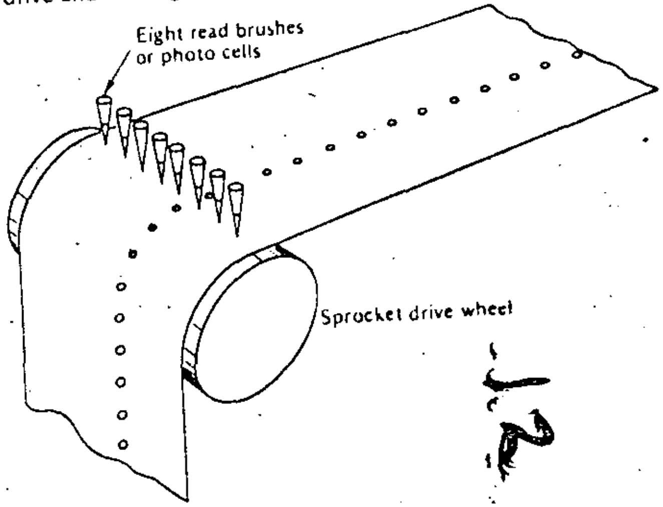
The first character has punches in channels 0, 1, and 2, and this is the letter T. The second character is the letter H, since there are punches in channels X, 0, and 8. Continuing, you should see that the message is the same as that punched on the card in Example 10-1.

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 that

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.

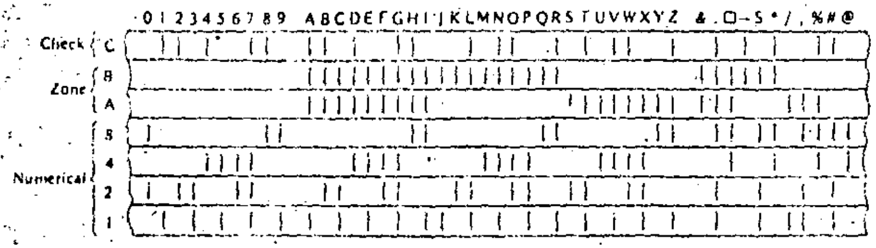
A simplified presentation of the *write* and *read* operations is shown in Fig. 10-7. The magnetic spots are recorded on the tape as it passes over the *write* head as shown in Fig. 10-7*a*. If a positive pulse of current is applied to the *write*-head coil, as shown in the figure, a magnetic flux is 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 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. 10-6. Magnetic-tape code.


0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z & . ☐ – $ * / , % # @
Check
C
Zone
B
A
Numerical
8
4
2
1

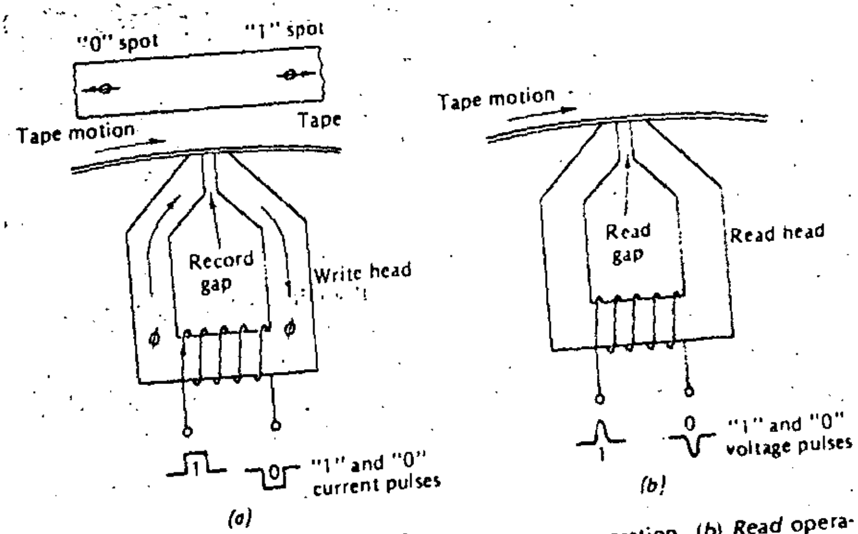


Fig. 10-7. Magnetic-tape recording and reading. (a) *Write* operation. (b) *Read* operation.

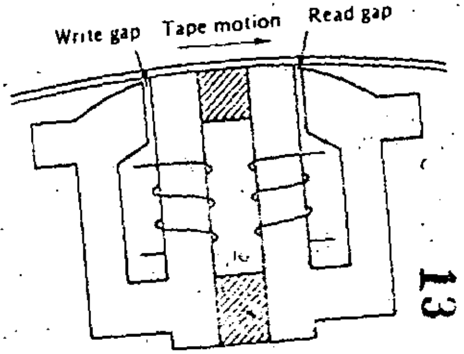


Fig. 10-8. Magnetic-tape *read/write* heads.

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.

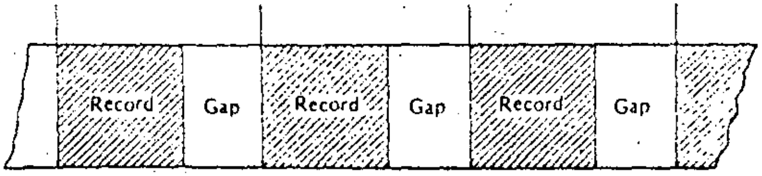
The vertical spacing between the recorded spots on the tape is fixed by the positions of the *read/write* heads. The horizontal spacing is a function of the tape speed and the recording speed. Tape speeds vary from 50 to 200 in/s, but 75 and 112.5 in/s are quite common.

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 G. Thus the total number of characters which could be stored in this length of tape is given by $R + GD$. The ratio of the characters actually recorded R to the total possible could be called a tape-utilization factor F and is given by

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

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 factor.

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

The maximum number of characters that can be stored on the tape is 2,400 × 12 × 556 = 16.0128×10^6. 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-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.

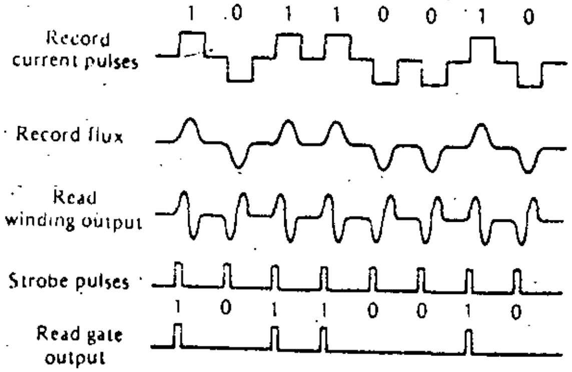


Fig. 10-10. Return-to-zero recording and reading.

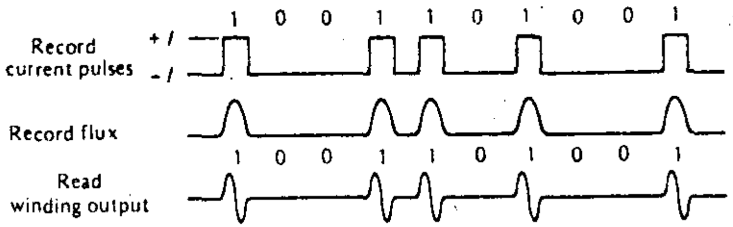


Fig. 10-11. Biased return-to-zero recording and reading.

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

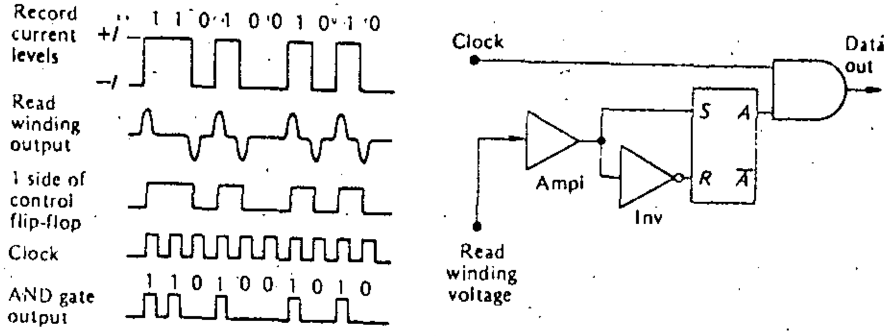


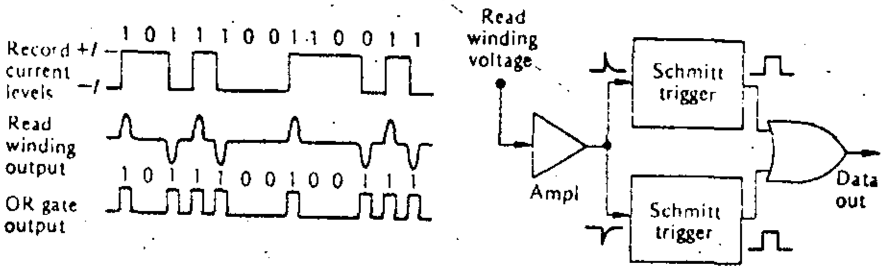
Fig. 10-12. NRZ recording and reading.

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 teletypwriter (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	ƀ	0	@	*P*	Unassigned	
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*		
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_1- - - - - - - - - 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
FOT	End of transmission	$DC_1 - DC_2$	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)
$\overline{SK}$	Skip (punched card)	*ACK*	Acknowledge
LF	Line feed	②	Unassigned control
V/TAB	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

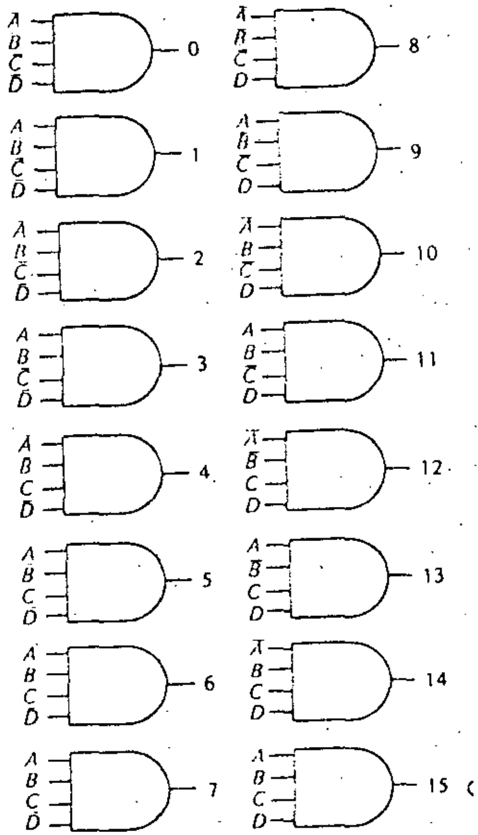

$\bar{A}$ B $\bar{C}$ $\bar{D}$ 0
A B $\bar{C}$ $\bar{D}$ 1
$\bar{A}$ B $\bar{C}$ $\bar{D}$ 2
A B $\bar{C}$ $\bar{D}$ 3
$\bar{A}$ B C $\bar{D}$ 4
A B C $\bar{D}$ 5
A B C $\bar{D}$ 6
A B C $\bar{D}$ 7
$\bar{A}$ B $\bar{C}$ D 8
A B $\bar{C}$ D 9
$\bar{A}$ B $\bar{C}$ D 10
A B $\bar{C}$ D 11
$\bar{A}$ $\bar{B}$ C D 12
A $\bar{B}$ C D 13
$\bar{A}$ B C D 14
A B C D 15

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

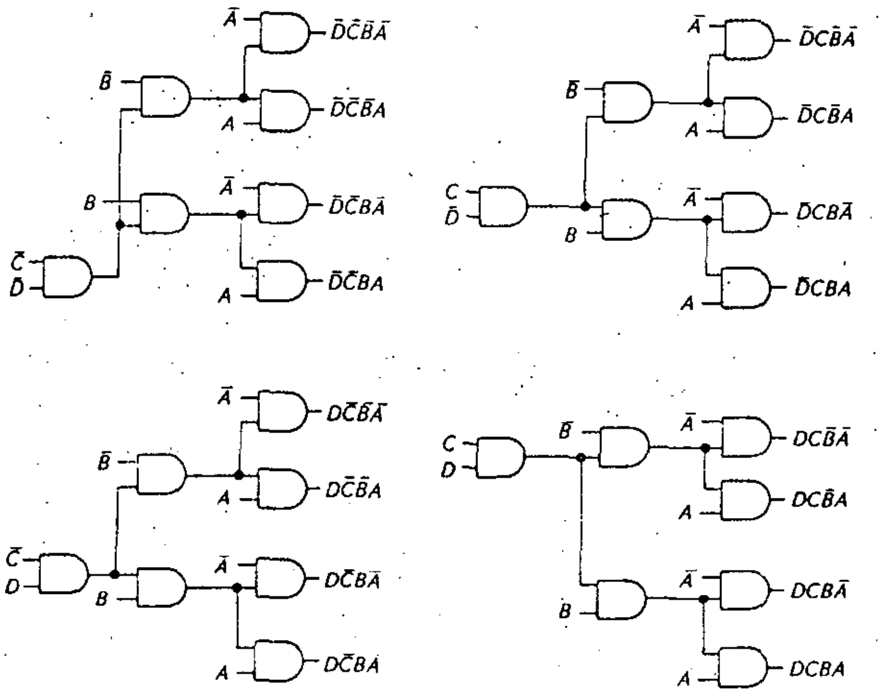
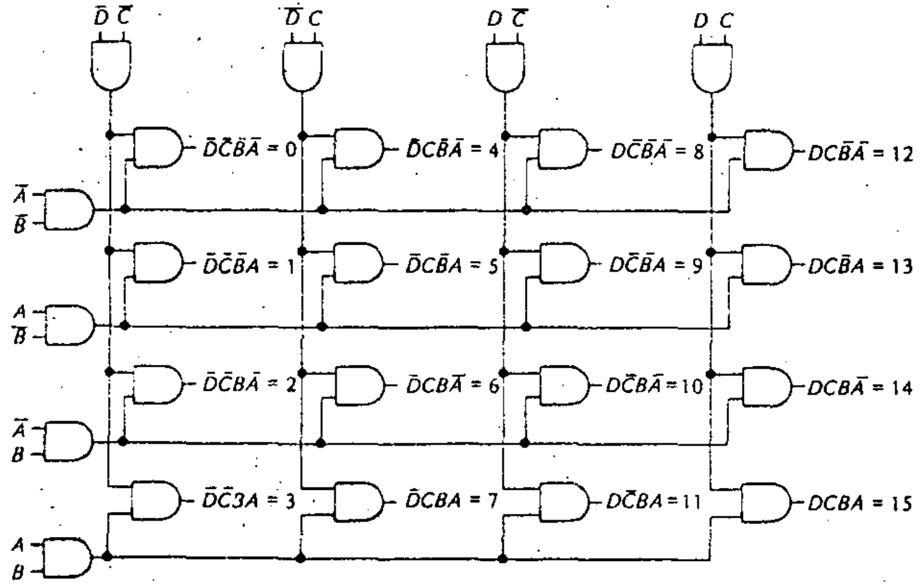


Fig. 10-15. Tree decoding matrix.

Fig. 10-16. Balanced multiplicative decoder.

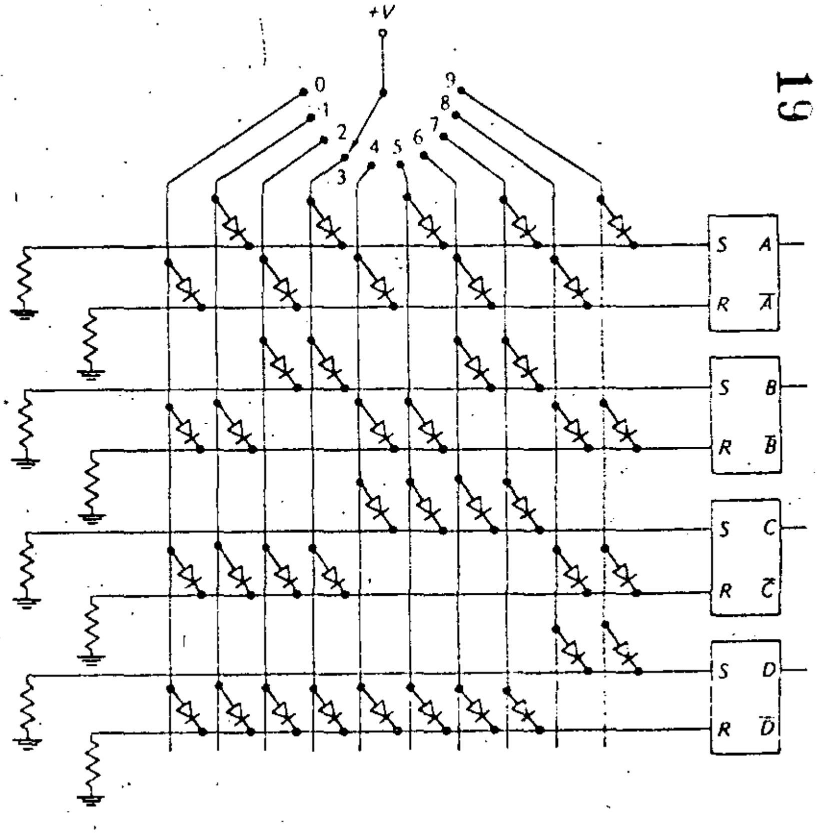


the desired output signals. To decode the four flip-flops discussed previously, four groups are formed by combining flip-flops C and D just as before. In addition, flip-flops B and A are combined in a similar arrangement. The outputs of these eight gates are then combined in 16 AND gates to form the 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.



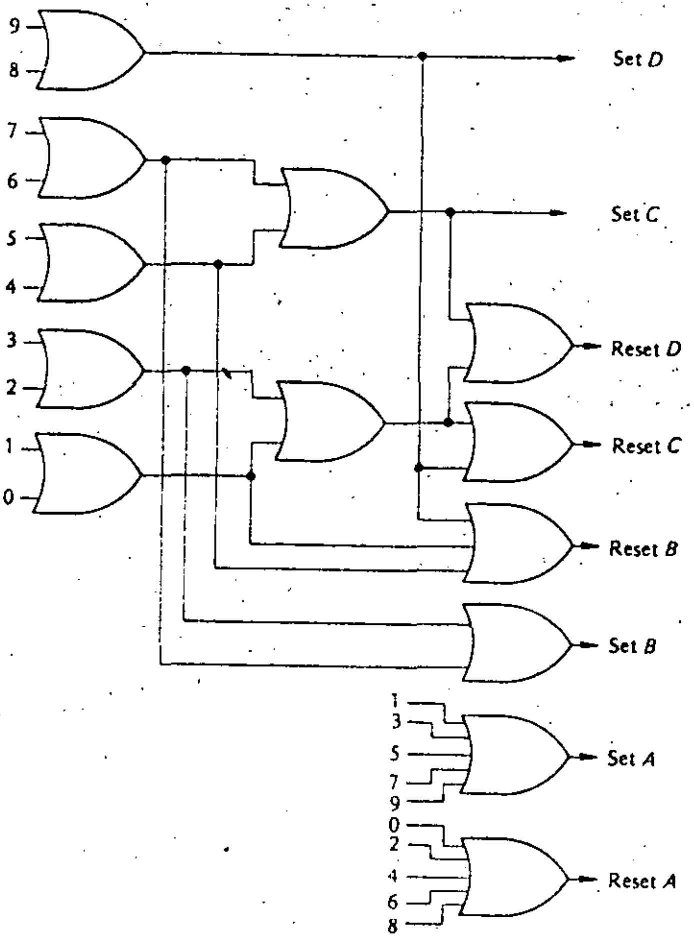


Fig. 10-18. Another decimal encoding matrix.

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 10^6 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.

22

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 ⅛-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.

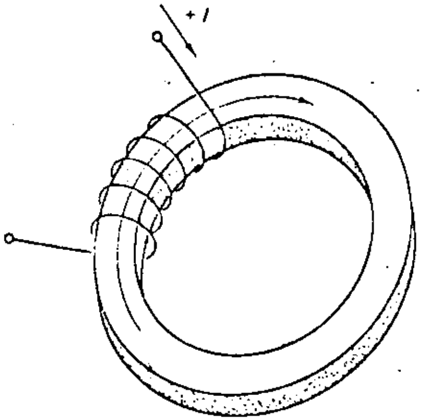


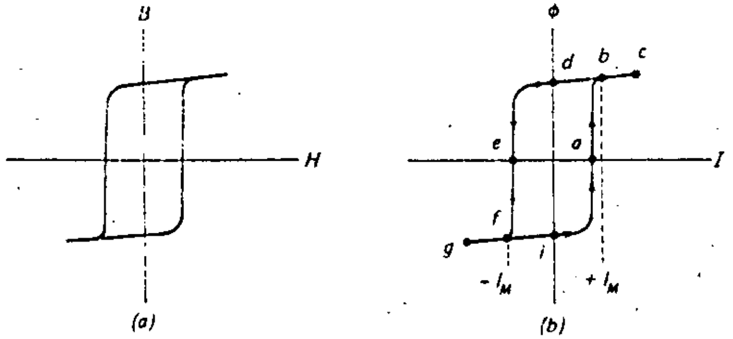
Fig. 12-1. Magnetic core.

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 10^5 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 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.

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



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.

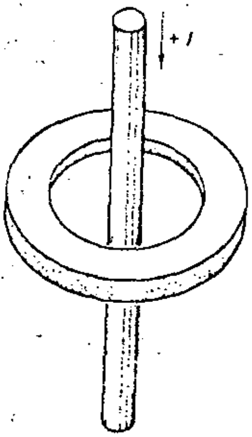


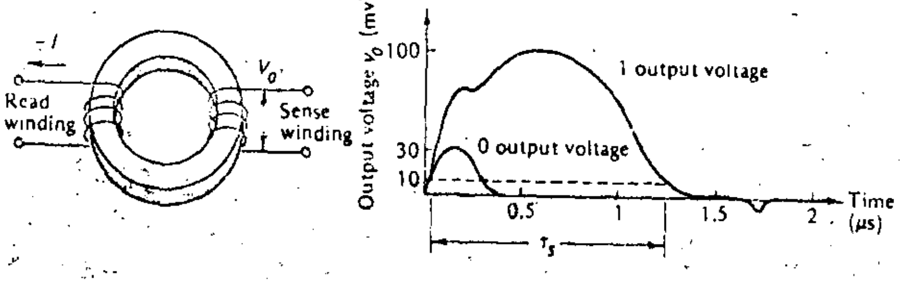
Fig. 12-3.

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 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.

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



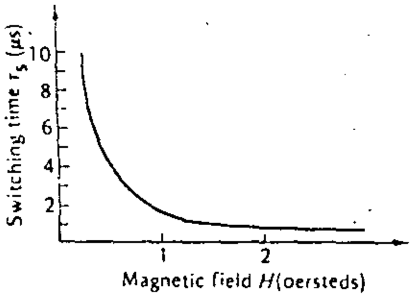


Fig. 12-5. Magnetic-core switching time characteristics.

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 0.1μs 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 detected by resetting the core to the 0 state, any logic system using cores must necessarily be a

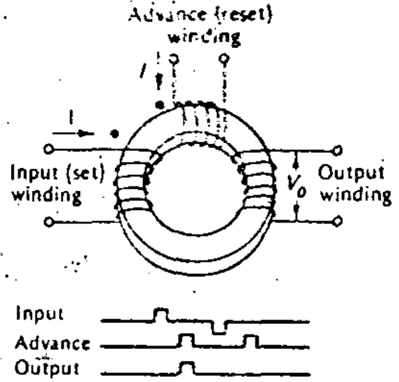


Fig. 12-6. Basic magnetic-core logic element.

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 $X = 1$, 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 $\bar{X} = 0$.

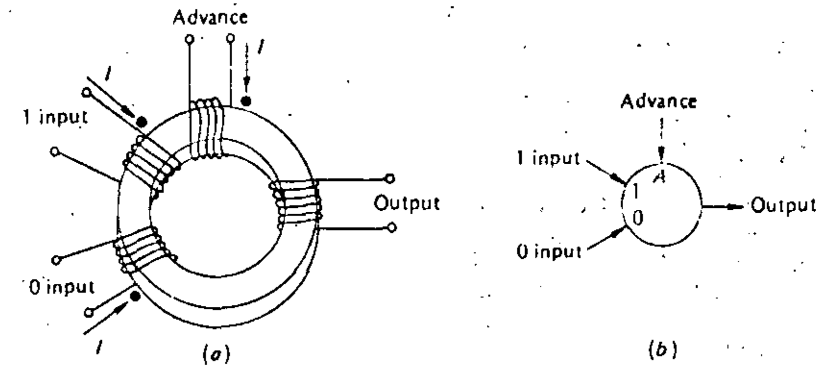
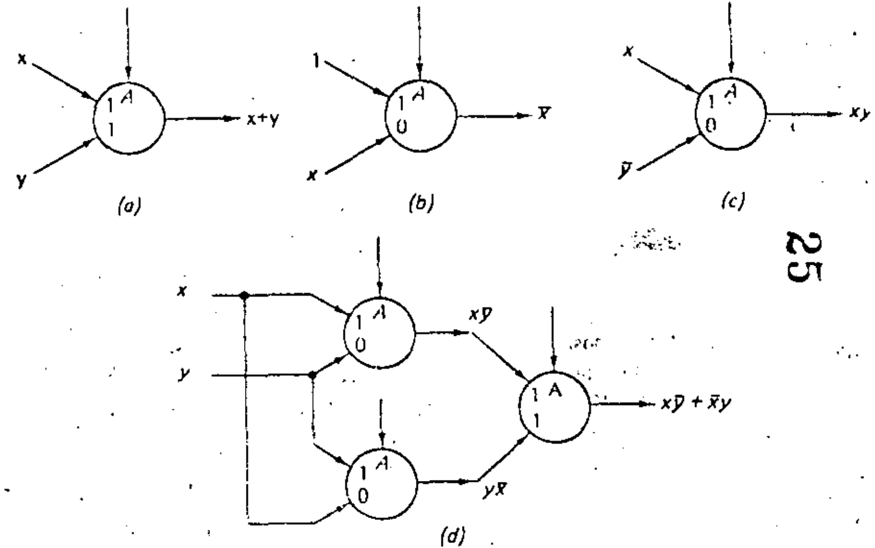


Fig. 12-7. Magnetic-core logic element. (a) Core windings. (b) Logic symbol.

The AND function can be implemented using a core as shown in Fig. 12-8c. 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 certain minimum current I_m is required. This is sometimes referred to as the *select current*. The core arrangement shown in Fig. 12-8*a* can be used to implement an AND function if the X and Y inputs are each limited to one-half the select current $\frac{1}{2}I_m$. In this way, the only time the core can be set is when both X and Y are present, since this is the only time the core receives a full select current I_m. Core logic functions can be constructed using the half-select current idea. This idea is quite important; it forms the basis of one type of large-scale memory system which we discuss later in this chapter.

12-3 MAGNETIC-CORE SHIFT REGISTER

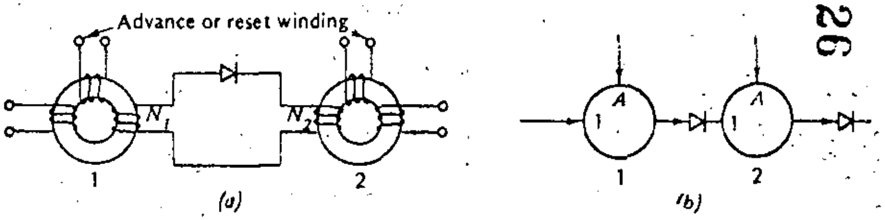
A review of the previous section will reveal that a magnetic core exhibits at least two of the major characteristics of a flip-flop: first, it is a binary device capable of storing binary information; second, it is capable of being set or reset. Thus it would seem reasonable to expect that the core could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

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

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

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

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



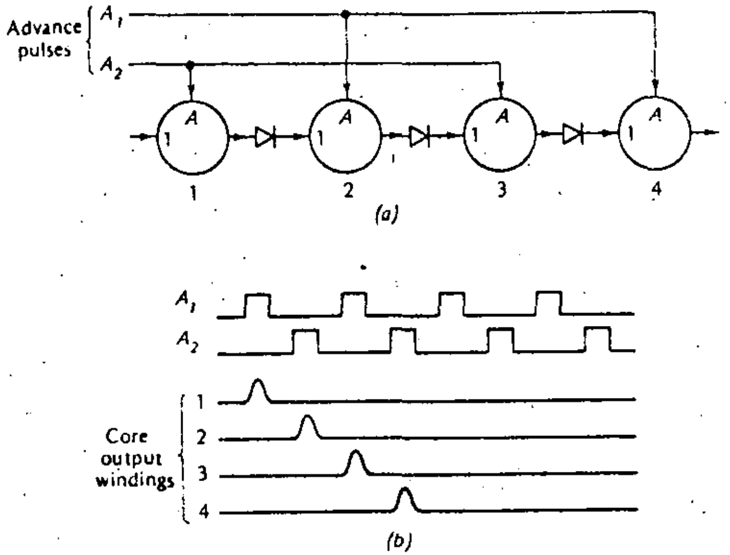


Fig. 12-10. Four-core shift register. (a) Symbolic circuit. (b) Waveforms.

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_2 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_2 has fewer windings than N_1, this reverse signal will not have suffcient 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.

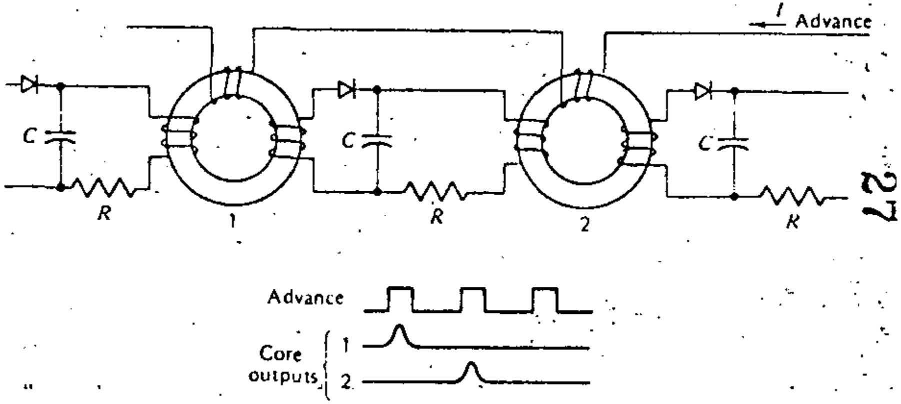
The need for a two-phase clock or advance pulse system could be eliminated if some delay were introduced between the output of each core and the input of the next core. Suppose that a delay greater than the width of the advance pulses were introduced between each pair of cores. In this case, it would 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.

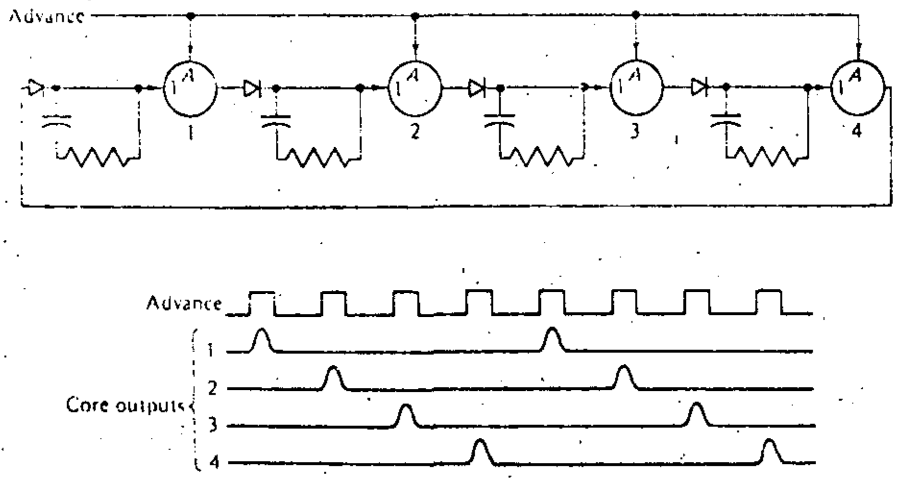
Solution

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 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.

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



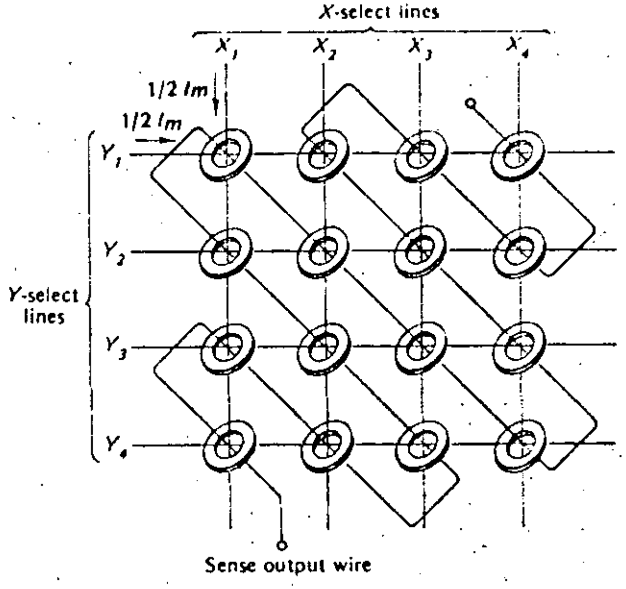


Fig. 12-13. Magnetic-core coincident-current memory.

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_4.

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_1, it is easily seen that current must pass upward through line X_2 in order to set core X_2Y_1. 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.

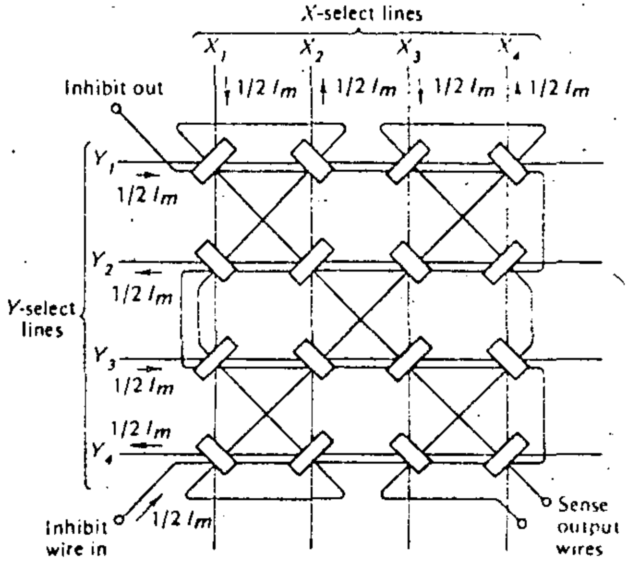


Fig. 12-14. Coincident-current memory matrix (one plane).

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

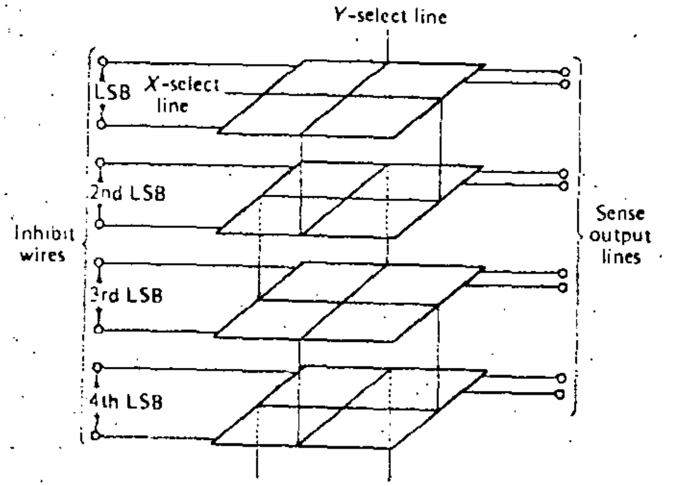


Fig. 12-15. Complete coincident-current memory system.

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 $\frac{1}{2}I_m$ to lines X_a and Y_b results in a selection of core X_aY_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 $-\frac{1}{2}I_m$ 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 $\frac{1}{2}I_m$ 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 $\frac{1}{2}I_m$ to the *inhibit* line. For example, to write 1001 in the upper four planes in Fig. 12-15, we apply $\frac{1}{2}I_m$ to the proper X and Y lines and at the same time apply $\frac{1}{2}I_m$ 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, $-\frac{1}{2}I_m$ 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 currents disappear,

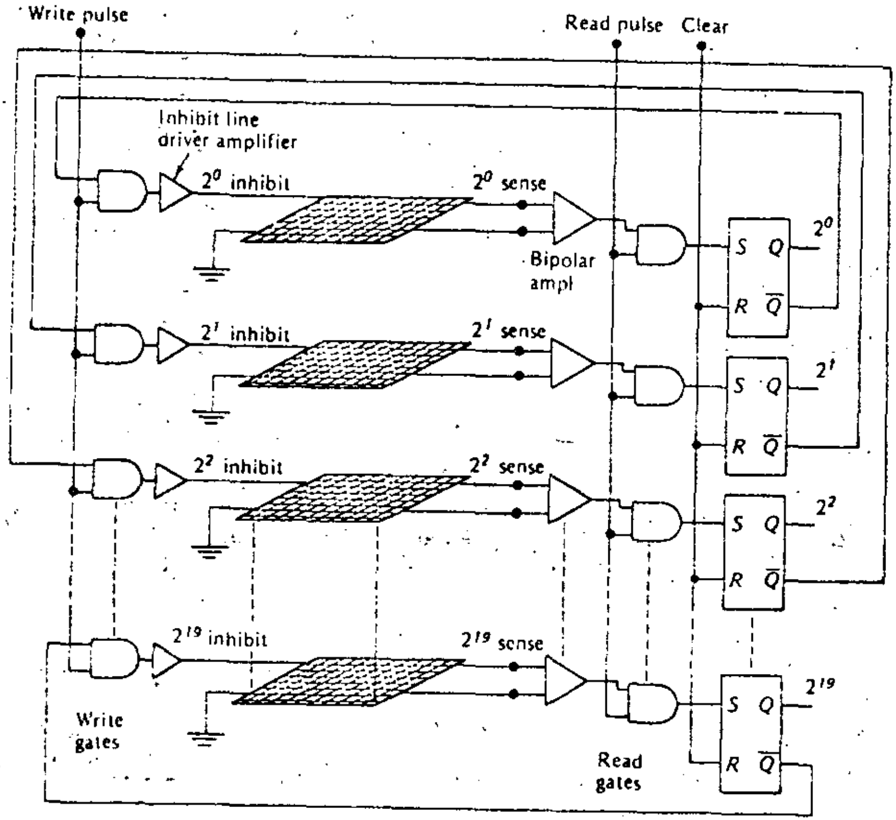


Fig. 12-16. NDRO system for Example 12-6.

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 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.

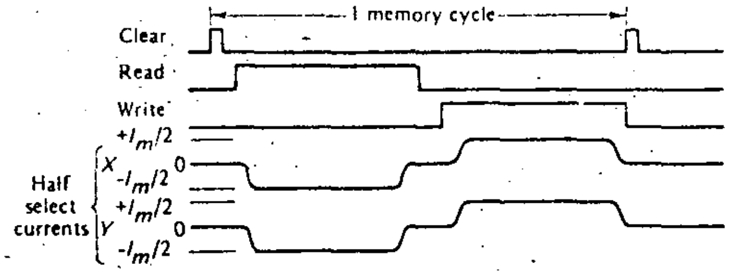


Fig. 12-17. NDRO waveforms for Fig. 12-16 (read from memory).

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.

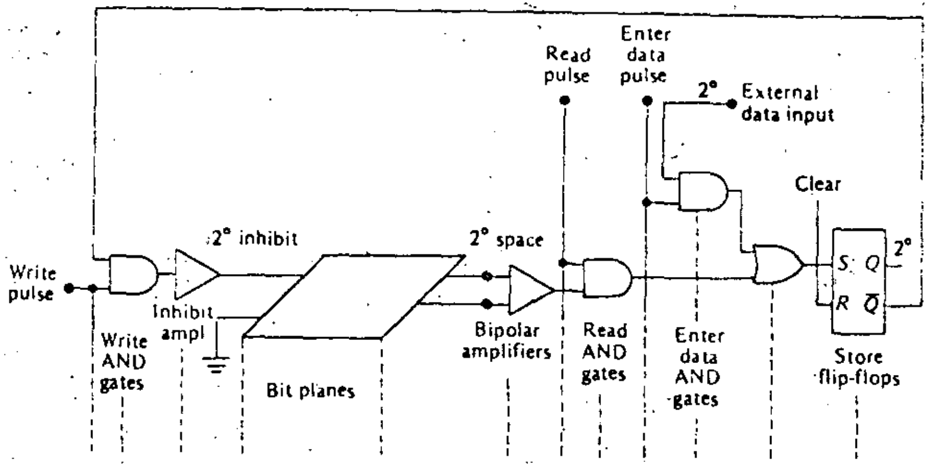


Fig. 12-18. Complete NDRO memory system (LSB plane only).

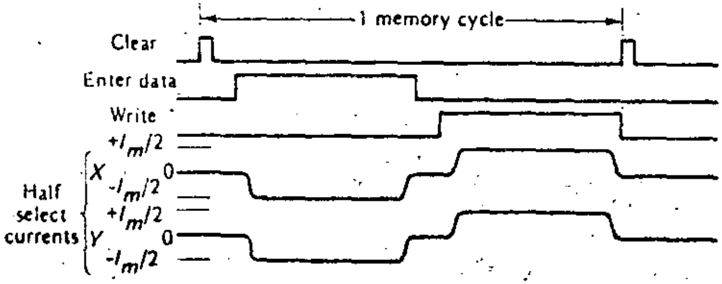
3. During the *write* pulse, data held in the flip-flops are stored in the cores exactly as before.

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 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.

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



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. 12-20*a*. 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

$$\text{Squareness ratio} = \frac{\mathbf{B}_r}{\mathbf{B}_m}$$

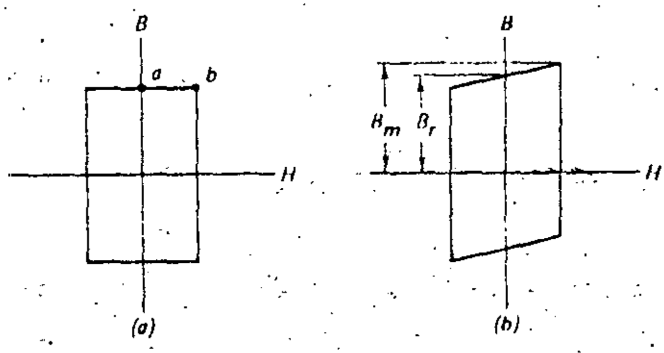
This is the ratio of the flux density at the remanent point $\mathbf{B}_r$ to the flux density at the switching point $\mathbf{B}_m$ and is shown graphically in Fig. 12-20*b*. The ideal value is, of course, 1.0, but values between 0.9 and 1.0 are the best obtainable.

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 $Y_1, Y_2, Y_3, \ldots, Y_n$. 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.

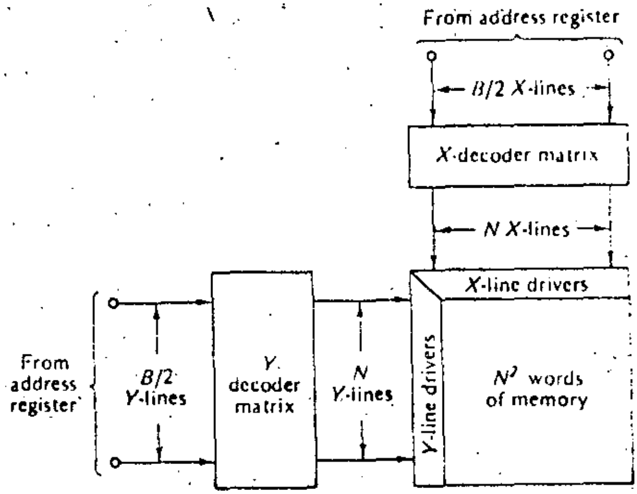


Fig. 12-21. Coincident-current memory addressing.

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-

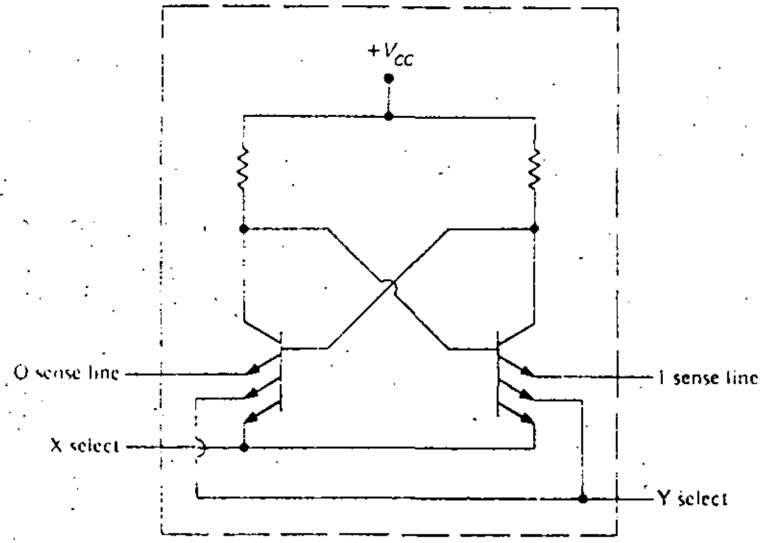


Fig. 12-22. Bipolar memory cell circuit.

trix 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. 12-24*a*. 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. 12-24*b*. 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 ($A_0A_1A_2A_3A_4A_5A_6A_7A_8A_9$) to obtain 1,024 words. That is, x bits provide 2^x word

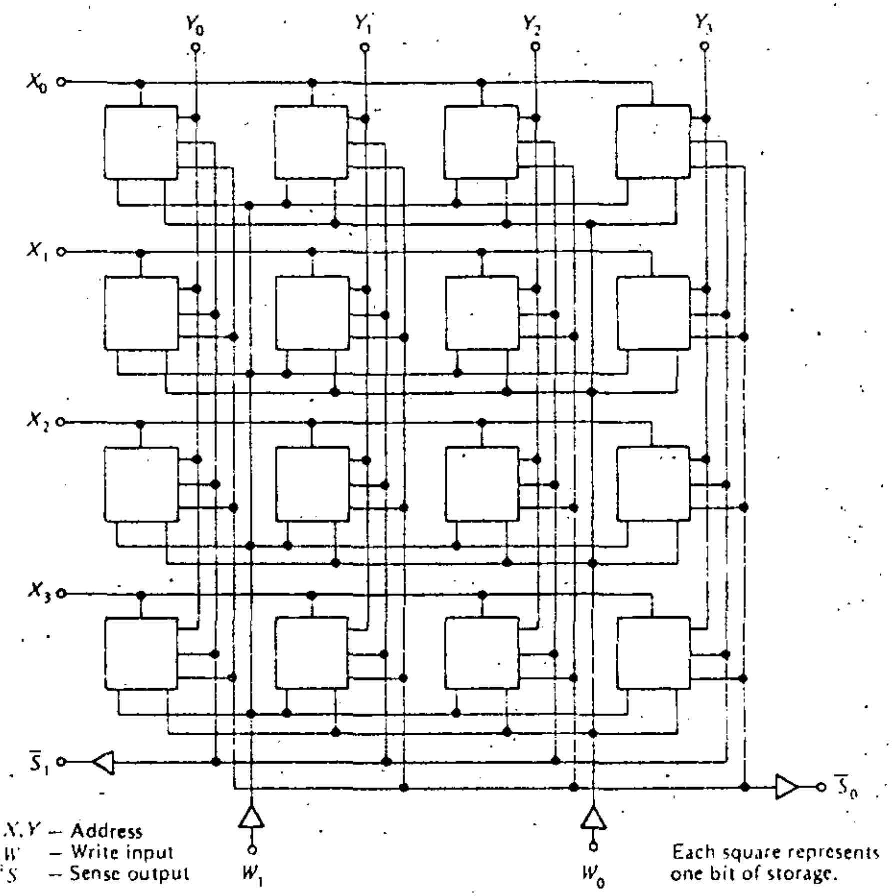


Fig. 12-23. 16-word 1-bit memory.

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 ($2^5 = 32$).

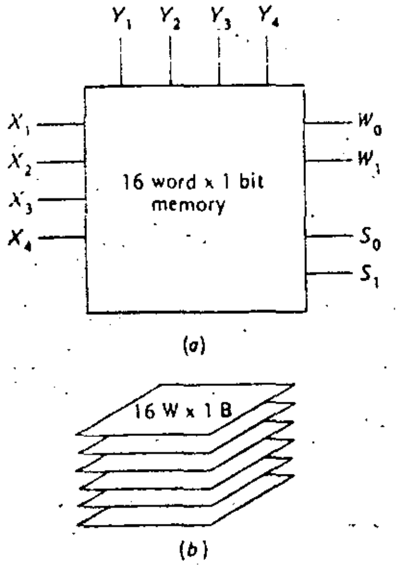
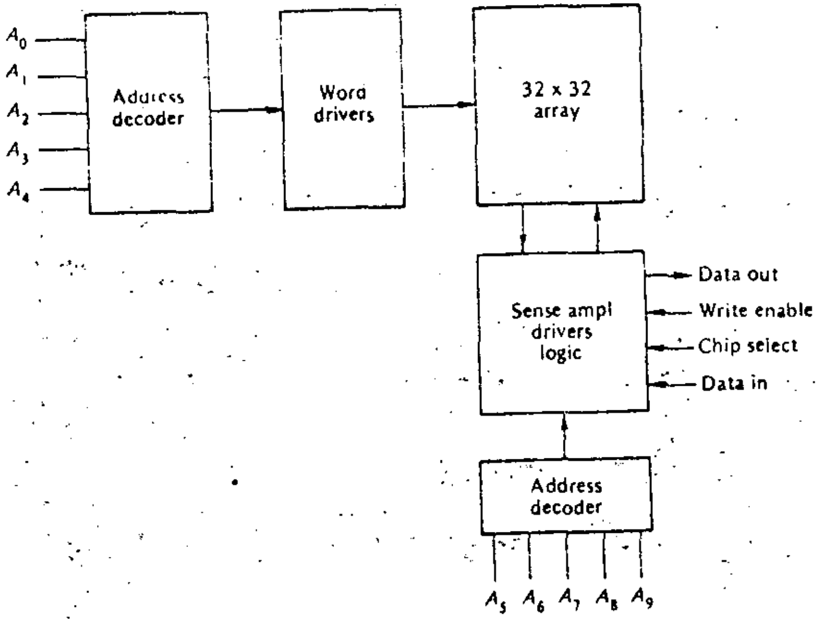


Fig. 12-24. (a) Logic diagram. (b) Six chips stacked to get a 16-word × 6-bit memory.

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?

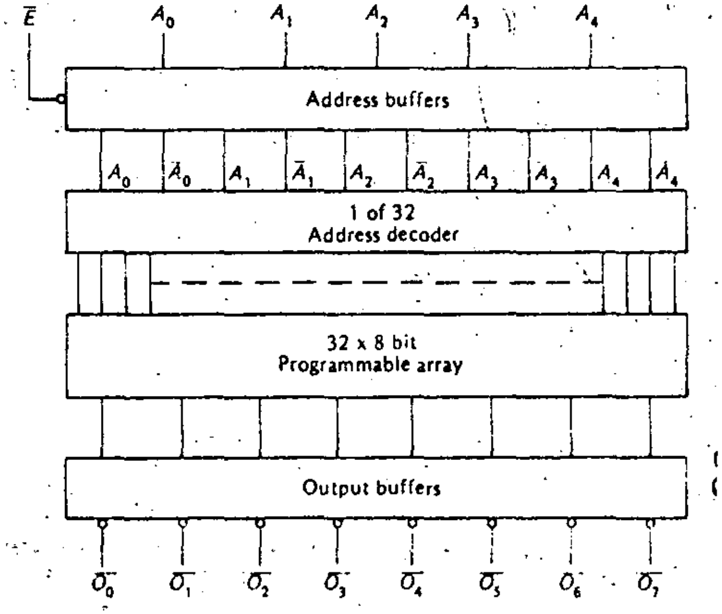
Solution

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.


$\bar{E}$
A_0 A_1 A_2 A_3 A_4
Address buffers
A_0 $\bar{A}_0$ A_1 $\bar{A}_1$ A_2 $\bar{A}_2$ A_3 $\bar{A}_3$ A_4 $\bar{A}_4$
1 of 32 Address decoder
32 x 8 bit Programmable array
Output buffers
$\bar{O}_0$ $\bar{O}_1$ $\bar{O}_2$ $\bar{O}_3$ $\bar{O}_4$ $\bar{O}_5$ $\bar{O}_6$ $\bar{O}_7$

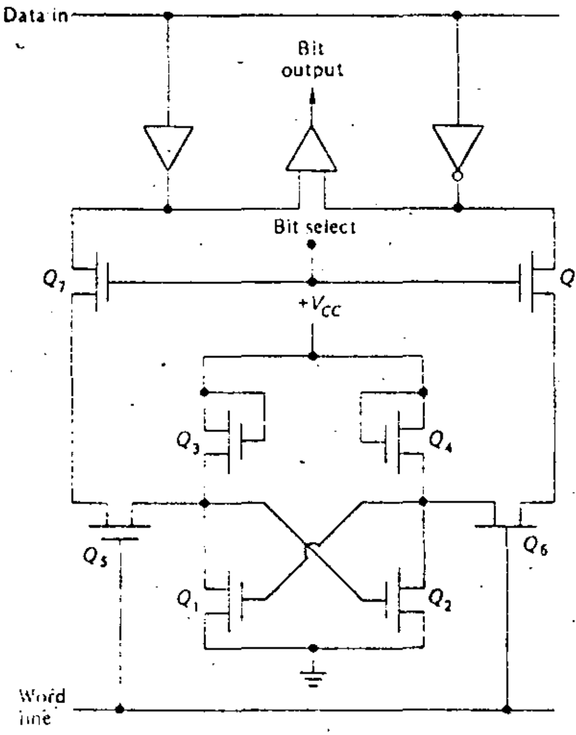


Fig. 12-27.

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

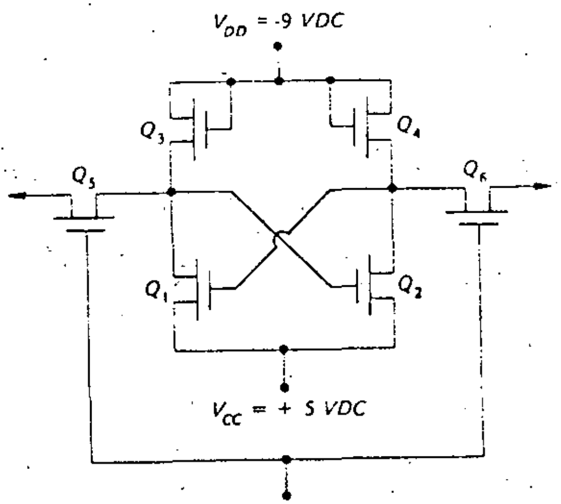
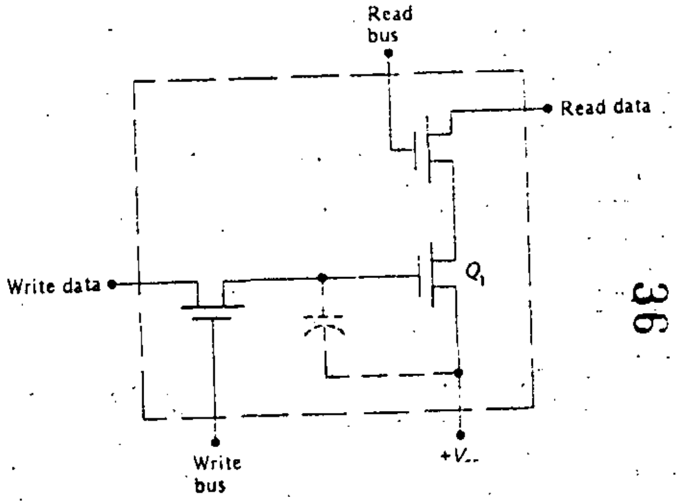


Fig. 12-28.

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.



36

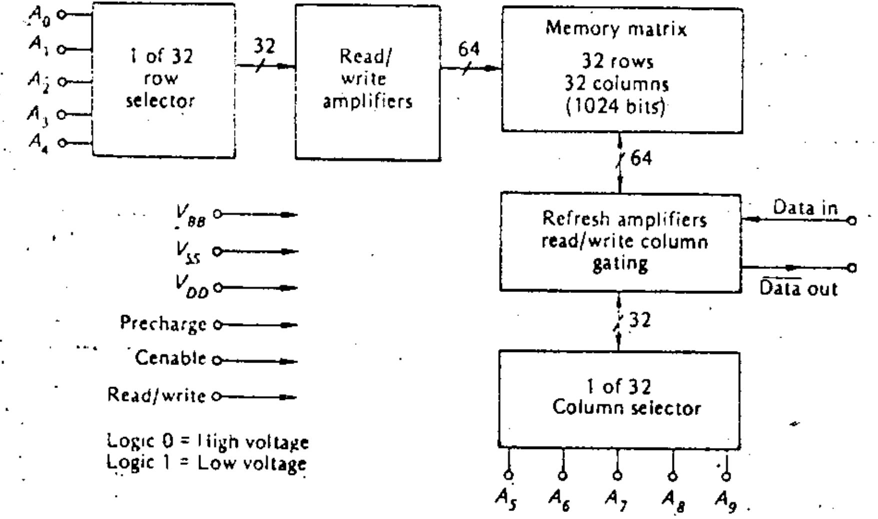


Fig. 12-30. 1103 Dynamic RAM logic diagram.

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 $+V_{ss}$; 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 × 100 tracks =

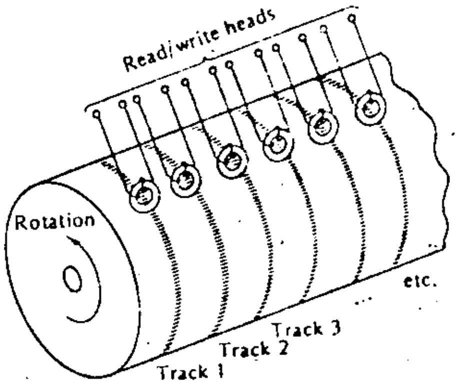


Fig. 12-31. Magnetic-drum storage.

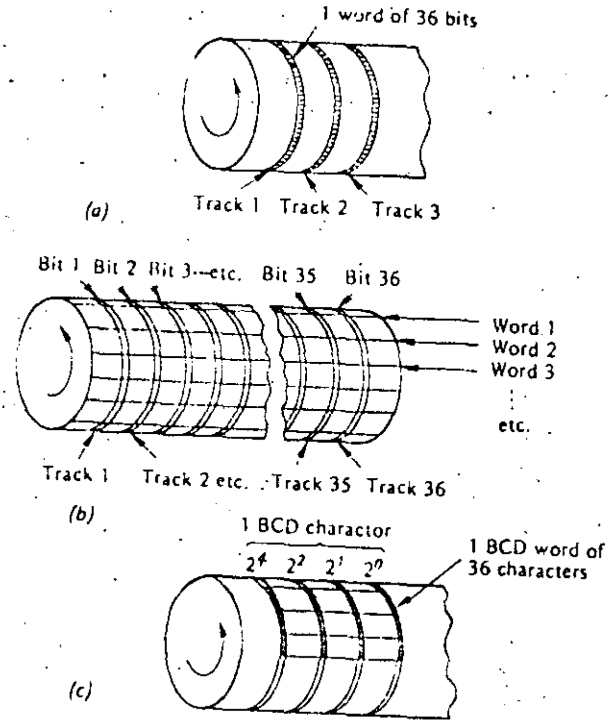


Fig. 12-32. Magnetic-drum organization. (*a*) Bit-serial storage. (*b*) Bit-parallel storage. (*c*) Bit-serial-parallel storage.

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 $\cong$ 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. 12-32*b*.

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 × 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 × 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. Alternatively, 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?

39

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?

40

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

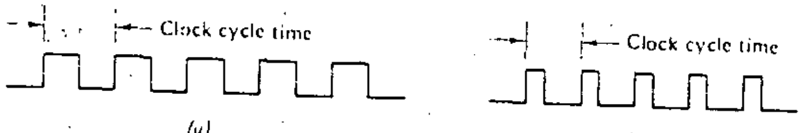


Fig. 14-1. Basic system clock.

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. 14-1*b*. 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?

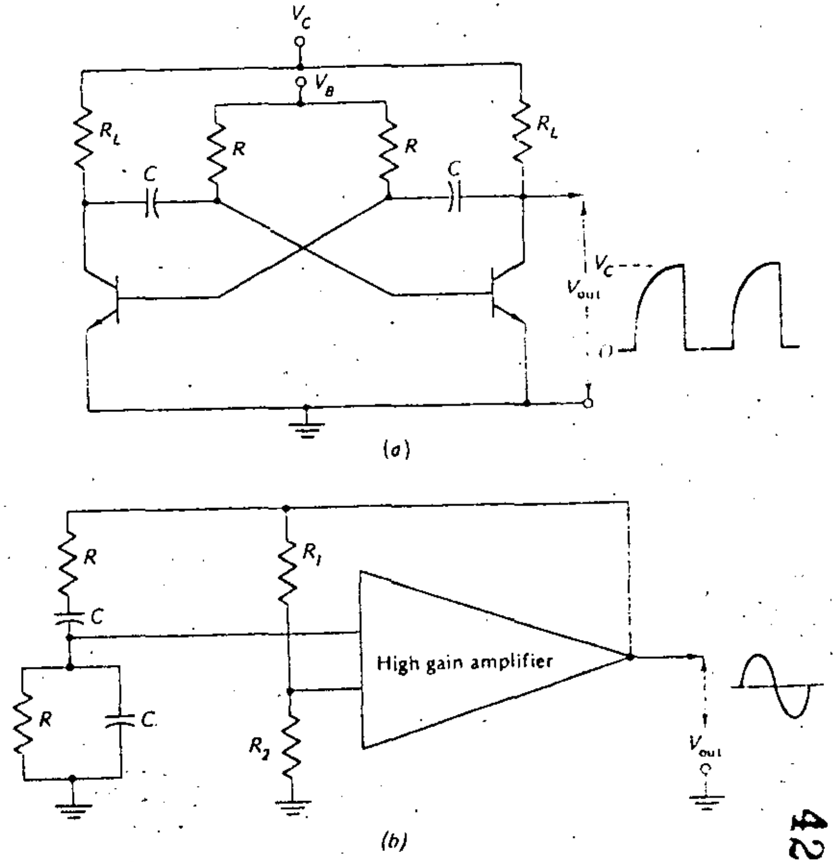
Solution

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

$$f \cong \frac{1}{2RC \ln (1 + V_C/V_B)} \tag{14-1}$$

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



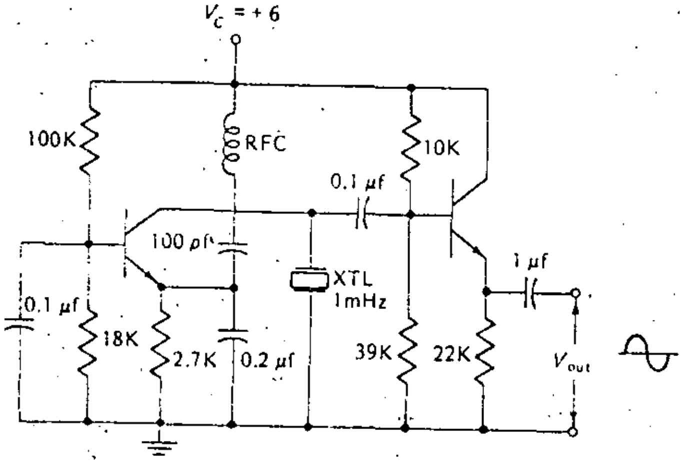


Fig. 14-3. Crystal oscillator.

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 10^8 per day.

Example 14-3

The multivibrator in Fig. 14-2*a* 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.

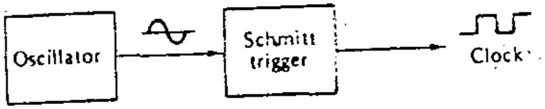


Fig. 14-4. Oscillator and output amplifier.

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.

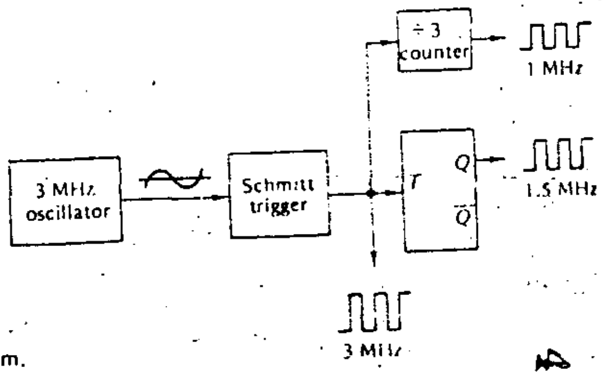


Fig. 14-5. Basic clock system.

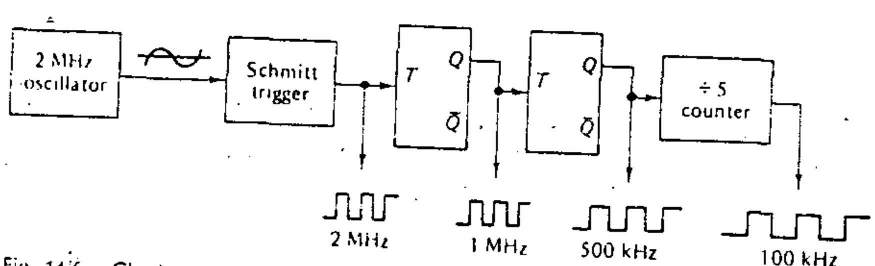


Fig. 14-6. Clock system.

Example 14-4

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

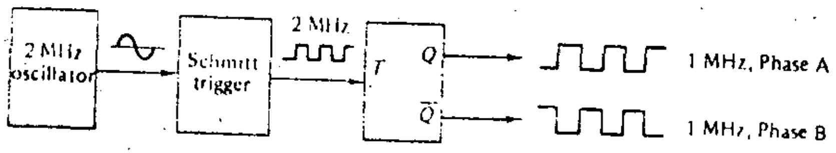
Solution

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 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.

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



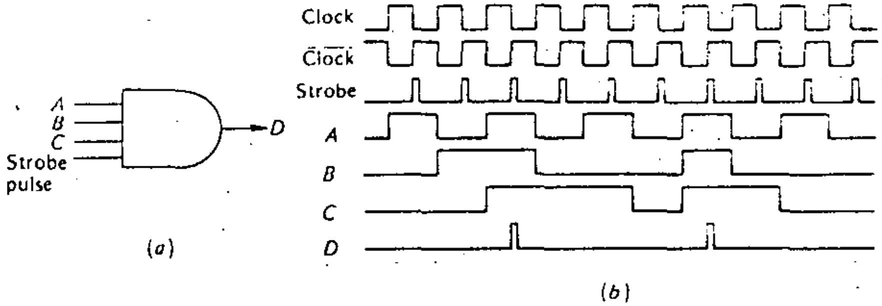
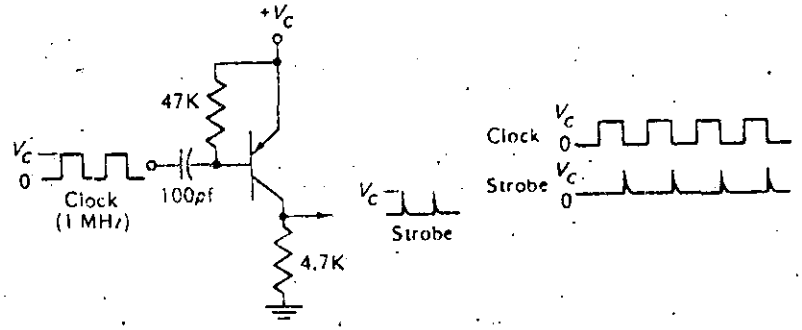


Fig. 14-8. The use of a strobe pulse. (*a*) Three-input AND interrogated by a strobe pulse. (*b*) Waveforms for the AND gate.

14-3 MPG 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 vol-

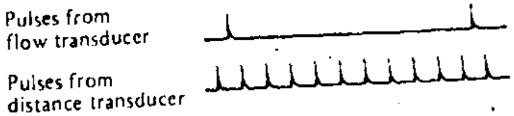


Fig. 14-10. Transducer pulses for the MPG computer when the rate is 10 miles per gallon.

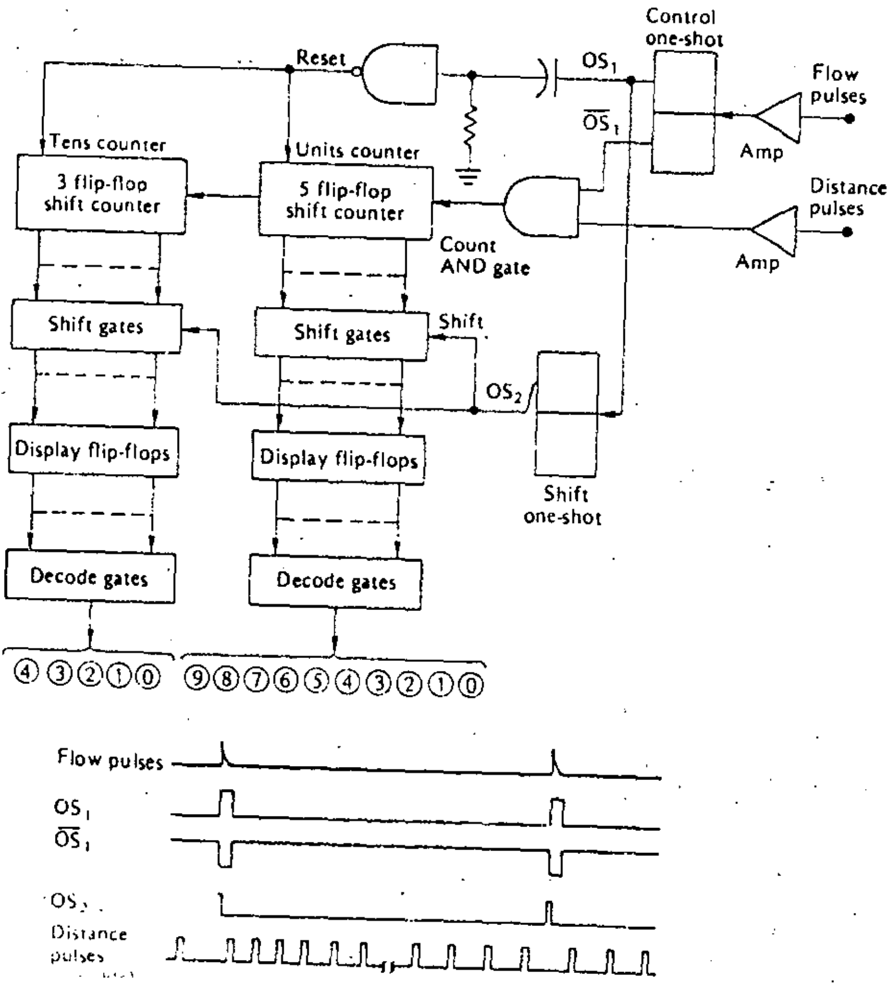
ume of fuel flowing into the engine. This flow tranducer provides an electrical pulse each time 1/1000 of a gallon of fuel passes through it. The second transducer is used to measure the distance traveled and is driven by the speedometer cable. This distance transducer provides an electrical pulse each time the vehicle has traveled a distance of 1/1000 of a mile.

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

Fig. 14-11. Complete MPG computer.



One conversion time is the time between two flow pulses, and we want to shift the accumulated count into the display flip-flops at the end of each conversion cycle. Notice first of all that, when $\overline{OS}_1$ is low, the *count* AND gate is disabled and therefore the *units* and *tens* counters cannot change states. It is during this time that we must shift the contents of these counters into the display flip-flops. We use the leading edge of OS_1 to trigger the *shift* one-shot and develop the shift waveform OS_2. The falling edge of OS_2 is applied to the shift gates, and at this time the count stored in the *units* and *tens* counters is shifted into the display flip-flops. The falling edge of OS_1 is then used to reset all flip-flops in the *units* and *tens* counters. The contents of the display flip-flops are then decoded and used to illuminate the indicator lights. In this system, the distance pulses can be considered to be the basic system clock. The flow pulses form a variable control gate by means of the *control* one-shot which determines the period of time that the *count* AND gate is enabled and therefore the number of distance pulses counted. The output of the *shift* one-shot OS_2 can be considered as a strobe pulse which shifts data from the counters into the display flip-flops in such a way that racing is avoided. The system clearly 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-purpose computer since it is designed and constructed to perform a single function; to alter it so that it could perform another function would require a major change in design. On the other hand, a general-purpose computer is designed so that it can perform a number of fundamental operations—addition, subtraction, multiplication, division, comparison, etc. The computer can then be used in any number of different applications by simply instructing it to perform the appropriate operations in an orderly fashion. The functions to be performed, listed in the order in which they are to be accomplished, is known as a *program* (instruction set). This list of instructions, or program, is normally stored in the computer memory; when the computer is started, it simply performs these instructions in the order stored. Herein lies the difference between an electronic calculator and a general-purpose digital computer—the calculator performs a function (add, subtract, etc.) each time an operator depresses a button, but the stored-program computer performs the complete list of stored instructions without human intervention. Furthermore, the computer is capable of completing the instruction set in a very short period of time (addition in perhaps a few microseconds), and the operation is virtually error free.

The simplified block diagram in Fig. 14-12 shows the basic units to be found in any general-purpose computer system. The *input/output* block represents the interface between man and machine. It could simply be a teletype unit, where input information is typed in on the keyboard and output information is printed on paper. It could also represent any of the other input/output media previously discussed, such as punched paper tape, punched unit-record cards, and magnetic tape. In any case,

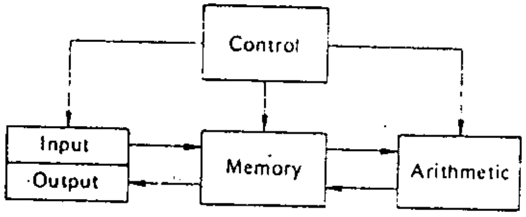


Fig. 14-12. Basic computer unit block diagram.

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.

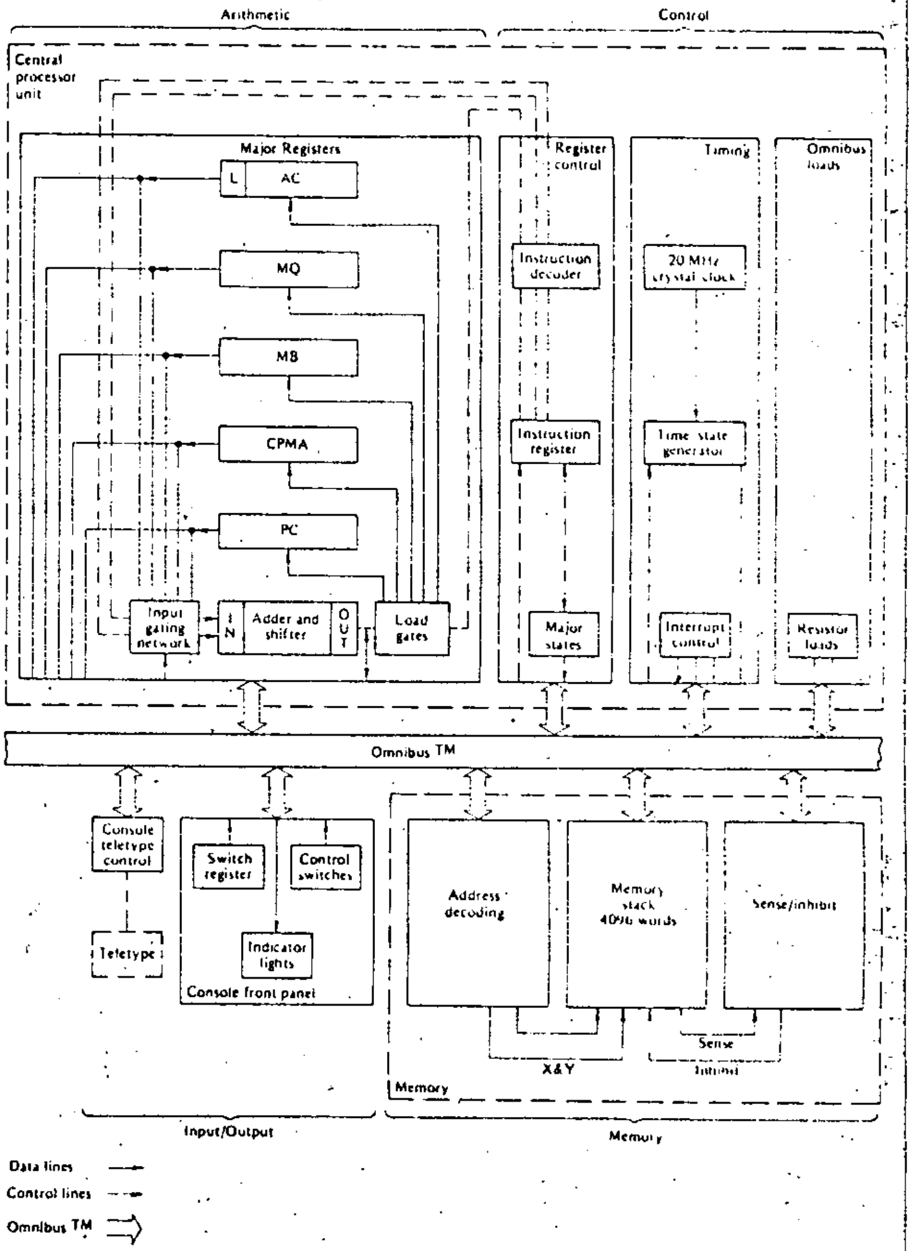


Fig. 14-13. PDP-8/E basic system block diagram.

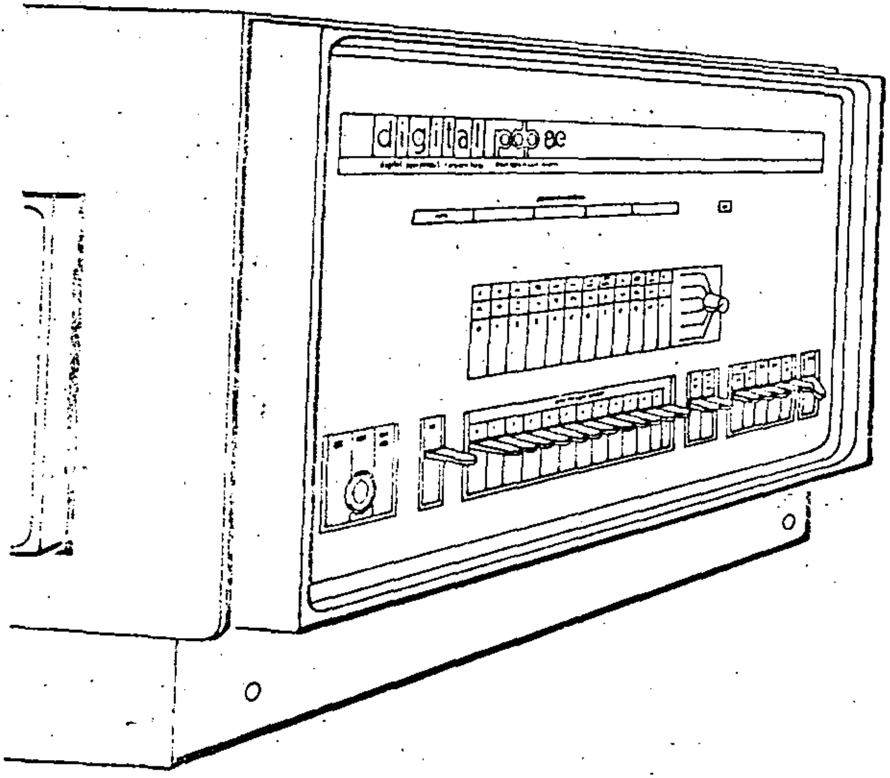


Fig. 14-14. PDP-8/E programmed data processor.

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 indiret 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

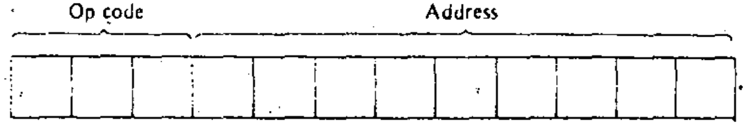


Fig. 14-15. Instruction word format.

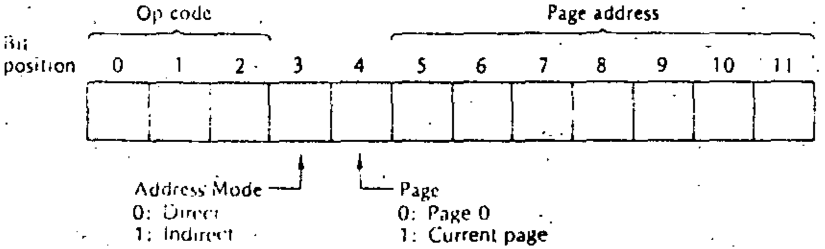
memory located at address 12_{10} (000001100) to the contents of the accumulator register in the arithmetic unit.

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.



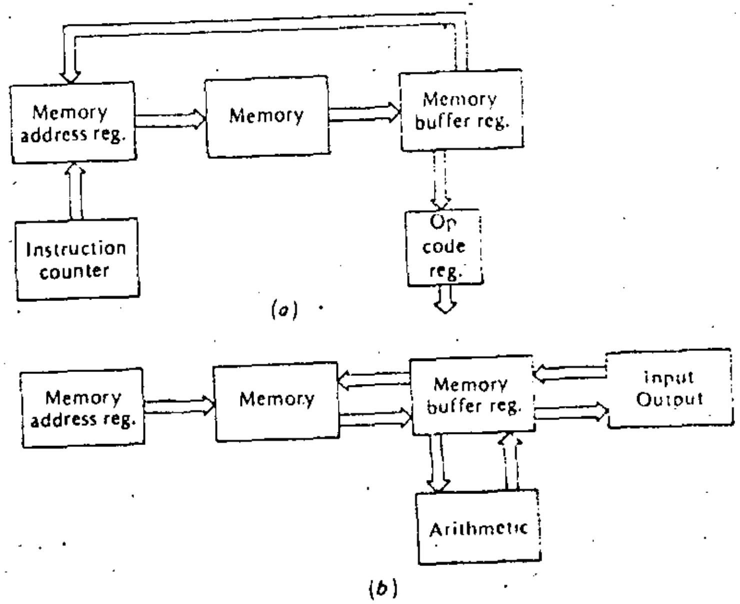


Fig. 14-17. Basic computer operating cycles. (a) *Fetch*. (b) *Execute*.

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. 14-2a oscillate if $R = 100$ kΩ, $C = 100$ pF, $V_c = 20$ v dc, and $V_B = 10$ v dc?

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

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

14-7. What is the oscillating frequency of the Wien-bridge oscillator in Fig. 14-2b is $R = 47$ 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

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

ARQUITECTURA DE LA PDP-11

1734

PDP-11
04/34/45/55
PROCESSOR
HANDBOOK

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.

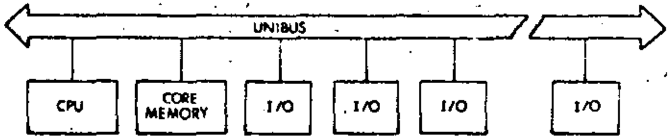


Figure 2-1 PDP-11 System Simplified Block Diagram

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.

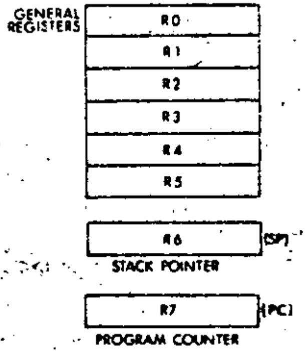
Full 16-bit words or 8-bit bytes of information can be transferred on the bus between a master and a slave. The information can be instructions, addresses, or data. This type of operation occurs when the processor, as master, is fetching instructions, operands, and data from memory, and storing the results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



Fgure 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

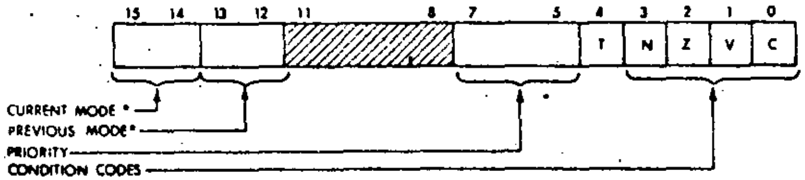
ADD A,B	;add contents of location A to location B, store results at location B

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.

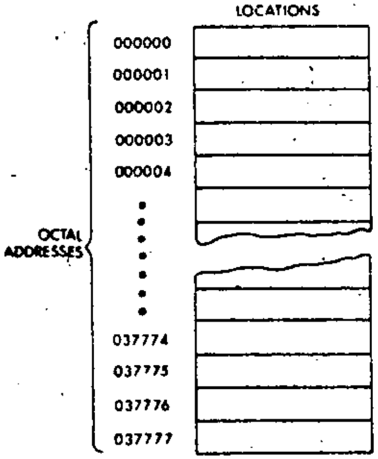


Figure 2-4 Memory Addresses

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.

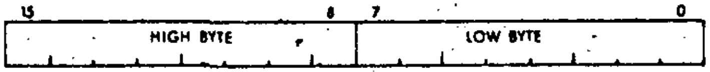


Figure 2-5 High & Low Byte

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.

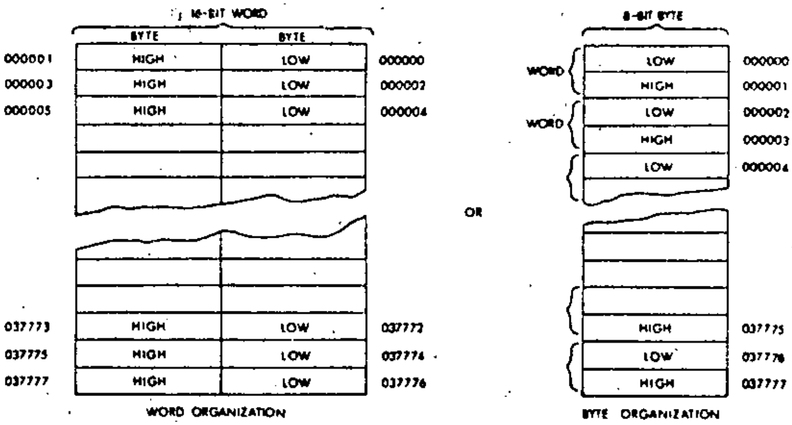


Figure 2-6 Word and Byte Addresses

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.

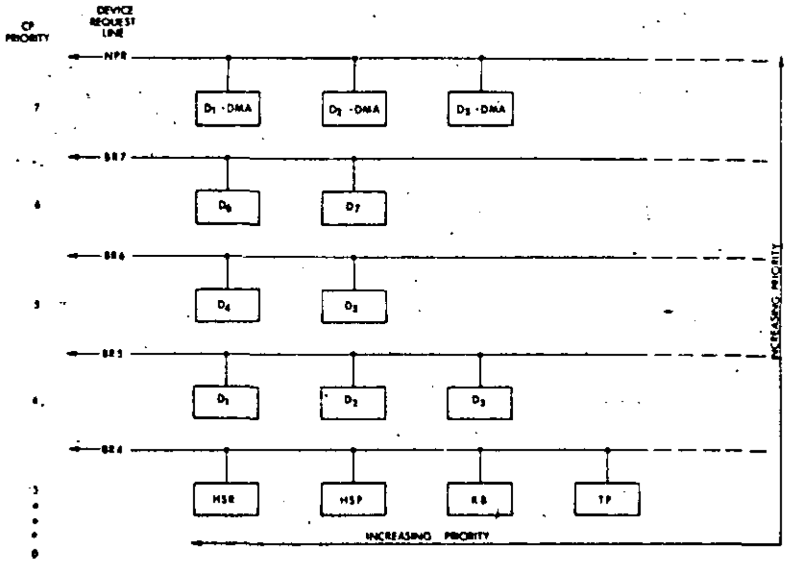


Figure 2-7 UNIBUS Priority

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 multiprogramming 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.

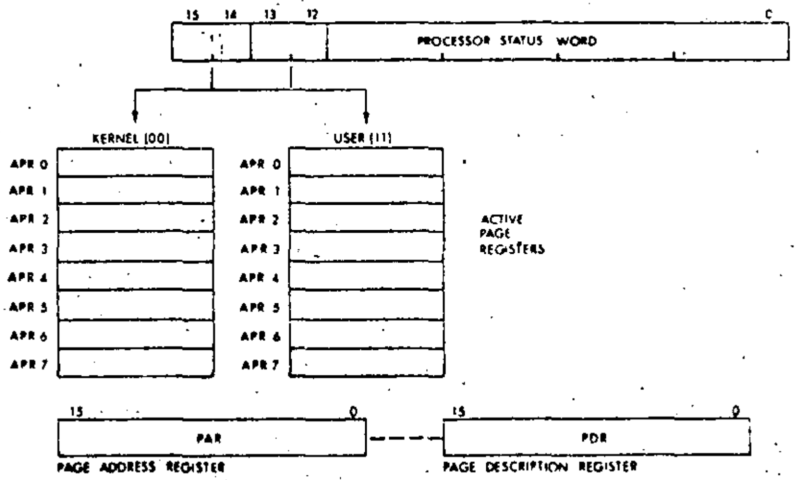


Figure 8-1 Active Page Registers

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.

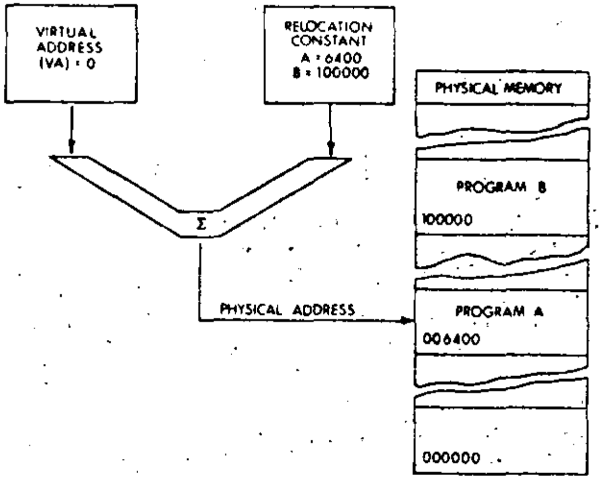


Figure 8-2 Simplified Memory Relocation Concept

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 then 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)

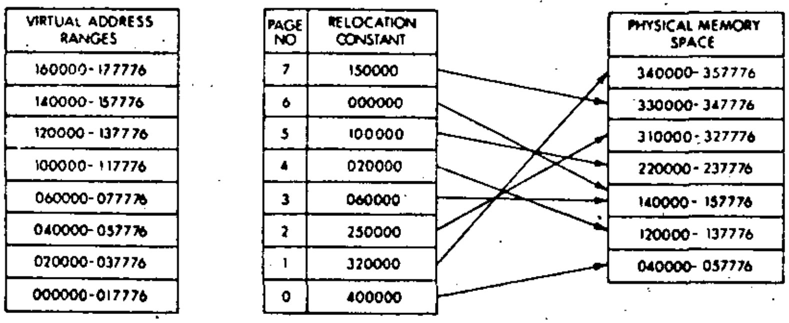


Figure 8-3 Relocation of a 32K Word Program into 124K Word Physical Memory

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 10	Illegal (all references aborted on access)
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.

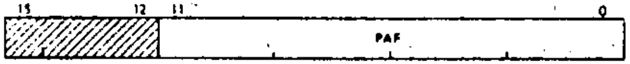


Figure 8-4 Page Address Register

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.

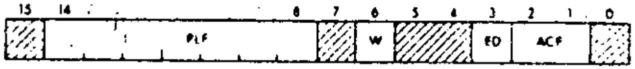


Figure 8-5 Page Descriptor Register

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.

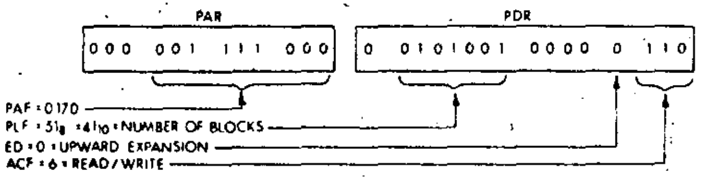
In the context of access control, the term "write" is used to indicate the action of any instruction which modifies the contents of any addressable word. A "write" is synonymous with what is usually called a "store" or "modify" in many computer systems. Table 8-2 lists the ACF keys and their functions.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

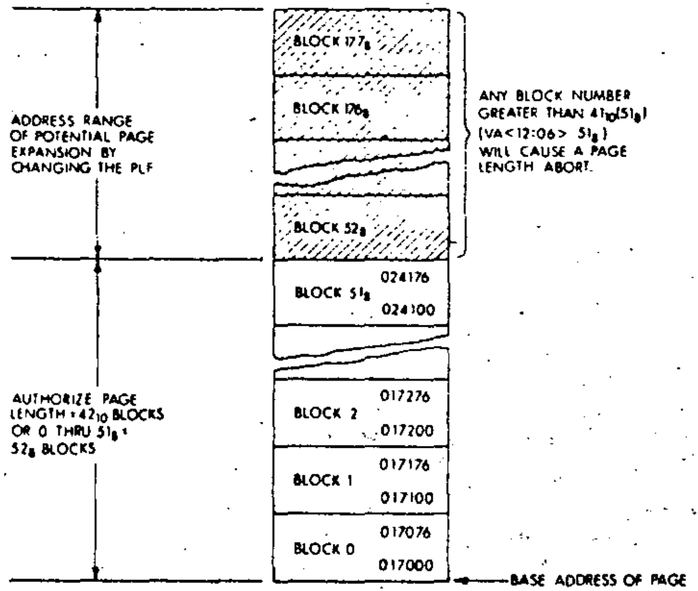

BLOCK 177₈
BLOCK 176₈
BLOCK 52₈
BLOCK 51₈
024176
024100
BLOCK 2
017276
017200
BLOCK 1
017176
017100
BLOCK 0
017076
017000
ADDRESS RANGE OF POTENTIAL PAGE EXPANSION BY CHANGING THE PLF
AUTHORIZE PAGE LENGTH = 42₁₀ BLOCKS OR 0 THRU 51₈ = 52₈ BLOCKS
ANY BLOCK NUMBER GREATER THAN 41₁₀(51₈) (VA<12:06> 51₈) WILL CAUSE A PAGE LENGTH ABORT.
BASE ADDRESS OF PAGE

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.

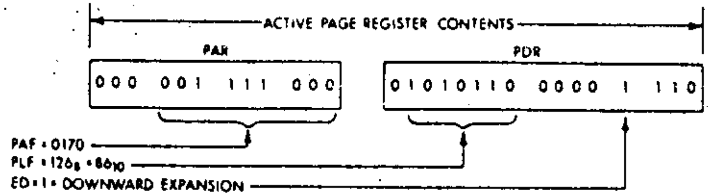
PLF for an Upward Expandable Page
When the page expands upward, the PLF must be set to one less than the intended number of blocks authorized for that page. For example, if 52_8 (42_{10}) blocks are authorized, the PLF is set to 51_8 (41_{10}) (Figure 8-6). The hardware compares the virtual address block 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 $= 126_8$.

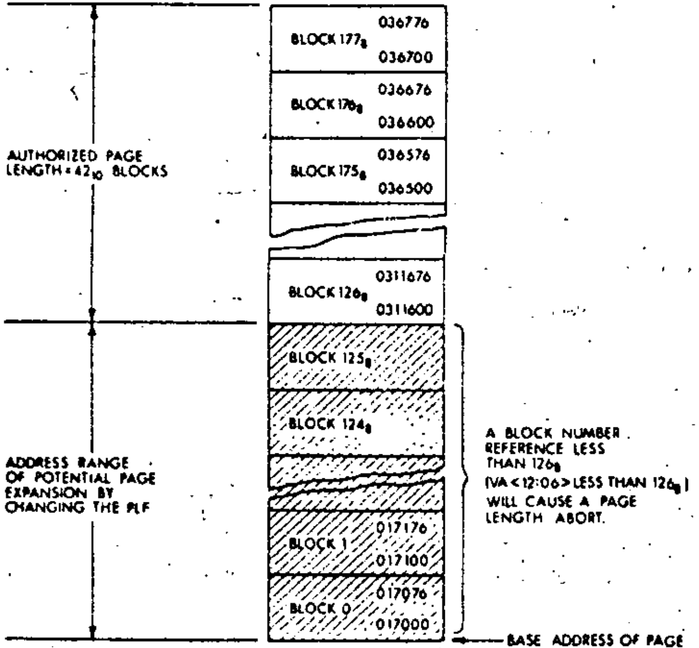

BLOCK 177₈
036776
036700
BLOCK 176₈
036676
036600
BLOCK 175₈
036576
036500
BLOCK 126₈
0311676
0311600
BLOCK 125₈
BLOCK 124₈
BLOCK 1
017176
017100
BLOCK 0
017076
017000
AUTHORIZED PAGE LENGTH = 42₁₀ BLOCKS
ADDRESS RANGE OF POTENTIAL PAGE EXPANSION BY CHANGING THE PLF
A BLOCK NUMBER REFERENCE LESS THAN 126₈ (VA<12:06> LESS THAN 126₈) WILL CAUSE A PAGE LENGTH ABORT.
BASE ADDRESS OF PAGE

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.

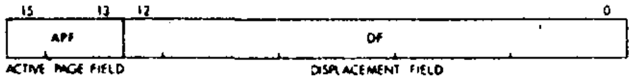


Figure 8-8 Interpretation of a Virtual Address

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.

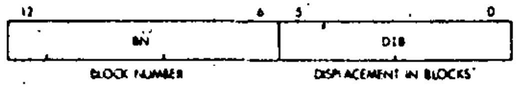


Figure 8-9 Displacement Field of Virtual Address

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 × 32 = 96) words in physical memory.

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

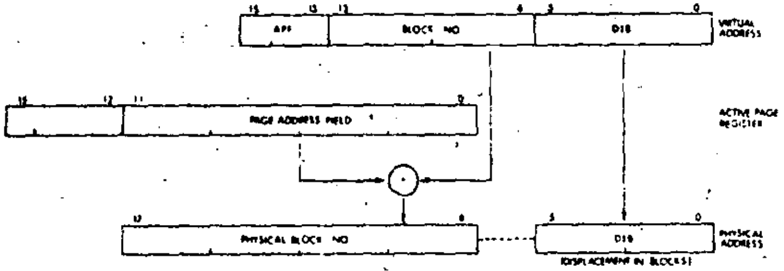


Figure 8-10 Construction of a Physical Address

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.

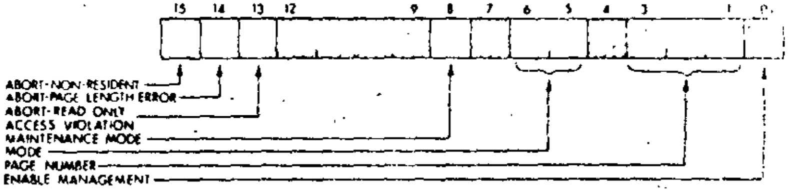


Figure 8-11 Format of Status Register #0 (SR0)

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.

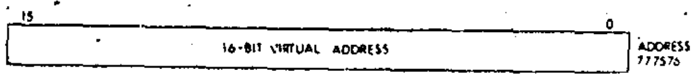


Figure 8-12 Format of Status Register 2 (SR2))

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

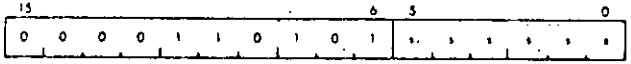
These instructions are directly compatible with the larger 11 computers.

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.

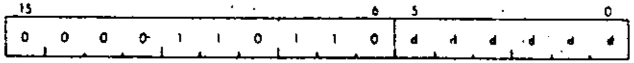
Example: MFPI @ (R2) R2 = 1000
1000 = 37526

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   @(6)+                ; 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
TA6X:   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=O (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 operations can be made if the Processor Status is mapped in User space.

MINICOMPUTERS FOR ENGINEERS AND SCIENTISTS.

G. A. KORN.

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          Comments
                   (main program)

   .                  .
   .                  .
                      .
1713               current instruction              / Interrupt occurs here

0000               1714                             / Incremented program
                                                    /   counter (1714) will be
                                                    /   stored here by hard-
                                                    /   ware
0001               JUMP TO SRVICE                   / Trap location, contains
                                                    /   jump to relocatable
3600      SRVICE   STORE ACCUMULATOR IN   SAVAC     /   service routine
```

```
600   SRVICE   STORE ACCUMULATOR IN   SAVAC   / Save accumulator
601            READ ADC                       / Read ADC into
                                              /   accumulator and
                                              /   clear ADC flag
602            STORE ACCUMULATOR IN   X       / Store ADC reading
603            LOAD ACCUMULATOR       SAVAC   / Restore accumulator
604            INTERRUPT ON                   / Turn interrupt back on
605            JUMP INDIRECT VIA      0000    / Return jump

714            (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- pri
                                               /   error message
ADC      STORE ACCUMULATOR IN   SAVAC          / ADC service routin
         READ ADC
         STORE ACCUMULATOR IN   X
         LOAD ACCUMULATOR       SAVAC          / Restore accumulat
         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.

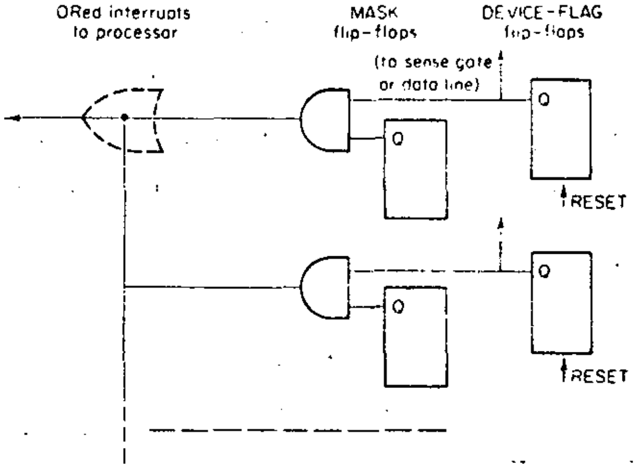


Fig. 5-9. Interrupt masking. The mask flip-flops are treated as a control register (*mask register*), which can be cleared and loaded by I/O instructions.

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

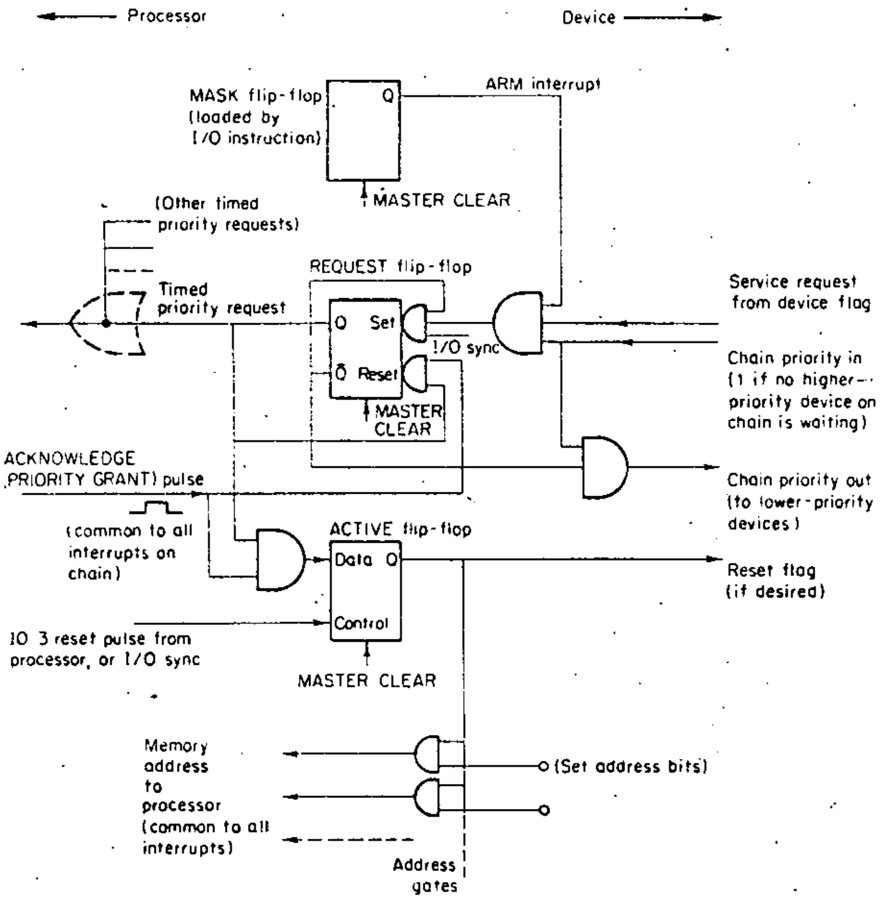

Processor
Device
MASK flip-flop (loaded by I/O instruction)
Q
ARM interrupt
MASTER CLEAR
(Other timed priority requests)
Timed priority request
REQUEST flip-flop
Q
Set
Q̄
Reset
MASTER CLEAR
I/O sync
Service request from device flag
Chain priority in (1 if no higher-priority device on chain is waiting)
ACKNOWLEDGE (PRIORITY GRANT) pulse
(common to all interrupts on chain)
Chain priority out (to lower-priority devices)
ACTIVE flip-flop
Data Q
Control
Reset flag (if desired)
IO 3 reset pulse from processor, or I/O sync
MASTER CLEAR
Memory address to processor (common to all interrupts)
(Set address bits)
Address gates

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

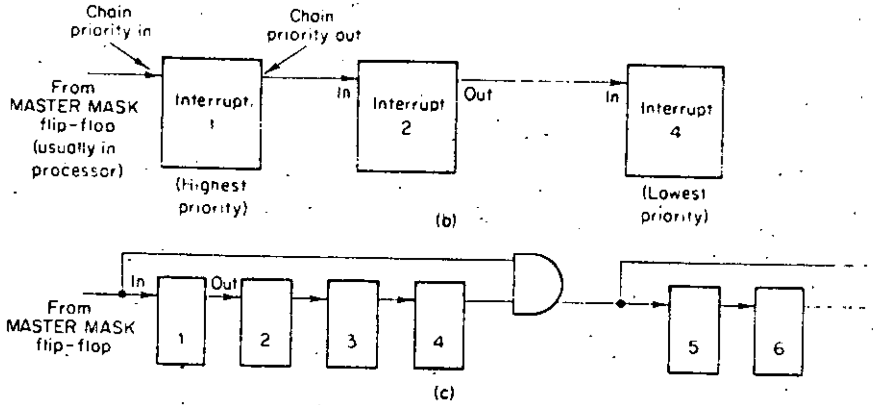


Fig. 5-10*b* and *c*. Wired-chain priority-propagation circuits. Since each subsystem (and its associated wiring) delays the propagated REQUEST flip-flop steps (Fig. 5-10*a*) by 10 to 30 nsec, the simple chain of Fig. 5-10*b* should not have more than four to six links; the circuit of Fig. 5-10*c* bypasses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

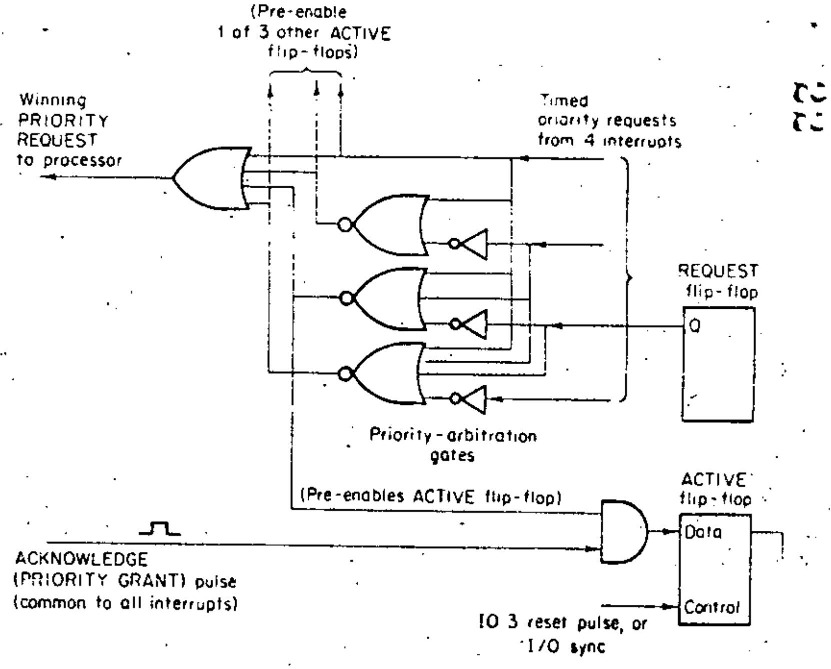

(Pre-enable 1 of 3 other ACTIVE flip-flops)
Winning PRIORITY REQUEST to processor
Timed priority requests from 4 interrupts
REQUEST flip-flop
Q
Priority-arbitration gates
(Pre-enables ACTIVE flip-flop)
ACTIVE flip-flop
Data
Control
ACKNOWLEDGE (PRIORITY GRANT) pulse (common to all interrupts)
IO 3 reset pulse, or I/O sync

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

device (usually a power-failure, parity-error, or real-time-clock interrupt in the processor itself) connects to a processor flip-flop ("master-mask" flip-flop), which can thus arm or disarm the entire chain (Fig. 5-10*b* and *c*).

The computer program can load mask-register flip-flops (Fig. 5-10*a*) to *disarm* selected interrupts in such a wired chain, but the relative priorities of all armed interrupts are determined by their positions in the chain. It is possible, though, to assign two or more different priorities to a given device flag: we connect it to two or more separate priority circuits in the chain and arm one of them under program or device control.

Figure 5-11 illustrates the second type of priority-propagation logic, which permits every armed interrupt to set its REQUEST flip-flop. The timed PRIORITY REQUEST steps from different interrupts are combined in a **"priority-arbitration" gate circuit,** which lets only the highest-priority REQUEST step pass to preenable its "ACTIVE" flip-flop. Some larger digital computers implement dynamic priority reallocation by modifying their priority-arbitration logic under program control, but most minicomputers are content with programmed masking.

The two priority-propagation schemes can be *combined.* Several minicomputer systems (e.g., PDP-9, PDP-15) employ four separate wired-priority chains, each armed or disarmed by a common "master-mask" flip-flop in the processor. Interrupts from the four chains are combined through a priority-arbitration network which, together with the program-controlled "master-mask" flip-flops, establishes the relative priorities of the four chains.

5-15. Complete Priority-interrupt Systems. (a) Program-controlled Address Transfer. The "ACTIVE" flip-flop in Fig. 5-10*a* or 5-11 places the starting address of the correct interrupt-service routine on a set of address lines common to all interrupts. Automatic or "hardware" priority-interrupt systems will then immediately trap to the desired address (Sec. 5-15*b*). But in many small computers (e.g., PDP-8 series, SUPERNOVA), the priority logic is only an add-on card for a basic single-level (ORed) interrupt system. Such systems cannot access different trap addresses directly. With the interrupt system on, *every* PRIORITY REQUEST disables further interrupts and causes the program to trap to *the same* memory location, say 0000, and to store the program counter, just as in Sec. 5-9. The trap location contains a jump to the service routine

```
SRVICE   STORE ACCUMULATOR IN      SAVAC   / Unless we have
                                           /  a spare
                                           /  accumulator

         READ INTERRUPT ADDRESS
         STORE ACCUMULATOR IN      PTR
         JUMP INDIRECT VIA         PTR
```

READ INTERRUPT ADDRESS is an ordinary I/O instruction, which employs a device selector to read the interrupt-address lines into the accumulator (Sec. 5-9). The IO2 pulse from the device selector can serve as the ACKNOWLEDGE pulse in Fig. 5-10*a* or 5-11 (in fact, the "ACTIVE" flip-flop can be omitted in this simple system). The program then transfers the address word to a pointer location PTR in memory, and an indirect jump lands us where we want to be.

Unfortunately, the service routine for each individual device, say for an ADC, must save and restore program counter, mask, *and* accumulator (see also Sec. 5-12):

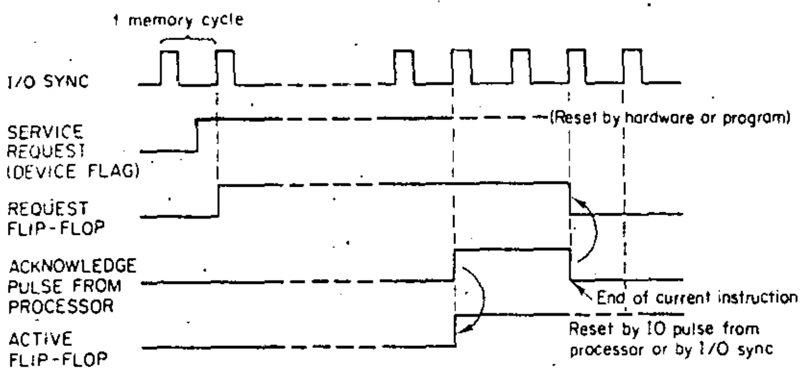
```
ADC   LOAD ACCUMULATOR          0000
      STORE ACCUMULATOR IN      SAVPC
      LOAD ACCUMULATOR          SAVAC
      STORE ACCUMULATOR IN      SAVAC2
      LOAD ACCUMULATOR          MASK
      STORE ACCUMULATOR IN      SVMSK
      LOAD ACCUMULATOR          MASK 1
      STORE ACCUMULATOR         MASK
      LOAD MASK REGISTER
      INTERRUPT ON
      READ ADC                                  / Useful work
      STORE ACCUMULATOR IN      X               /   done only here
      INTERRUPT OFF
      LOAD ACCUMULATOR          SVMSK
      STORE ACCUMULATOR         MASK
      LOAD MASK REGISTER
      LOAD ACCUMULATOR          SAVAC 2
      INTERRUPT ON
      JUMP INDIRECT VIA         SAVPC
```

23

Note that most of the time and memory used up by this routine is **overhead** devoted to storing and saving registers.

(b) A Fully Automatic ("Hardware") Priority-interrupt System. In an **automatic or "hardware" priority-interrupt system,** the "ACTIVE" flip-flop in Fig. 5-10*a* or 5-11 gates the trap address of the active interrupt into the processor memory address register as soon as the current instruction is completed (Fig. 5-12). This requires special address lines in the input/output bus and a little extra processor logic. This hardware buys improved response time and simplifies programming:

1. **The program traps immediately to a different trap location for each interrupt;** there is no need for the program to identify the interrupt.
2. There is no need to save program counter and registers twice as in Secs. 5-11, 5-12, and 5-15*a*.


1 memory cycle
I/O SYNC
SERVICE REQUEST (DEVICE FLAG)
(Reset by hardware or program)
REQUEST FLIP-FLOP
ACKNOWLEDGE PULSE FROM PROCESSOR
End of current instruction
ACTIVE FLIP-FLOP
Reset by IO pulse from processor or by I/O sync

NOTE: Worst case is shown. ACKNOWLEDGE pulse could end 1 cycle earlier.

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

In a typical system, each hardware-designated trap location is loaded with a modified JUMP AND SAVE instruction (Sec. 2-11). Its effective address, say SRVICE, will store the interrupt return address (plus some status bits); this is followed by the interrupt-service routine, which can be relocatable:

```
SRVICE  XXXX                                  / Incremented program-
                                              /   counter reading
                                              /   (return address)
                                              /   saved here
        STORE ACCUMULATOR IN   SAVAC          / Save accumulator

        LOAD ACCUMULATOR       MASK           / Save current
        STORE ACCUMULATOR IN   SVMSK          /   mask
        LOAD ACCUMULATOR       MASK 1         / Get
        STORE ACCUMULATOR IN   MASK           /   new
        LOAD MASK REGISTER                    /   mask
        INTERRUPT ON
        READ ADC                              / Actual work begins here
        .
        .
        .
```

Saving (and later restoring) the interrupt mask in this program is the same as in Secs. 5-12 and 5-15*a* and is seen to be quite a cumbersome operation. A little extra processor hardware can simplify this job:

1. We can combine the LOAD MASK REGISTER and INTERRUPTION instructions into a single I/O instruction.

2. We can use only masks disarming *all* interrupts with priorities *below* level 1, 2, 3, ... Such simple masks are easier to store automatically.

In the more sophisticated interrupt systems, the interrupt return-jump instruction is replaced by a special instruction (RETURN FROM INTERRUPT), which automatically restores the program-counter reading and all automatically saved registers. Be sure to consult the interface manual for your own minicomputer to determine which hardware features and software techniques are available.

5-16. Discussion of Interrupt-system Features and Applications. Interrupts are the basic mechanism for sharing a digital computer between different, often time-critical, tasks. The practical effectiveness of a minicomputer interrupt system will depend on:

1. **The time needed to service possibly critical situations**
2. **The total time and program overhead** imposed by saving, restoring, and masking operations associated with interrupts
3. The number of priority levels needed versus the number which can be readily implemented
4. Programming flexibility and convenience

The minimum time needed to obtain service will include:

1. The "raw" **latency time,** i.e., the time needed to complete the longest possible processor instruction (including any indirect addressing); most minicomputers are also designed so that the processor will always execute the instruction following any I/O READ or SENSE/SKIP instruction. We are sure you will be able to tell why! Check **your interface manual.**
2. The time needed for any necessary saving and/or masking operation.

A look at the interrupt-service programs of Secs. 5-11, 5-12, 5-15*a*, and 5-15*b* will illustrate how successively more sophisticated priority-interrupt systems provide faster service with less overhead. You should, however, take a hard-nosed attitude to establish whether you really need the more advanced features in your specific application.

It is useful at this point to list the principal applications of interrupts. Many interrupts are associated with I/O routines for relatively slow devices such as teletypewriters and tape reader/punches, and thousands of minicomputers service these happily with simple skip-chain systems. Things become more critical in instrumentation and control systems, which must not miss real-time-clock interrupts intended to log time, to read instruments, or to perform control operations. Time-critical jobs require *fast responses*. If there are many time-critical operations or any time-sharing computations, *the computing time wasted in overhead operations* becomes interesting. Some real-time systems may have periods of peak loads when it becomes actually impossible to service *all* interrupt requests. At this point, the designer must decide whether to buy an improved system or which interrupt requests are at least temporarily expendable. It is in the latter connection that *dynamic priority allocation* becomes useful: it may, for instance, be expedient *to mask certain interrupts during peak-load periods.* In other situations we might, instead, *lower the relative priority of the main computer program by unmasking additional interrupts during peak real-time loads.*

If two or more interrupt-service routines employ the same library subroutine, we are faced, as in Sec. 4-16, with the problem of *reentrant programming.* Temporary-storage locations used by the common subroutine may be wiped out unless we either duplicate the subroutine program in memory for each interrupt or unless we provide true reentrant subroutines. This is not usually the case for FORTRAN-compiler-supplied library routines. Only a few minicomputer manufacturers and software houses provide reentrant FORTRAN (sometimes called "real-time" FORTRAN). The best way to store saved registers and temporary intermediate results is in a stack (Sec. 4-16); a stack pointer is advanced whenever a new interrupt is recognized and retracted when an interrupt is dismissed. *The best minicomputer interrupt systems have hardware for automatically advancing and retracting such a stack pointer* (Sec. 6-10).

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

In conclusion, remember that the chief purpose of interrupt systems is to initiate computer operations more complicated than simple data transfers. The best method for time-critical reading and writing as such is not through interrupt-service routines with their awkward programming overhead but with a *direct-memory-access system*, which has no such problems at all.

DIRECT MEMORY ACCESS AND AUTOMATIC BLOCK TRANSFERS

5-17. Cycle Stealing. Step-by-step program-controlled data transfers limit data-transmission rates and use valuable processor time for alternate instruction fetches and execution; programming is also tedious. It is often preferable to use additional hardware for interfacing a parallel data bus directly with the digital-computer memory data register and to request and grant 1-cycle pauses in processor operation for **direct transfer of data to or from memory (interlace or cycle-stealing operation).** In larger digital machines, and optionally in a few minicomputers (PDP-15), a data bus can even access one memory bank without stopping processor interaction with other memory banks at all.

Note that **cycle stealing in no way disturbs the program sequence.** Even though smaller digital computers must stop computation during memory transfers, the program simply skips a cycle at the end of the current memory cycle (no need to complete the current *instruction*) and later resumes just where it left off. One does not have to save register contents or other information, as with program interrupts.

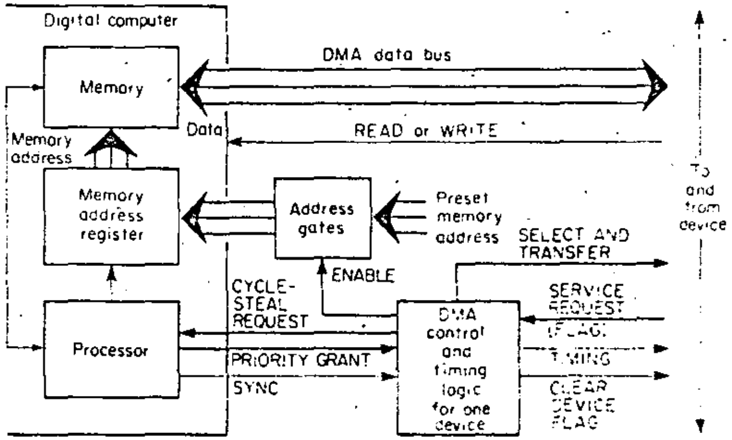


Fig. 5-13. A direct-memory-access (DMA) interface.

5-18. DMA Interface Logic. To make **direct memory access (DMA)** practical, the interface must be able to:

1. **Address desired locations in memory**
2. Synchronize cycle stealing with processor operation
3. **Initiate transfers** by device requests (this includes clock-timed transfers) or by the computer program
4. Deal with **priorities** and queuing of service requests if two or more devices request data transfers

DMA priority/queuing logic is essentially the same as the priority-interrupt logic of Figs. 5-10 and 5-11; indeed, identical logic cards often serve both purposes. **DMA service requests are always given priority over concurrent** *interrupt requests.*

Just as in Fig. 5-11, a **DMA service request** (caused by a device-flag level) produces a **cycle-steal request** unless it is inhibited by a higher-priority request; the processor answers with an **acknowledge (priority-grant) pulse.** This signal then sets a processor-clocked "ACTIVE" flip-flop, which strobes a suitable **memory address** into the processor memory address register and then causes memory and device logic to transfer data from or to the DMA data bus (Fig. 5-13).

In some computer systems (e.g., Digital Equipment Corporation PDP-15), the DMA data lines are identical with the programmed-transfer data lines. This simplifies interconnections at the expense of processor hardware. In other systems, the DMA data lines are also used to transmit the DMA address to the processor before data are transferred. This further reduces the number of bus lines, but complicates hardware and timing.

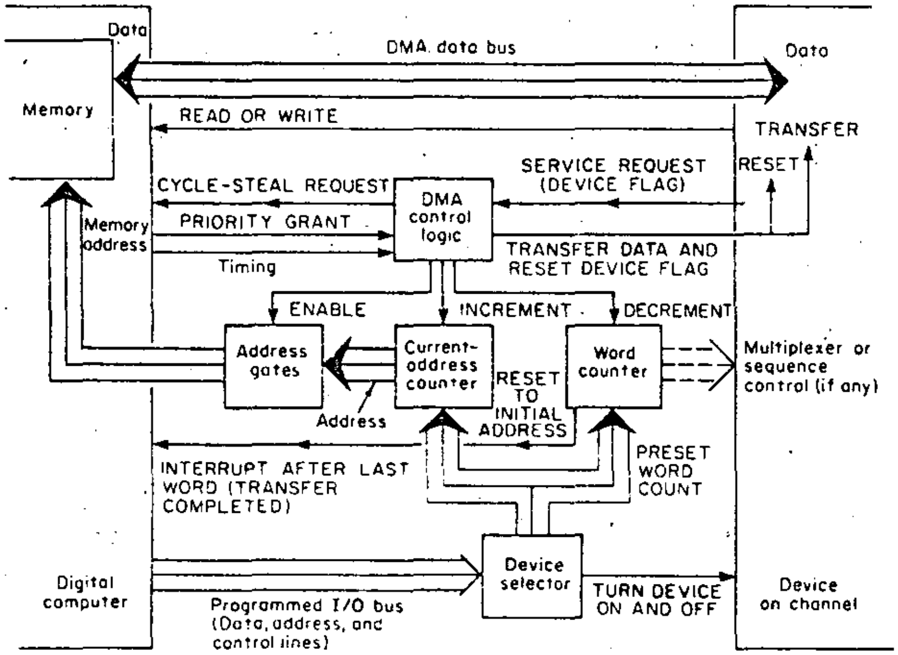


Fig. 5-14. A simple data channel for automatic block transfers.

5-19. Automatic Block Transfers. As we described it, the DMA data transfer is *device*-initiated. A *program-dependent* decision to transfer data, even directly from or to memory, still requires a programmed instruction to cause a DMA service request. This is hardly worth the trouble for a *single-word* transfer. Most DMA transfers, whether device or program initiated, involve not single words but **blocks** of tens, hundreds, or even thousands of data words.

Figure 5-14 shows how the simple DMA system of Fig. 5-13 may be expanded into an **automatic data channel** for block transfers. Data for a block can arrive or depart asynchronously, and the DMA controller will steal cycles as needed and permit the program to go on between cycles. A block of words to be transferred will, in general, occupy a corresponding block of adjacent memory registers. Successive memory addresses can be gated into the memory address registered by a counter, the **current-address counter.** Before any data transfer takes place, a programmed instruction sets the current-address counter to the desired initial address; the desired number of words (**block length**) is set into a second counter, the **word counter,** which will count down with each data transfer until 0 is reached after the desired number of transfers. As service requests arrive from, say, an analog-to-digital converter or data link, the DMA control logic implements successive cycle-steal requests and gates successive current addresses into the memory address register as the current-address counter counts up (see also Fig. 5-5*a*).

The word counter is similarly decremented once per data word. When a block transfer is completed, the word counter can stop the device from requesting further data transfers. The word-counter carry pulse can also cause an *interrupt* so that a new block of data can be processed. The word counter may, if desired, also serve for sequencing device functions (e.g., for selecting successive ADC multiplexer addresses).

Some computers replace the word counter with a program-loaded **final-address register,** whose contents are compared with the current-address counter to determine the end of the block.

A DMA system often involves several data channels, each with a DMA control, address gates, a current-address counter, and a word counter, with different priorities assigned to different channels. For efficient handling of randomly timed requests from multiple devices (and to prevent loss of data words), data-channel systems may incorporate buffer registers in the interface or in devices such as ADCs or DACs.

5-20. Advantages of DMA Systems (see Ref. 6). Direct-memory-access systems can transfer data blocks at very high rates (10^6 words/sec is readily possible) without elaborate I/O programming. The processor essentially deals mainly with buffer areas in its own memory, and only a few I/O instructions are needed to initialize or reinitialize transfers.

Automatic data channels are especially suitable for servicing peripherals with high data rates, such as disks, drums, and fast ADCs and DACs. But fast data transfer with minimal program overhead is extremely valuable in many other applications, especially if there are many devices to be serviced. To indicate the remarkable efficiency of cycle-stealing direct memory access with multiple block-transfer data channels, consider the operation of a training-type digital flight simulator, which solves aircraft and engine equations and services an elaborate cockpit mock-up with many controls and instrument displays. During each 160-msec time increment, the interface not only performs 174 analog-to-digital conversions requiring a total conversion time of 7.7 msec but also 430 digital-to-analog conversions, and handles 540 eight-bit bytes of discrete control information. The actual

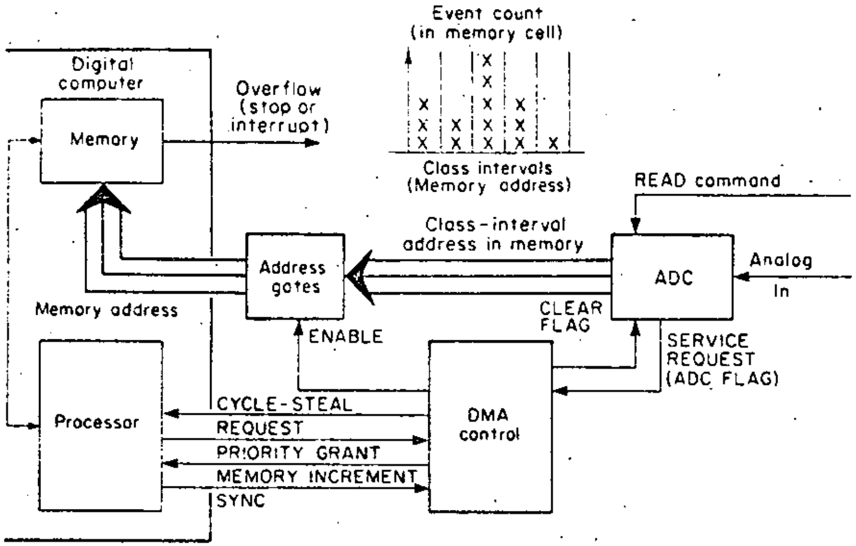


Fig. 5-15a. Memory-increment technique of measuring amplitude distributions (*based on Ref.* 6).

time required to transfer all this information in and out of the data channels is 143 msec per time increment, but because of the fast direct memory transfers, cycle-stealing subtracts only 3.2 msec for each 160 msec of processor time (Ref. 2).

5-21. Memory-increment Technique for Amplitude-distribution Measurements. In many minicomputers, a special pulse input will *increment* the contents of a memory location addressed by the DMA address lines; an interrupt can be generated when one of the memory cells is full. When ADC outputs representing successive samples of a random voltage are applied to the DMA address lines, **the memory-increment feature will effectively generate a model of the input-voltage amplitude distribution in the computer 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

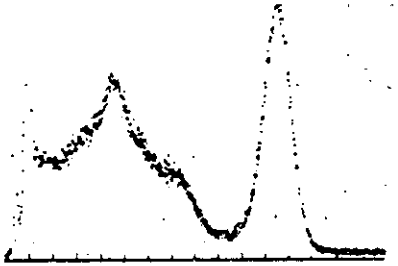


Fig. 5-15b. An amplitude-distribution display obtained by the method of Fig. 5-15*a*. (*Digital Equipment Corporation.*)

$$\bar{X} = \frac{1}{n}\sum_{k=1}^{n} X_k \qquad \overline{X}^2 = \frac{1}{n}\sum_{k=1}^{n} X_k^2 \qquad \cdots$$

are readily computed after the distribution is complete. This technique has been extensively applied to the analysis of pulse-energy spectra from nuclear-physics experiments.

Joint distributions of two random variables X, Y can be similarly compiled. It is only necessary to apply, say, a 12-bit word X, Y composed of two 6-bit bytes corresponding to two ADC outputs X and Y to the memory address register. Now each addressed memory location will correspond to the region $X_i \le X < X_{i+1}$, $Y_k \le Y < Y_{k+1}$ in XY space.

5-22. Add-to-memory Technique of Signal Averaging. Another command-pulse input to some DMA interfaces will *add* a data word on the I/O-bus data lines to the memory location addressed by the DMA address lines without ever bothering the digital-computer arithmetic unit or the program. This "add-to-memory" feature permits useful linear operations on data obtained from various instruments; the only application well known at this time is in **data averaging.**

Figure 5-16*a* and *b* illustrates an especially interesting application of data averaging, which has been very fruitful in biological-data reduction (e.g., electroencephalogram analysis). Periodically applied stimuli produce the same system response after each stimulus so that one obtains an analog waveform periodic with the period T of the applied stimuli. To pull the desired function $X(t)$ out of additive zero-mean random noise, one adds $X(t)$, $X(t + T)$, $X(t + 2T)$, . . . during successive periods to enhance the signal, while the noise will tend to average out. Figure 5-16*c* shows the extraction of a signal from additive noise in successive data-averaging runs.

5-23. Implementing Current-address and Word Counters in the Processor Memory. Some minicomputers (in particular, PDP-9, PDP-15, and the PDP-8 series) have, in addition to their regular DMA facilities, a set of fixed core-memory locations to be used as data-channel address and word counters. Ordinary processor instructions (not I/O instructions) load these locations, respectively, with the block starting address and with minus the block count. The data-channel interface card (Fig. 5-17) supplies the address of one of the four to eight address-counter locations available in the processor; the word counter is the location following the address counter.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

MODOS DE DIRECCIONAMIENTO

1984

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 direccionamiento 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 operación 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 .

Dirección de la sig. inst.	Registro en el procesador	Código de operac.	Dirección primer operando	Dirección segundo operando	Dirección resultado	Palabra n de memoria

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

DIR. DE LA SIG. INST. A EJECUTAR	REG. EN EL PROC.	COD. OP.	DIR. P. OP.	DIR. SEG. OP.	Palabra n de memoria

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

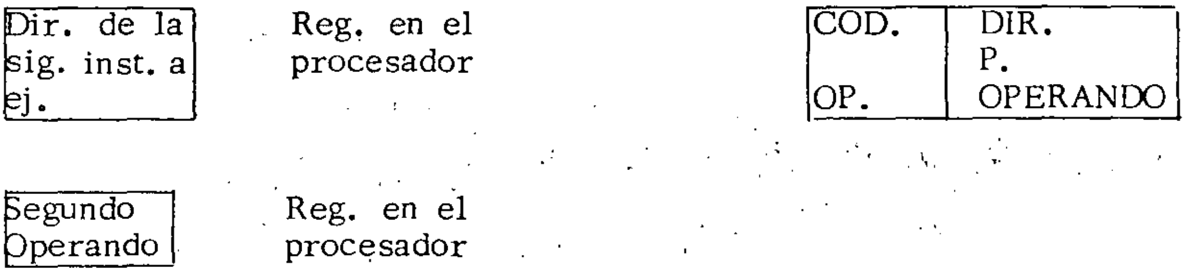


FIG. III.4

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

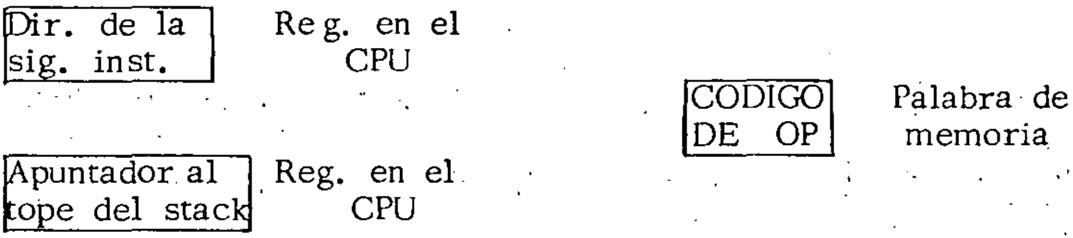


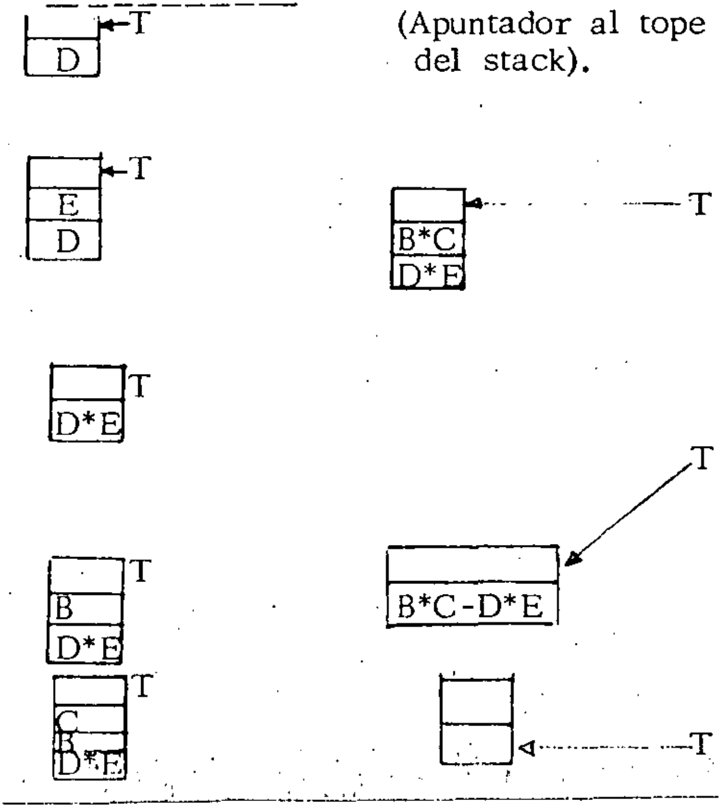
FIG. III.5

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

```
PUSH  D

PUSH  E

MUL

PUSH  B
PUSH  C

MUL
SUB
POP   A
```



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 específica 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

15 12	11 9	8 6	5 3	2 0
	Modo	Registro	Modo	Registro

Código op. dir. fuente dir. destino

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

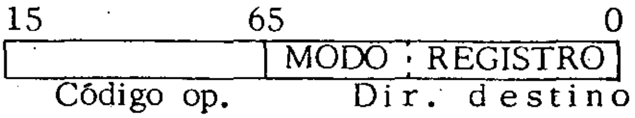


FIG. III.8

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 instrucción, sumada al PC proporcionan la dirección del operando.

Modo: 6 usando R7

Ejemplos: 11

e.4) Relativo diferido

Forma general: OPR @A

Descripción: La localidad de memoria siguiente a la instrucción sumada al PC proporciona el apuntador a la dirección del operando.

Modo: 7 usando R7

Ejemplos: 12

LUIS CORDERO BORBOA

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                          Despues

 001210/005024                  001210/005024
_$4/000010                     _$4/000012
_000010/174216                 _000010/000000
_$7/001210                     _$7/001212
_$S/170021                     _$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                          Despues

 001212/105024                  001212/105024
_$4/000006                     _$4/000007
_000006/173215                 _000006/173000
_$7/001212                     _$7/001214
_$S/170024                     _$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                          Despues

 001214/060022                  001214/060022
_$0/000007                     _$0/000007
_$2/000024                     _$2/000026
_000024/000007                 _000024/000016
_$7/001214                     _$7/001216
_$S/170024                     _$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                  _$7/001220
      _$S/170020                  _$S/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                  _$7/001222
      _$S/170020                  _$S/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                  _$7/001224
      _$S/170000                  _$S/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

```
 001250/005155
_$5/000040
_000036/000020
_000020/000000
_$7/001250
_$S/170020
_
```

Despues

```
 001250/005155
_$5/000036
_000036/000020
_000020/177777
_$7/001252
_$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 RO.
                    ;
```

Antes

```
 001252/067300
_001254/000200
_$0/000015
_$3/000010
_000210/000012
_000012/000016
_$7/001252
_$S/170031
_
```

Despues

```
 001252/067300
_001254/000200
_$0/000033
_$3/000010
_000210/000012
_000012/000016
_$7/001256
_$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
      _                              _
```

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

CONJUNTO DE INSTRUCCIONES

1984

PDP-11
04/34/45/55
PROCESSOR
HANDBOOK

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"

Λ = boolean AND

v = boolean OR

⊻ = exclusive OR

~ = boolean not

Reg or R = register

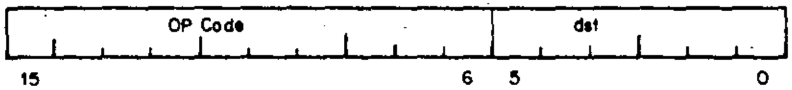
B = Byte

$$■ = \begin{cases} 0 \text{ for word} \\ 1 \text{ for byte} \end{cases}$$

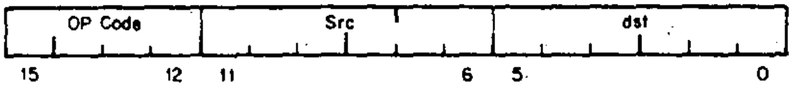
4.2 INSTRUCTION FORMATS

The major instruction formats are:

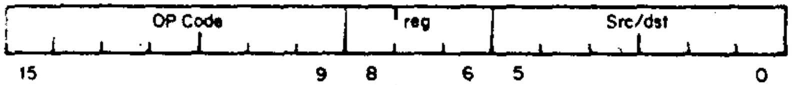
Single Operand Group



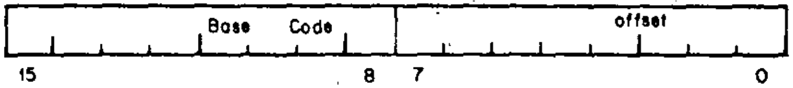
Double Operand Group



Register-Source or Destination

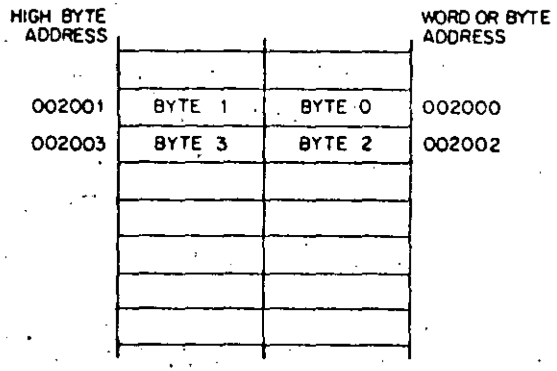


Branch



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:



HIGH BYTE ADDRESS			WORD OR BYTE ADDRESS
002001	BYTE 1	BYTE 0	002000
002003	BYTE 3	BYTE 2	002002

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

4.4 SINGLE OPERAND INSTRUCTIONS

CLR
CLRB

clear destination ■050DD

0/1	0	0	0	1	0	1	0	0	0	d	d	d	d	d	d
15									6	5					0

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

0/1	0	0	0	1	0	1	0	0	1	d	d	d	d	d	d
15									6	5					0

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

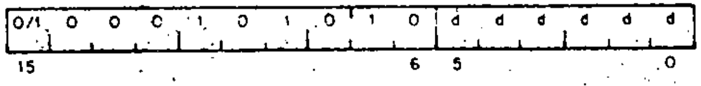
Description: Replaces the contents of the destination address by their logical complement (each bit equal to 0 is set and each bit equal to 1 is cleared)
Byte: Same

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

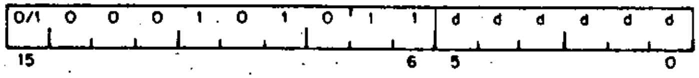
Example: INC R2

Before (R2)=000333 After (R2)=000334

N Z V C 0 0 0 0 N Z V C 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 (R5)=000001 After (R5)=000000

N Z V C 1 0 0 0 N Z V C 0 1 0 0

NEG
NEGB

negate dst ■054DD

0/1	0	0	0	1	0	1	1	0	0	d	d	d	d	d	d
15									6	5					0

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

0/1	0	0	0	1	0	1	1	1	1	d	d	d	d	d	d
15									6	5					0

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

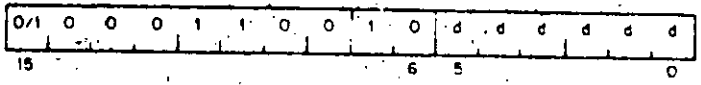
The sign bit (bit 15) of the operand is replicated in shifts to the right. The low order bit is filled with O in shifts to the left. Bits shifted out of the C bit, as shown in the following examples, are lost.

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

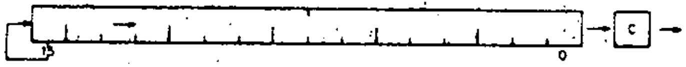


Operation: (dst)←(dst) shifted one place to the right

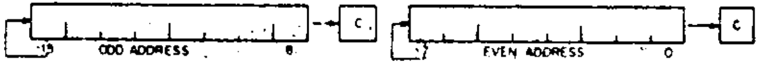
Condition Codes: N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result =0; cleared otherwise
V: loaded from the Exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded from low-order bit of the destination

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:

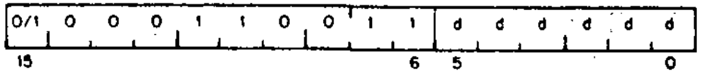


Byte:



ASL
ASLB

arithmetic shift left ■063DD

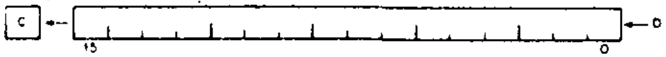


Operation: (dst)←(dst) shifted one place to the left

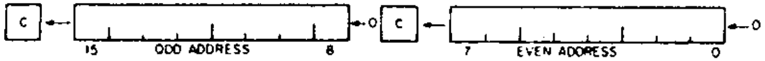
Condition Codes:
N: set if high-order bit of the result is set (result < 0); cleared otherwise
Z: set if the result = 0; cleared otherwise
V: loaded with the exclusive OR of the N-bit and C-bit (as set by the completion of the shift operation)
C: loaded with the high-order bit of the destination

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:

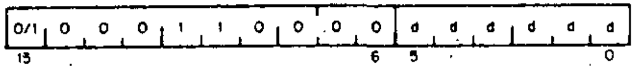


Byte:



ROR
RORB

rotate right ■060DD

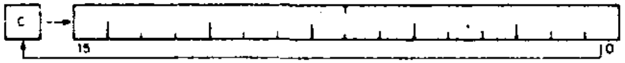


Condition Codes:
N: set if the high-order bit of the result is set (result < 0); cleared otherwise
Z: set if all bits of result = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the low-order bit of the destination

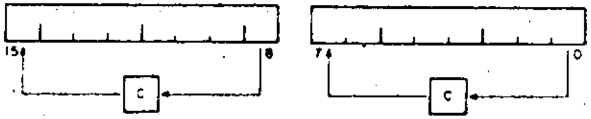
Description: Rotates all bits of the destination right one place. Bit 0 is loaded into the C-bit and the previous contents of the C-bit are loaded into bit 15 of the destination.
Byte: Same

Example:

Word:

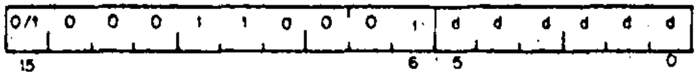


Byte:



ROL
ROLB

rotate left ■061DD

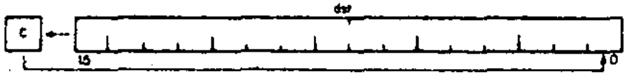


Condition Codes: N: set if the high-order bit of the destination is set (result < 0): cleared otherwise
Z: set if all bits of the destination = 0; cleared otherwise
V: loaded with the Exclusive OR of the N-bit and C-bit (as set by the completion of the rotate operation)
C: loaded with the high-order bit of the destination

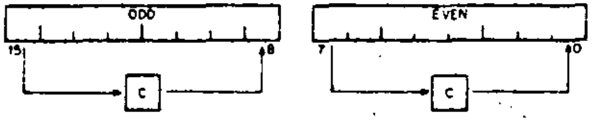
Description: Word: Rotate all bits of the destination left one place. Bit 15 is loaded into the C-bit of the status word and the previous contents of the C-bit are loaded into Bit 0 of the destination.
Byte: Same

Example:

Word:

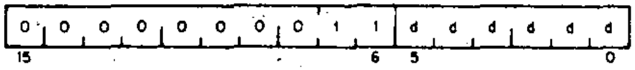


Bytes:



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).

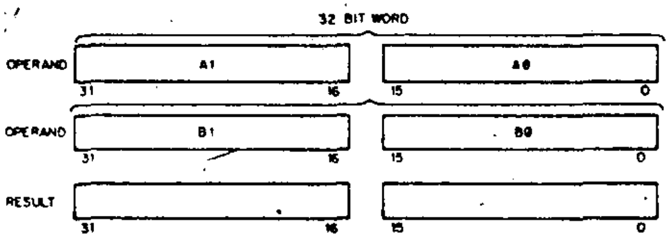
Example: SWAB R1

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:

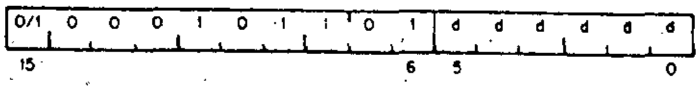
-1 = 37777777777

(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

add carry ■055DD



15									6	5					0
0/1	0	0	0	1	0	1	1	0	1	d	d	d	d	d	d

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

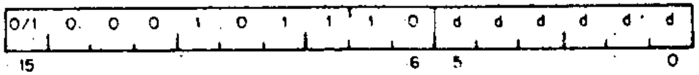
Description: Adds the contents of the C-bit into the destination. This permits the carry from the addition of the low-order words to be carried into the high-order result.
Byte: Same

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

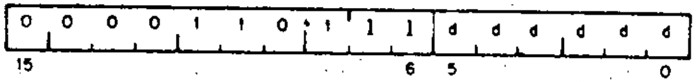
Example: Double precision subtraction is done by:

```
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

1	0	0	0	1	1	0	1	1	1	d	d	d	d	d	d

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

1	0	0	0	1	1	0	1	0	0	s	s	s	s	s	s

Operation: PS <0:7> ← (SRC)

Condition Codes: Set according to effective SRC operand bits 0–3.

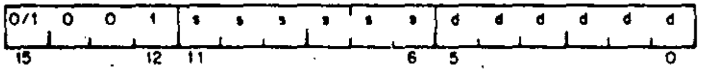
Description: The 8 bits of the effective operand replaces the current contents of the PS <0:7>. The source operand address is treated as a byte address.
Note that the T bit (PS bit 4) cannot be set with this instruction. The SRC operand remains unchanged. This instruction can be used to change the priority 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

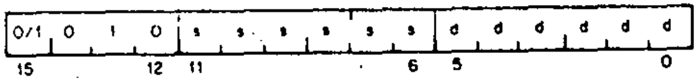
MOV (R6)+,@#177566 ; pops the operand off the stack and moves it into memory location 177566 (terminal print buffer)

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 ■2SSDD



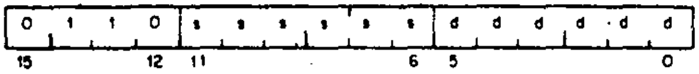
Operation: (src)-(dst)

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.

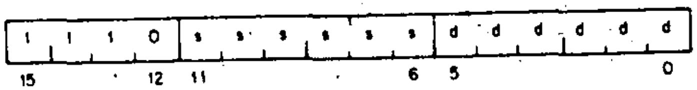
Examples:

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 orignial 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

0/1	0	1	1	s	s	s	s	s	s	d	d	d	d	d	d
15			12	11					6	5					0

Operation: (src) Λ (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
```

$(30)_8$=0 000 000 000 011 000

BIC
BICB

bit clear ■4SSDD

0/1	1	0	0	s	s	s	s	s	s	d	d	d	d	d	d
15			12	11					6	5					0

Operation: (dst)←~(src)Λ(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

16

BIS
BISB

bit set ■5SSDD

0/1	1	0	1	s	s	s	s	s	s	d	d	d	d	d	d
15			12	11					6	5					0

Operation: (dst)←(src) v (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 074RDD

0	1	1	1	1	0	0	r	r	r	d	d	d	d	d	d
15						9	8		6	5					0

Operation: (dst)←R∨(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.

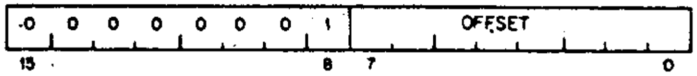
The PDP-11 assembler handles address arithmetic for the user and computes and assembles the proper offset field for branch instructions in the form:

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

15							8	7 … 0
0	0	0	0	0	0	1	0	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

15							8	7 … 0
0	0	0	0	0	0	1	1	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.

19

BPL

branch if plus

100000 Plus offset

1	0	0	0	0	0	0	0	OFFSET	
15							8	7	0

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

1	0	0	0	0	0	0	1	OFFSET	
15							8	7	0

Operation: PC ← PC + (2 x offset) if N = 1

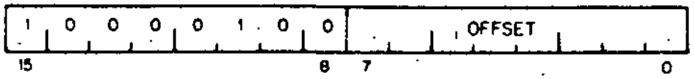
Condition Codes: Unaffected

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.

20

BVC

branch if overflow is clear 102000 Plus offset

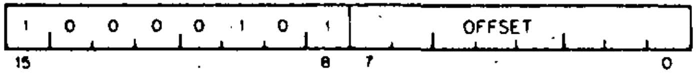


Operation: PC ← PC + (2 x offset) if V=0

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

15							8	7–0
1	0	0	0	0	1	1	0	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

15							8	7–0
1	0	0	0	0	1	1	1	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.

22

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

0	0	0	0	0	1	0	0	OFFSET
15							8	7 0

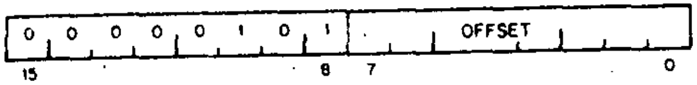
Operation: PC ← PC + (2 x offset) if N ⊻ V = 0

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.

23

BLT

branch if less than (zero) 002400 Plus offset

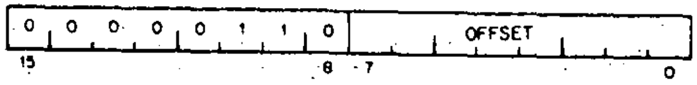


Operation: PC ← PC + (2 x offset) if $N \veebar V = 1$

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).

BGT

branch if greater than (zero) 003000 Plus offset



Operation: PC ← PC + (2 x offset) if $Z \vee (N \veebar V) = 0$

Description: Operation of BGT is similar to BGE, except BGT will not cause a branch on a zero result.

24

BLE

branch if less than or equal (to zero) 003400 Plus offset

15							8	7	0
0	0	0	0	0	1	1	1	OFFSET	

Operation: $PC \leftarrow PC + (2 \times \text{offset})$ if $Z \vee (N \vee V) = 1$

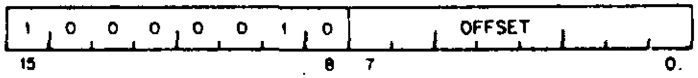
Description: Operation is similar to BLT but in addition will cause a branch if the result of the previous operation was zero.

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

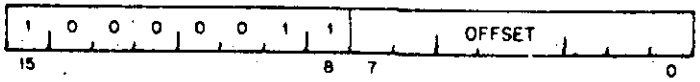


Operation: PC ◄ PC + (2 x offset) if C = 0 and Z = 0

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.

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.

BHIS

branch if higher or same 103000 Plus offset

1	0	0	0	0	1	1	0	OFFSET
15							8	7 0

Operation: PC ← PC + (2 x offset) if C = 0

Description: BHIS is the same instruction as BCC. This mnemonic is included only for convenience.

BLO

branch if lower 103400 Plus offset

1	0	0	0	0	1	1	1	OFFSET
15							8	7 0

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

15									6	5					0
0	0	0	0	0	0	0	0	0	1	d	d	d	d	d	d

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 O. Execution of a jump with mode O 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.

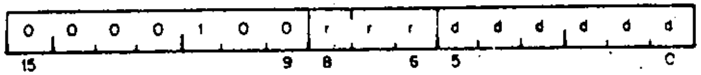
Subroutine Instructions

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.

28

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.

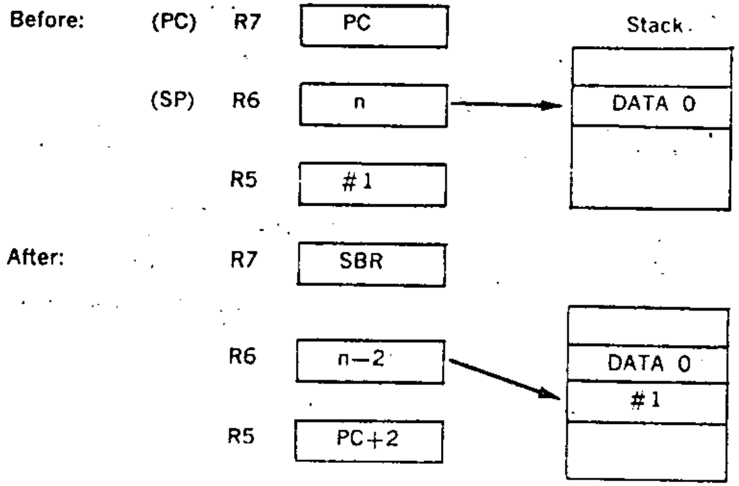
A subroutine called with a JSR reg,dst instruction can access the arguments following the call with either autoincrement addressing, (reg) + , (if arguments are accessed sequentially) or by indexed addressing, X(reg), (if accessed in random order). These addressing modes may also be deferred, @(reg) + 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."

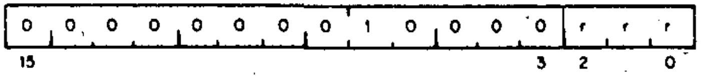
Return from a subroutine is done by the RTS instruction. RTS reg loads the contents of reg into the PC and pops the top element of the processor stack into the specified register.

Example: JSR R5, SBR



RTS

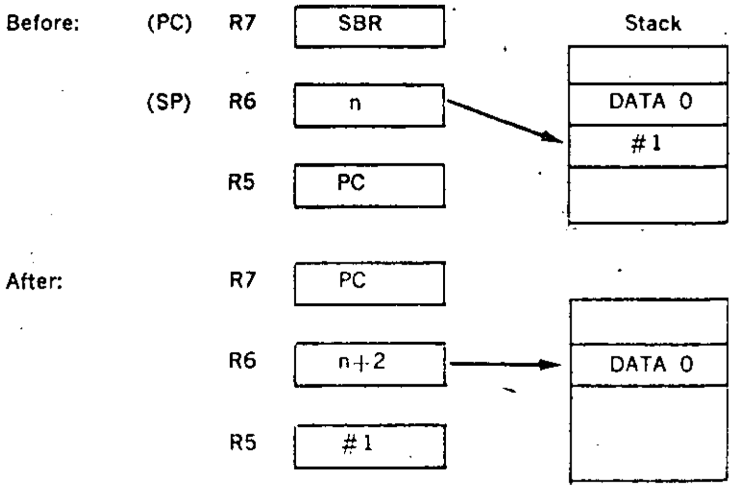
return from subroutine 00020R



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

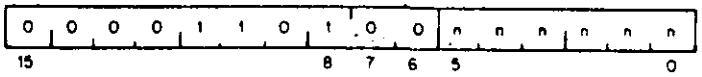
Example: RTS R5



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)↑

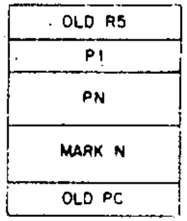
Condition Codes: unaffected

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 it-
                              self



        RTS R5                ;the return begins: this causes
```

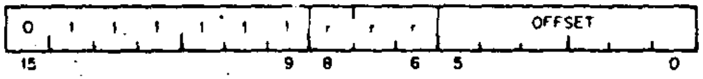
the contents of R5 to be placed in the PC which then results in the execution of the instruction MARK N. The contents of old PC are placed in R5

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 x 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:

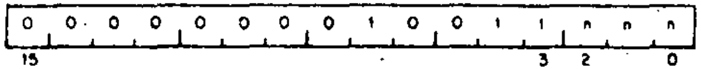
```
SOB    R,A
```

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

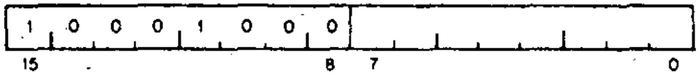
Description The least significant three bits of the instruction are loaded into the Program Status Word (PS) bits 7-5 thus causing a changed priority. The old priority is lost.
Assembler syntax is: SPL N

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

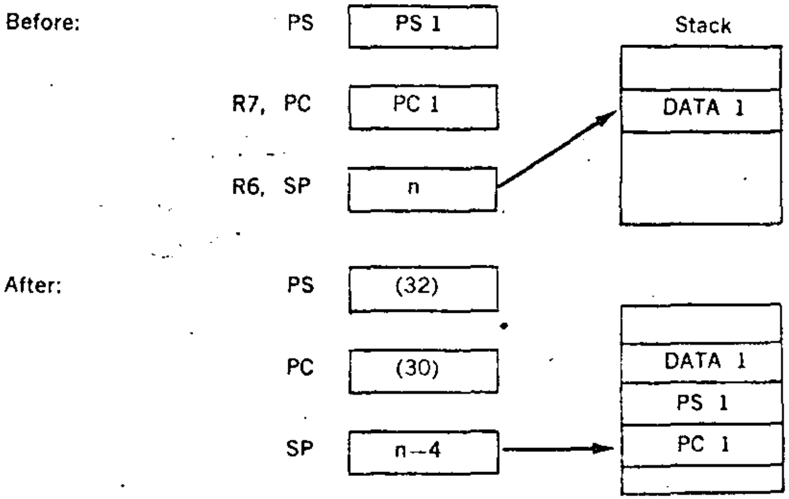


Operation:
↓(SP)←PS
↓(SP)←PC
PC←(30)
PS←(32)

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

15							8	7							0
1	0	0	0	1	0	0	1								

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.

BPT

breakpoint trap 000003

15															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

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.)

IOT

input/output trap 000004

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
15															0

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)

RTI

return from interrupt 000002

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
15															0

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.

34

RTT

Used in the PDP-11/34, 11/45 and 11/55

return from interrupt 000006

15															0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

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: 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

35

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.

36

HALT

halt 000000

15															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

15															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

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.

37

RESET

reset external bus 000005

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
15															0

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

0	0	0	0	0	0	0	0	1	0	1	0/1	N	Z	V	C
15										5	4	3	2	1	0

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

Combinations of the above set or clear operations may be ORed together to form combined instructions.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

MANEJO DE ENTRADA/SALIDA

1984

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 ès innecesaria. La arquitectura de la máquina permite direccionar los registros de estado y datos de los perifé 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 prioridadesdel 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:

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 anteriorres.

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.

Descripción DL11 Teletype Control

Transmisión

Cuando el CPU bus 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.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

MANEJO DE SUBRRUTINAS

1984

MINICOMPUTER SYSTEMS

R.H. ECKHOUSE, JR.

4 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
BR      X
```

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 point to location 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 X(PC) where PC-R7.

5

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 ABS LOC 340
MOV     #340,@#342      ;AND ABS 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 010700 S:       MOV       PC,R0                 ;R0 = (ADDR OF S)+2
000002 062700          ADD       #A-S-2,R0             ;ADD IN OFFSET
       001776
000006 010067          MOV       R0,X                  ;MOVE POINTER TO X
       000766
000012 005077          CLR       @X                    ;CLEAR VALUE INDIRECTLY
       000762
000016 000000          HALT                            ;STOP
                       ;
                       ;
                       ;
       001000          .=.+760
001000 002000 X:       .WORD     A                     ;POINTER TO A
                       ;
                       ;
                       ;
       002000          .=.+776
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

```
        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
       000030
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 ≤ INDEX ≤ maximum value, 1 is subtracted from the index so that its range is 0 ≤ INDEX ≤ 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

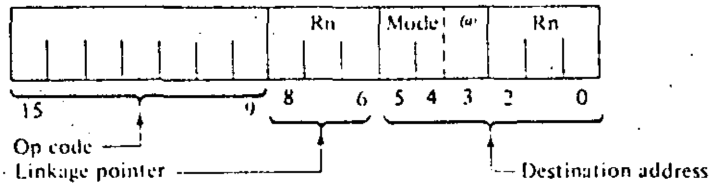
1. Allow the transfer of control from one routine to another.

2. Pass values among the various routines.

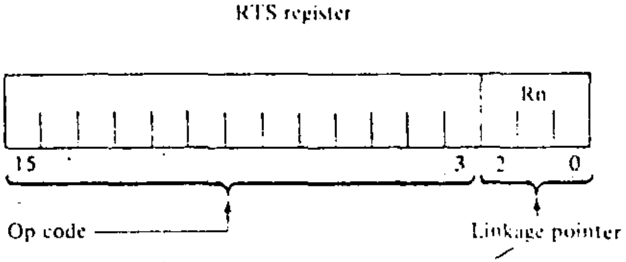
The mechanism that accomplishes these requirements is called the *subroutine linkage* and is, in general, a combination of hardware features and software conventions.

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.

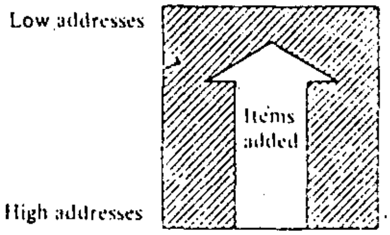


Fig. 4-3 Stack addresses.

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).

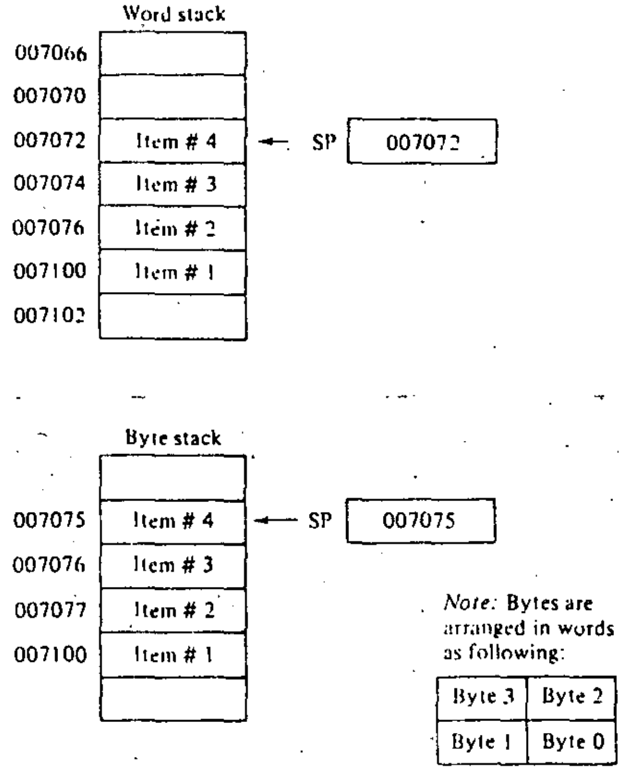


Fig. 4-4 Word and byte stacks.

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.

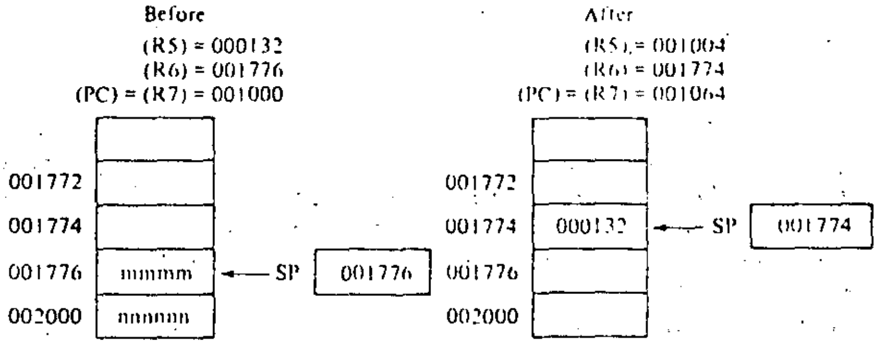


Fig. 4-5 JSR instruction.

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.

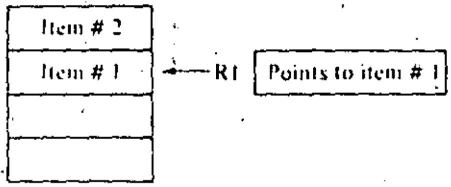


Fig. 4-7 Transmitting stacks as arguments.

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:

17

```
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.

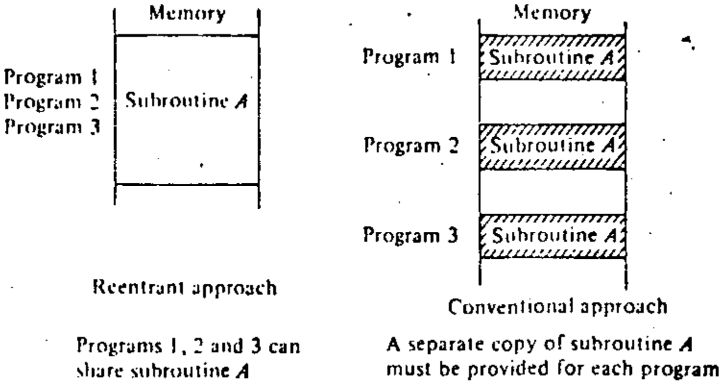


Fig. 4-8 Reentrant routines.

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.

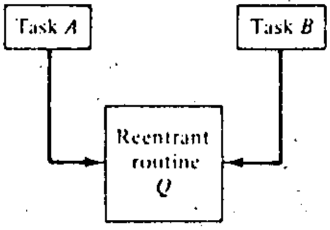


Fig. 4-9 Reentrant routine sharing.

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 RO 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.

Routine # 1 is operating, it then executes:

JSR PC, @ (R6) +

with the following results:

(1) PC2 is popped from the stack and the SP autoincremented

(2) SP is autodecremented and the old PC (i.e., PC1) is pushed

(3) control is transferred to the location PC2 (i.e., routine # 2)

Routine # 2 is operating, it then executes:

JSR PC, @ (R6) +

with the result that PC2 is exchanged for PC1 on the stack and control is transferred back to routine # 1.

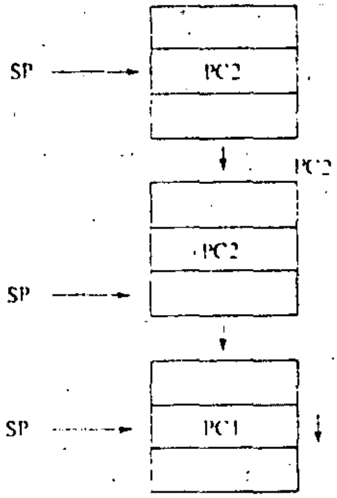


Fig. 4-11 Coroutine interaction.

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.

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

ANEXOS 1

1984

CUANDO SE TIENE QUE PRACTICAR UN CONJUNTO DE INSTRUCCIONES SOBRE DIFERENTES VALORES.

= ACOMODAR UNA LISTA DE VALORES EN ORDEN CRECEINTE O DECRECIENTE, (N! COMPARACIONES)=

Existen dos formas para solucionar esto:

- Copiar el código tantas veces como se necesite
- Agrupar las instrucciones y usar algún mecanismo para llegar a este lugar
- Ejecutar las instrucciones y regresar.

El mecanismo utilizado para brincar el conjunto de instrucciones se le conoce como "llamada" y al conjunto de instrucciones se le conoce como "subrutina".

El mecanismo para manejar subrutinas consiste de dos pasos:

1o. Preservar la dirección de regreso

2o. Cargar al PC con la dirección de la subrutina y se usan dos têcnicas en ayda de esto

- Liga o apuntador (una localidad) *
- Anidación (stack)

En PDP-11 la instrucción que permite el manejo de subrutinas es

```
JSR  R1 , dat
          128 byts
```

```
15    98   65       0
| op  | R1 | dat    |
```

ALGORITMO

1.- Preservar el valor de Registro involucrado

2.- Preservar el PC en el Registro involucrado

3.- Se carga el "PC" con la dirección de la subrutina

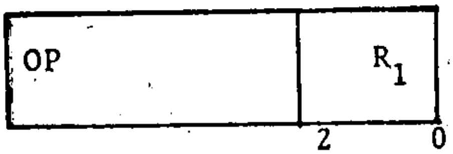
FECH (ALGORITMO) JSR

```
MAR  ←— PC
PC   ←— PC + 2
MDR  ←— MEMORIA [Linf...MAR]     ; OFFSET

TMP  ←— MDR + PC                 ; # DE PALABRAS A SALTAR

R [6]-2                          ; TOP + 2
MAR  ←— R [6]                    ; APUNTA AL SP
MEMORIA [Linf..MAR] ←— R [5]     ; SALVA EL R [5]

R [5] ←— PC                      ; PC SALVADO

PC ←— TMP                        ; DIRECCION
```

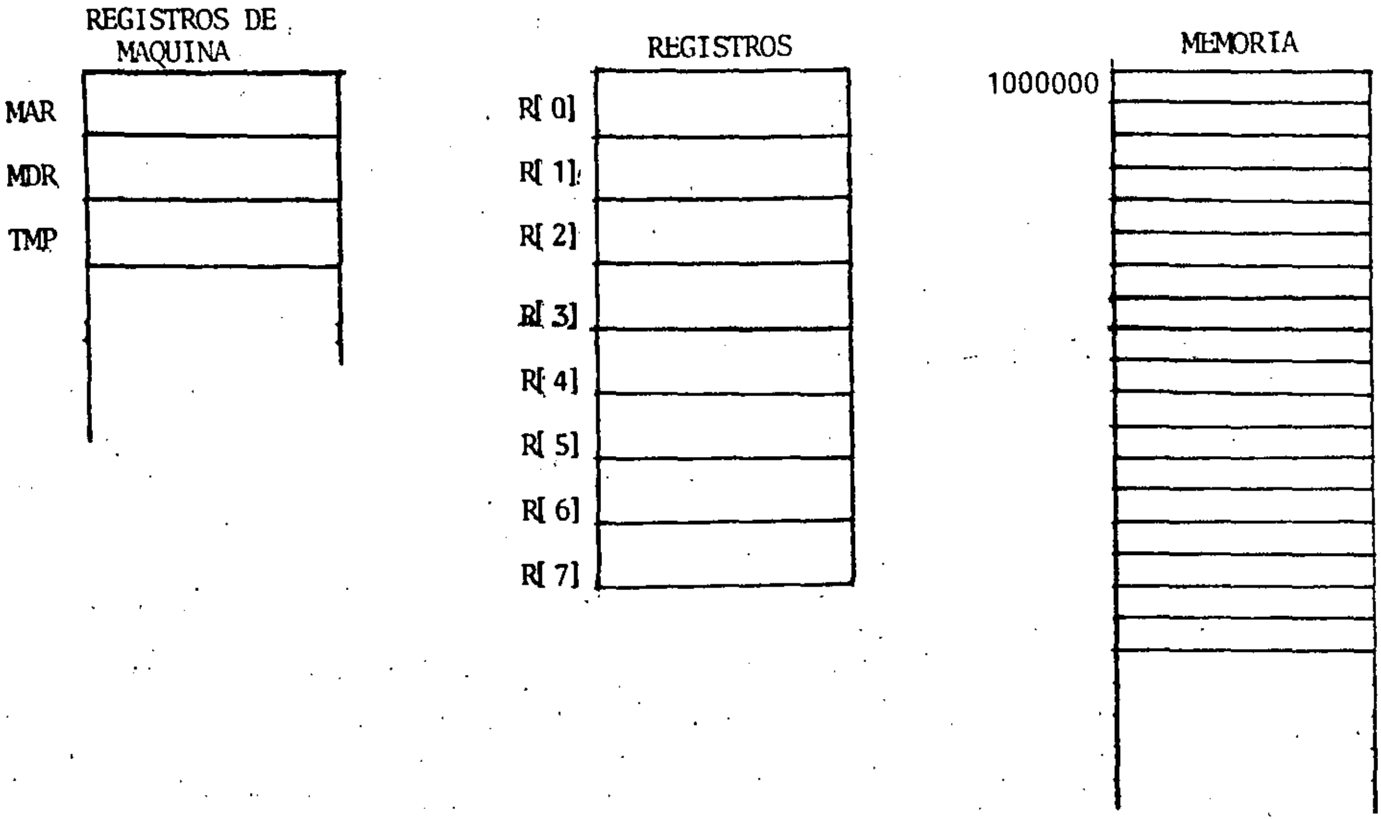
RETURN FROM SUBROUTINES (RTS)

RTS %5 el efecto de esta instrucción es de reemplazar el PC por el contenido de REG [5] y reemplazar REG [5] por el contenido que se encuentra en el TOP del stack.

FECH DE RTS

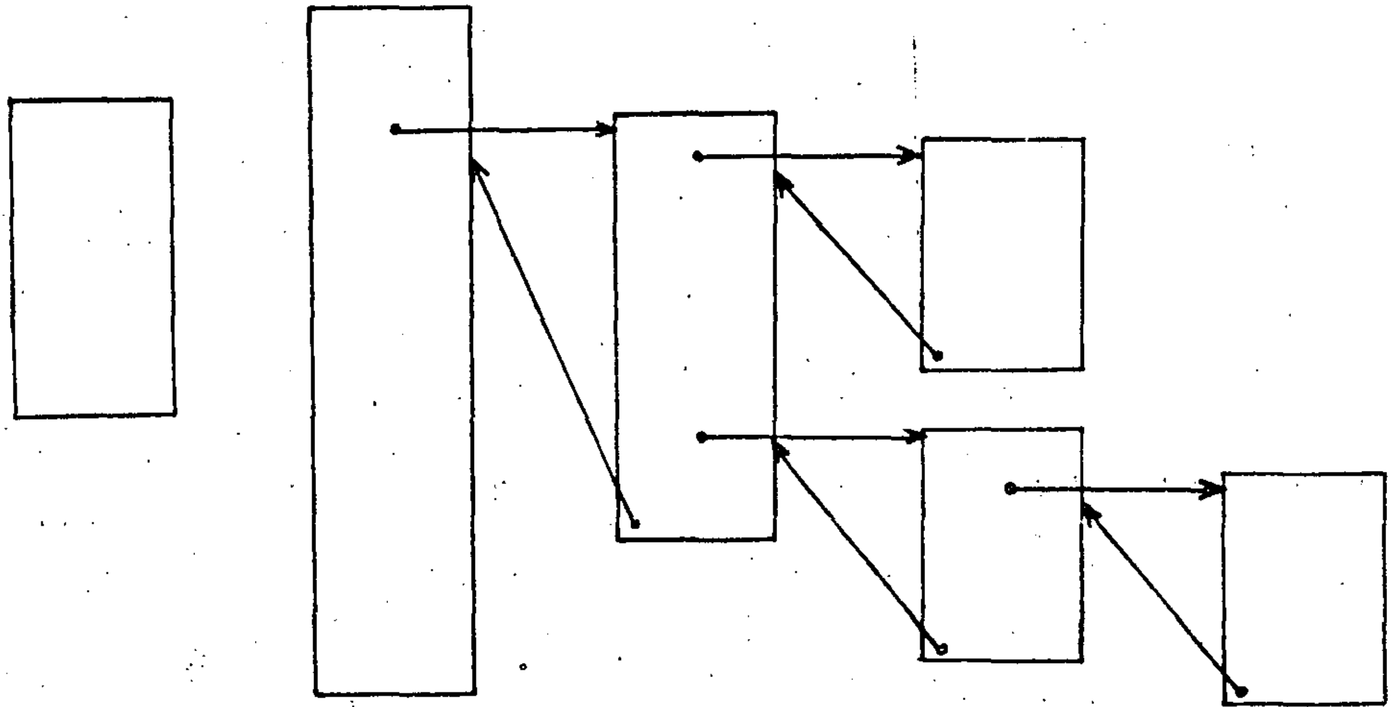


```
PC     <----   R [5]
MAR    <----   R [6]
MDR    <----   MEMORIA [Linf...MAR]     ; RESTAURA EL PC.

R [5]  <----   MDR                      ; RESTAURA R [5]

R [6]  <----   R [6] + 2                ; RESTAURA EL SP
```


REGISTROS DE MAQUINA
MAR
MDR
TMP
REGISTROS
R[0]
R[1]
R[2]
R[3]
R[4]
R[5]
R[6]
R[7]
MEMORIA
1000000

Las instrucciones etiquetadas con B1 y B2 pasa control a la instrucción etiquetada con "SUB", cuando la etiqueta "RETURN" es encontrada, el control regresa a C1 y C2 dependiendo de cuál fue la llamada.

ETIQUETA	CODIGO	OPERANDOS	COMENTARIOS
...	...	...	...
1.- B1:	JER	%5, SUB	LLAMADA A LA SUBRUTINA
2.- C1:	MOV	X, AC	REGRESO
...	...	...	...
3.- B2:	JSR	%5, SUB	LLAMADA A SUB
4.- C2:	MOV	Y, AC	REGRESO
...	...	...	...
5.- SUB:	INC	AC	1era. INSTRUCCION DE SUB
...	...	...	...
RETORNO:	RTS	%5	DE R[5] SE OBTIENE EL REGRESO.



* ENVIAR OPERANDOS A LA SUBRUTINA Y RECIBIR LOS RESULTADOS ES LO QUE SE CONOCE COMO PASAR PARAMETROS.

- Cuando transmitimos parâmetros lo que pretendemos es minimizar el tiempo de ejecución y los requerimientos de memoria.
- Cuatro maneras bâsicas de pasar parâmetros.

1o. AREA DE DATOS COMUN (GLOBAL)

+ P. P. y SB TIENEN ACCESO A ELLA
+ LA DISTANCIA EN EL DIRECCIONAMIENTO (128, -127)

2o. USAR LOS REGISTROS

+ SON POCOS REGISTROS (R_1)
+ USAR MEMORIA PARA PRESERVAL LOS REGISTROS.

```
LABEL          CODE         OPERAND            COMMENTS
----------------------------------------------------------------------------
1. PARAM        =           %1
2.             MOV          PNAME, PARAM       ; REG[PARAM] holds the ad-
                                               ; dress of parameter area.
3.             MOV          ARG1,(PARAM)+      ; Transmit first parameter.
4.             MOV          ARG2,(PARAM)+      ; Second parameter.
   ...         ...          ...                ...
5.             MOV          ARG9,(PARAM)+      ; Last parameter.
6.             JSR          ...                ; Enter subroutine
   ...         ...          ...                ...
7. PNAME;                   PAREA
8. PAREA:.      =           .+18.              ; Parameter area
```

3o. AREA DE PARAMETROS (LA DIR SE PASA EN ALGUN R)

```
LABEL            CODE        OPERAND            COMMENTS
---------------------------------------------------------------------------
1. PARAM         =           %5

2. RET           =           %5

3.               MOV         PNAME, PARAM       ; REG[PARAM] holds the
                                                ; address of parameter
                                                ; area

4.               MOV         ARG1,(PARAM)+      ; Transmit first parameter

5.               MOV         ARG2,(PARAM)+      ; Second parameter

6.               MOV         ARG9,(PARAM)+      ; Last parameter

7.               JSR         RET,Y              ; Call Y with return
                                                ; address in REG[RET].

8. PAREA:        =           .+18.              ; Parameter are follows
                                                  JSR.

9. NEXT:         MOV         Z,T                ; First instruction execu-
                                                  ted
                                                ; after return from Y.

   ...           ...         ...                ...

10. Y:           ...

11.              MOV         4(RET),TEMP        ; Load third parameter
                                                ; into TEMP, REG[RET]
                                                ; contains the startin ad-
                                                  dress of the parameter
                                                ; area, and the third
                                                ; parameter is four bytes
                                                ; beyond the base of the
                                                ; area.

   ...           ...         ...                ...

12.              ADD         EIGHTEEN, RET      ; Calculate actual return
                                                ; address, nine words
                                                ; beyond address in
                                                ; REG[RET].

13.              RTS         RET                ; Exit from Y.

14. PNAME:                   PAREA

15. EIGHTEEN:                18.
```

3.1 AREA DE PARAMETROS EN LINEA (LA DIR BASE AHORA ES LA DIRECCION DE REGRESO)

```
    LABEL         CODE        OPERAND             COMMENTS
-----------------------------------------------------------------------
 1. POINT         =           %6
 2. RET           =           %5
 3.               MOV         ARG2,-(POINT)       ; Push down second param-
                                                  ; eter.
 4.               MOV         ARG1,-(POINT)       ; Push down first parameter.
 5.               JSR         RET,Y               ; Call Y with return address
                                                    in
                  ...                             ; REG[RET].
 6. Y:            MOV         (POINT)+,TEMP       ; Entry to Y. Save old value
                                                    of
                                                  ; REG[RET], now on top of
                                                  ; stack, in TEMP.
    ...           ...         ...                 ...
 7.               MOV         0(POINT),HOLD       ; Load first parameter
 8.               MOV         2(POINT),HOLD1      ; Load second parameter
    ...           ...         ...
 9.               ADD         #4,POINT            ; To return, first pop parame
                                                  ; ters from stack.
10.               MOV         TEMP,-(POINT)       ; Place old value of REG[RET]
                                                  ; on stack.
11.               RTS         RET                 ; Exit from Y.
```

4o. USO DEL STACK (MENOS MEMORIA)

```
        USERMAX=1
        USERMIN=1
EDITOR:
        JSR     PC,INISTR             ;INICIALIZA ESTRUCTURAS
        JSR     PC,ABRCNS             ;ABRIR CANALES:
                                      ;  0   CONSOLA
                                      ;  1-10  TERMINALES
                                      ;  11   ARCHIVO CUENTAS
                                      ;  12   ARCHIVO DIRECTORIA
        JSR     PC,DIGAME             ;DAR UN READ A LAS TERMINALES
        JSR     PC,RDRED              ;DAR RCVD AL FG
LOOP:   .SPND                         ;--A DORMIR--

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
ABRCNS:
        GOSUBS LOOKUP,<#CONSOLA,#CONSLA>        ;ABRIR CANAL A CONSOLA
        GOSUBS LOOKUP,<#ARQCTA,#CUENTA> ;ABRIR ARCHIVO CUENTAS
        GOSUBS LOOKUP,<#DIRARQ,#DIRECT> ;ABRIR DIRECTORIO ARCHIVOS
        MOV     #USERMIN,R2
        DO
             GOSUBS LOOKUP,<R2,#TERMNL>
        UNTIL GT,<<INC R2><CMP R2,#USERMAX>>
        PRINT <<CR><LF>"CANALES ABIERTOS">
        RTS     PC
```

01234567890123456789   ** RSX-11M V3.1 **   24-JAN-80   20:49:31   DK

```
;       M A C R O   C A S E S
;
; LLAMADA
;       CASES   X,<LISTA>
; DONDE
;       X ES UN INDICE
;       LISTA ES UNA LISTA DE SUBRUTINAS
; EFECTO
;       SE HACE UN JSR PC,A DONDE A ES LA X-AVA SUBRUTINA DE
;               LA LISTA
;
        .MACRO  CASES   X,LISTA
                .PSECT  CASES
                CASES1=.
                .WORD   LISTA
                .PSECT
                MOV     X,-(SP)
                ASL     (SP)
                ADD     #CASES1,(SP)
                MOV     @0(SP),(SP)
                JSR     PC,@(SP)+
        .ENDM
```

```
;
;       M A C R O S   G O S U B
;
;       M A C R O   G O S U B S
;
; LLAMADA
;       GOSUBS  NOMBRE,<LISTA>
; EFECTO
;       LLAMA A LA RUTINA NOMBRE CON LOS PARAMETROS DE LA LISTA
;       EN EL STACK CON FORMATO
;               VIEJO R5
;               ARG. 0
;
;
;               ARG. N
;       R5->SP-> # DE ARGS.
;
        .MACRO  GOSUBS  NAME,LISTA                      ; LLAMA A FORTRAN
        MOV     R5,-(SP)
GOSUB1=0
        .IRP    X,<LISTA>
        MOV     X,-(SP)
GOSUB1=GOSUB1+1
        .ENDM
        MOV     #GOSUB1,-(SP)
        MOV     SP,R5
        JSR     PC,NAME
        ADD     #2*GOSUB1+2,SP
        MOV     (SP)+,R5
        .ENDM
;
;
;       M A C R O   G O S U B R
;
; LLAMADA
;       GOSUBR  NOMBRE,A0,A1,A2,A3,A4,A5
; EFECTO
;       SE CARGAN LOS ARGUMENTOS NO NULOS A0,...., A5 EN LOS REGS.
;       R0,....,R5 Y SE HACE UN JSR PC,NOMBRE
; NOTA
;       NO SE SALVAN PREVIAMENTE LOS REGISTROS
;       LOS REGS. NO USADOS NO SE MODIFICAN
;       A0,....,A5 DEBEN SER ARGUMENTOS VALIDOS PARA MOV
;
        .MACRO  GOSUBR  NAME,A0,A1,A2,A3,A4,A5  ; LLAMA CON LOS REGS.
        .MCALL  GGOSUB
        GGOSUB  <A0>,R0
        GGOSUB  <A1>,R1
        GGOSUB  <A2>,R2
        GGOSUB  <A3>,R3
        GGOSUB  <A4>,R4
        GGOSUB  <A5>,R5
        JSR     PC,NAME
        .ENDM
        .MACRO  GGOSUB  A,R
        .IIF    NB,<A>,MOV A,R
        .ENDM
;
;       M A C R O   G O S U B F
;
; LLAMADA
;       GOSUBF  NOMBRE,<LISTA>
; EFECTO
;       LLAMA A LA RUTINA NOMBRE . LOS PARAMETROS SE PASAN
;       EN EL FORMATO DE FORTRAN
; NOTA
;       FORTRAN ESPERA ARGUMENTOS CALL BY NAME, I.E. HAY QUE PONER
;       # A LOS ARGS.
;
        .MACRO  GOSUBF  NAME,LISTA
        .MCALL  PUSH,POP
        GOSUB1=0
        .IRP    X,<LISTA>
        GOSUB1=GOSUB1+1
        .ENDM
        PUSH    <R0,R1,R2,R3,R4,R5>
        SUB     #2*GOSUB1+2,SP
        MOV     SP,R5
        MOV     #GOSUB1,(R5)+
        .IRP    X,<LISTA>
        MOV     X,(R5)+
        .ENDM
        MOV     SP,R5
        JSR     PC,NAME
        ADD     #2*GOSUB1+2,SP
        POP     <R5,R4,R3,R2,R1,R0>
        .ENDM
01234567890123456789    ** RSX-11M V3.1 **      24-JAN-80       20:50:46        DK0:[2,2]PUSHPOP.MAC
01234567890123456789    ** RSX-11M V3.1 **      24-JAN-80       20:50:46        DK0:[2,2]PUSHPOP.MAC
01234567890123456789    ** RSX-11M V3.1 **      24-JAN-80       20:50:46        DK0:[2,2]PUSHPOP.MAC



SSSSSSSS    UU      UU    SSSSSSSS    HH      HH    PPPPPPPP      OOOOOO      PPPPPPPP
```

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS (PDP-11)

ANEXOS 2

1984

Palacio de Minería Calle de Tacuba 5 primer piso Deleg. Cuauhtemoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

PDP-11 FUNDAMENTALS & INSTRUCTIONS

COURSE OUTLINE

DAY 1

I. Introduction

II. PDP-11 Family of Computers

III. PDP-11 Hardware Overview

IV. Console Panels

V. Addressing Modes

VI. Programming Examples

VII. Demonstration Lab (Optional)

VIII. Homework
  A. Review Sheet #1
  B. Reading
    1. Intro. to Programming - Chaps. 1, 2, and begin 3
    2. Processor Handbook - Chaps 1, 2, and begin 3

DAY 2

I. Homework Review

II. Complete Addressing Modes

III. PAL - 11A Programming Examples

IV. Implementing A Program
  A. Bootstrap Loader
  B. Absolute Loader
  C. PAL - 11A Assembler

V. Lab

VI. Homework
  A. Review Sheet #2
  B. Reading - Intro. to Programming - complete Chap. 3

CONTENTS

NOTE

This handbook is for information purposes and is subject to change without notice.

Cou PDP-11 Fundamentals & Instructions

DEC __________ Customer __________

Week _____ of _____

Date __________

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
8:15					
9:00	Introduction Objectives Course Materials	System Block Diagram (p. A-5 to 7)	Review Quiz Handout (p. C-2)	Stack (Processor handbook Chapter 5)	Processor Handboo Chapter 5 Traps Interrupts
10:00	PDP-11 Hardware Specifications Intro 11 Chap. 2	Instruction Word Formats - Handout Pg.A-15.19	Instructions & Address Modes	Subroutine Techniques	Vectors Power Fail Handl
11:00	Review of Prerequisite Material Handout P. C-1	Basic Instructions & Address Modes Processor Handbook	Analysis of Sample Program "Sum 4"	Analysis of Sample Program	Analysis of Samp Program pg. A-39
	L	U	N	C	H
1:00	Unibus Intro-11 Ch. 3.1	Analysis of Sample Program	I/O Processing (p. A-32 thru 33) (and p.7,8,9)	↓	Review
2:00	Memory Intro-11 Ch. 3.8	↓ Lab2 Introduction	Analysis of Sample Program	↓	Final Exam Critique
3:00	Console Operation Handout P. B-1 Lab 1	Lab 2 "Echo" p. A-34	Lab 3 "Subroutine" p. A-37 - A-38	Lab 4 "Interrupt-driven I/O" (Choice ~~of p. A-39 or A-40~~ or A-41)	
4:00	"Sum 1" pg. B-1 ↓	↓	↓		
5:15					

A-2a

DAY 3

I. Homework Review

II. Direct Input / Output Processing

III. Programming Examples

IV. Text Editor

V. Lab

VI. Homework
A. Programming
B. Review Sheet #3
C. Reading
1. Processor Handbook - Chap. 4

DAY 4

I. Homework Review

II. Program Looping Techniques

III. Stacks

IV. Subroutine Processing

V. Lab

VI. Homework
A. Review Sheet #4
B. Reading
1. Peripherals Handbook
2. Processor Handbook - 5.1/5.4

DAY 5

I. Homework Review

II. Traps

III. Interrupts

IV. Example Programs

V. Position Independent Code

VI. Software Sneak Preview

VII. Optional Lab

VIII. Final

PDP-11 FUNDAMENTALS AND INSTRUCTIONS

This course presents the organization and features of the PDP-11. It is applicable to all processors in the PDP-11 family, and is designed to prepare the student for further training in PDP-11 hardware or software at a machine or assembly language level.

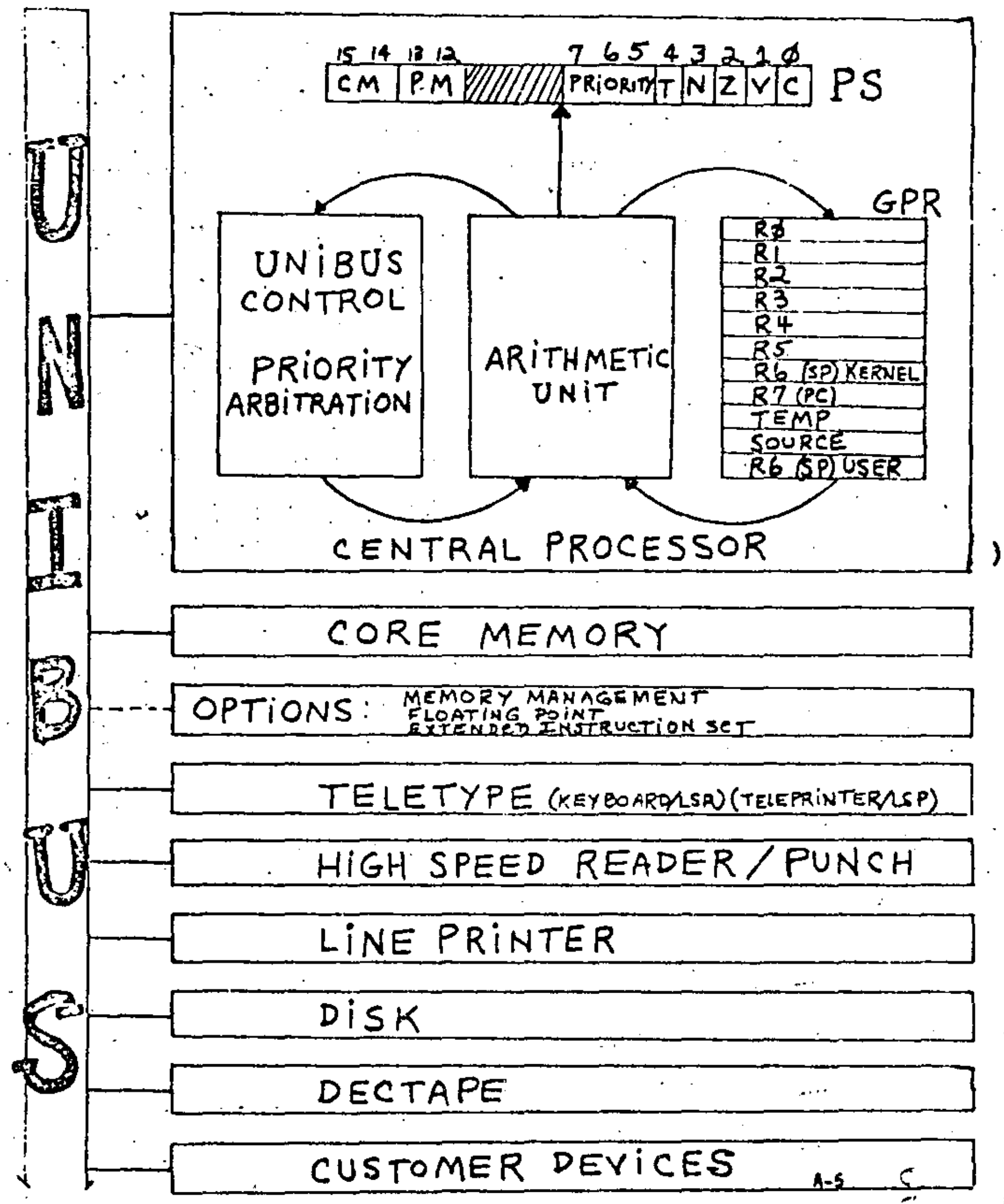
Length: 5 days

Prerequisites: The student should be familiar with binary and octal numbering systems, conversions, and arithmetic and logic operations in these bases. Prior experience with machine or assembly language instructions is required. Experience with "higher level" languages such as Fortran, Basic, Cobol, etc. does not generally prepare the student for this course. Attendance of the Introduction to Minicomputers course is recommended for those not meeting the above prerequisites.

Content: The following major topics are presented: Features common to all PDP-11s, memory organization, registers, operand addressing, instruction set, stack operations, subroutines, decision making, communication with peripherals, priority interrupt structure, traps, and paper tape loaders.

The course also presents an overview of: central processor organization and operation, unibus transactions, standard software, and additional features of larger PDP-11s. A portion of the course will be devoted to supervised laboratory sessions.

PDP 11 SYSTEM BLOCK DIAGRAM



MAJOR STATES

FETCH

1. Purpose: to obtain an instruction from memory.
2. CP will enter FETCH upon completion of previous instruction.
3. PC specifies from where instruction will come, and is incremented by two after use.
4. Instruction (octal code) delivered to instruction register.
5. Instruction decoded; its nature determines next major state.

SOURCE (SRC)

1. Purpose: obtain source operand.
2. CP will enter SRC if the instruction is double operand and source address mode ≠ Ø.
3. Address calculated; data obtained and stored in SRC register.
4. CP enters DESTINATION or EXECUTE major state.

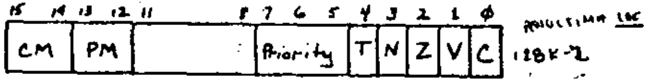
DESTINATION (DST)

1. Purpose: to obtain destination operand.
2. CP may enter DST from either FETCH or SRC.
3. Address calculated; data obtained and brought to arithmetic unit.
4. CP enters EXECUTE major state.

EXECUTE

1. Purpose: execute the instruction and store the result.
2. CP may enter EXECUTE from either FETCH, SRC or DST.

PROCESSOR STATUS WORD



CONDITION CODES

C bit (bit Ø) - set if carry from most significant bit.
V bit (bit 1) - set if arithmetic overflow.
Z bit (bit 2) - set if result = Ø.
N bit (bit 3) - set if result is negative.

TRACE TRAP

T bit (bit 4) - if set, causes processor trap (used by ODT).

PRIORITY

(bits 5, 6, 7) Specify current priority level of processor.

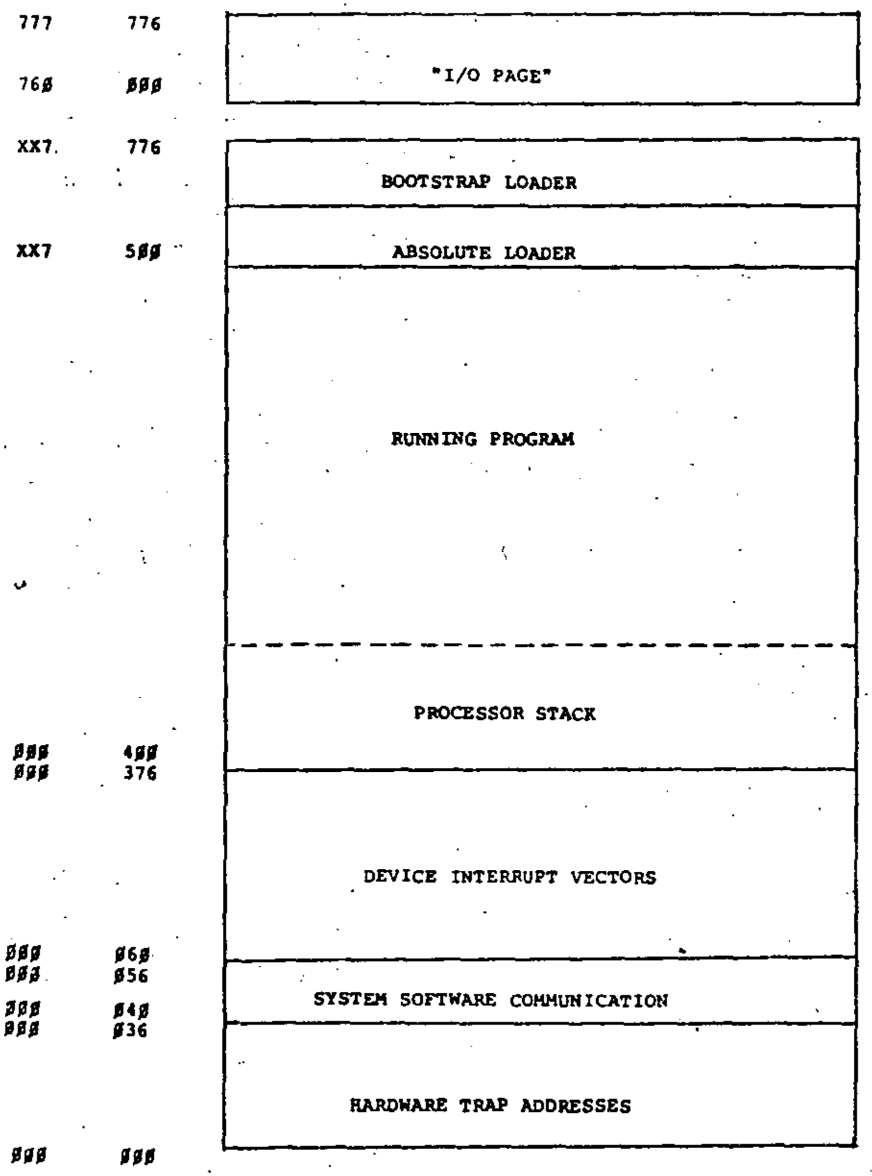
PREVIOUS MODE

(bits 12, 13) Mode prior to the last interrupt or trap.
Kernel = ØØ, User = 11.

CURRENT MODE

(bits 14, 15) Present mode. Kernel = ØØ, User = 11.

PDP-11 MEMORY ALLOCATION



	11/05 11/10	11/15 11/20	11/35 11/40	11/45 11/50
General Purpose Registers	8	8	8	16
Memory Management	NO	NO	optional	optional
Stack Overflow Detection	400 (fixed)	400 (fixed)	400 or programmable (option)	programmable
Extended Arithmetic (hardware)	option (external)	option (external)	option (internal) MUL, DIV, ASH, ASHC	standard (internal)
Floating Point	software only	software only	hardware option 32 bit word	hardware option 32 or 64 bit word

	11/05 11/10	11/15 11/20	11/35 11/40	11/45 11/50
Maximum Memory Size (words)	28K	28K	124K	124K
Maximum Address Space	32K	32K	128K	128K

PDP-11 DIFFERENCE LIST

11/Ø5 11/1Ø	11/15 11/2Ø	11/35 11/4Ø	11/45 11/5Ø
OEM End User	OEM End User	OEM End User	OEM End User
Basic Instruction set	Basic Instruction set	Basic instruction set & XOR, SOB, MARK, SXT, RTT	Same as 11/4Ø & MUL, DIV, ASH, ASHC, SPL
JMP/JRS (R)+ uses (register) after autoincr.	Same as 11/Ø5	JMP/JSR (R)+ uses (register) before autoincr.	Same as 11/4Ø
JMP/JSR %R traps to loc. 4 (illegal instruction).	Same as 11/Ø5	Same as 11/Ø5	JMP/JSR %R traps to 1Ø (reserved instruction).
OPR %R, (R)+ or -(R) or @(R)+ or @-(R) uses R before autoincr./autodec.	OPR %R,(R)+ or -(R) or @(R)+ or @-(R) uses R after autoincr./autodec.	Same as 11/2Ø	Same as 11/Ø5
MOV PC,LOC stores PC of instruction + 2 in LOC.	MOV PC,LOC stores PC of instruction + 4 in LOC.	Same as 11/2Ø	Same as 11/Ø5

2

11/Ø5 11/1Ø	11/15 11/2Ø	11/35 11/4Ø	11/45 11/5Ø
Upon program HALT, PC of instruction just past HALT is displayed.	Upon Program HALT, PC of the HALT instruction is displayed.	Same as 11/Ø5	Same as 11/Ø5
LOAD ADDR is not modified during execution. To start program again, depress START. LOAD ADDR is reg. 17 and can be addr by the CPU as 177717, so a program can set up a new start addr.	LOAD ADDR value is modified once START is pressed. To start again, first LOAD ADDR.	Same as 11/Ø5 except cannot be addr by program.	Same as 11/2Ø
Attempts to EXAM/DEP odd addrs (except GPRs) will cause bit Ø of addr to be disregarded (i.e. 1ØØ1 will result in 1ØØØ).	Attempts to EXAM/DEP odd addrs (except GPRs) will hang the CPU. To unhang, depress START with HALT switch enabled.	Same as 11/Ø5.	ADDR err light comes on. To unhang, depress START with HALT switch enabled.
Odd addr or nonexistent references using the SP cause a HALT (i.e. double bus error occuring in trap service of first error).	Same as 11/Ø5	Odd addr or nonexist. references using SP cause a fatal trap. On bus err in trap service, a new stack is created at locs. Ø and 2.	Odd addr or nonexist. references cause trap to loc. 4. Bus cycle aborted during bus pause of that instr. and same as 11/4Ø.
Byte operations to odd byte of PS do not trap. Not all bits may exist.	Byte operations to odd byte of PS cause odd addr traps.	Same as 11/Ø5	Same as 11/Ø5

11/Ø5 11/1Ø	11/15 11/2Ø	11/35 11/4Ø	11/45 11/5Ø
SWAB instr clear V.	SWAB instr does not affect V.	Same as 11/Ø5	Same as 11/Ø5
Stack limit boundary fixed at 4ØØ$_8$. Violations serviced by OVFL trap.	Same as 11/Ø5	Optional variable stack limit boundary. Use of red or yellow zones on either basic or variable boundary.	Same as 11/4Ø
No red zone on stack overflow.	Same as 11/Ø5	Red zone trap occurs if stack is > 16 words below boundary. This trap saves PC+2 and PS on new stack at locs. Ø and 2.	Red zone trap occurs if stack is 16 words beyond limit. Saves same as 11/4Ø.
Read reference to stack will <u>not</u> cause overflow trap.	Read reference to stack can cause overflow trap.	Same as 11/Ø5	Same as 11/Ø5
First instr in an interrupt routine will not be executed if another interrupt with higher priority occurs.	First instr in an interrupt service routine is guaranteed to be executed.	Same as 11/Ø5	Same as 11/Ø5

11/Ø5 11/1Ø	11/15 11/2Ø	11/35 11/4Ø	11/45 11/5Ø
NPRs are not serviced in HALT state.	Same as 11/Ø5	Same as 11/Ø5	NPRs are serviced in HALT state.
BUS REQUESTS are serviced in single instr mode.	BUS REQUESTS are not serviced in single instr mode.	Same as 11/Ø5	Same as 11/2Ø
If RTI sets T bit, T bit trap is acknowledged after instruction following RTI.	Same as 11/Ø5	If RTI sets T bit, T bit trap is acknowledged immediately following RTI. (Use RTT to accomplish same as 11/2Ø).	Same as 11/4Ø
No RTT instruction.	Same as 11/Ø5	If RTT sets T bit, T bit trap occurs after instruction following RTT.	Same as 11/4Ø
If an interrupt occurs during an instruction that has the T bit set, T bit trap is acknowledged before the interrupt.	Same as 11/Ø5	Same as 11/Ø5	If an interrupt occurs during an instruction that has the T bit set the interrupt is acknowledged before T bi trap.

TYPE OF 11 SYSTEM	SUPPORT FEATURES MEMORY MGT	EIS	FPP	MAX MEMORY SIZE (WORDS)	PROGRAMMABLE STACK LIMIT	ADDRESS RANGES	CONSOLE	BUS
11/Ø3 LSI	NA	OPT	FIS OPT	28K MOS &Core	NO	16 bit	ROM Emulator/or Key Pad	LSI BUS (No Unibus
11/Ø4 OEM	NA			28K MOS & CORE	NO	16 bit	Emulator/or Key Pad	Unibus
End User OEM 11/1Ø & 11/Ø5	NA			28K Core	NO	16 bit	ROM Yes	Unibus
(Original System) 11/2Ø & 11/15	NA			28K Core	NO	16 bit	Yes	Unibus
11/34	Yes	Yes	(OPT) Yes	124K Either Core or MOS	NO	18 bit	ROM Emulator/or Key Pad	Unibus
End User OEM 11/4Ø & 11/35	Yes (OPT)	Yes (OPT)	FIS ONLY	124K Core	OPT	18 bit	Yes	Unibus
End User OEM 11/45+11/50+11/55	Yes	Yes	Yes (OPT)	124K Bipolar MOS Core	OPT	18 bit	Yes	Unibus
End User 11/70	Yes	Yes	Yes (OPT)	1Ø24K Core	OPT	22 bit	Yes	Massbus

OVERVIEW

PDP-11 FAMILY OF COMPUTERS

5

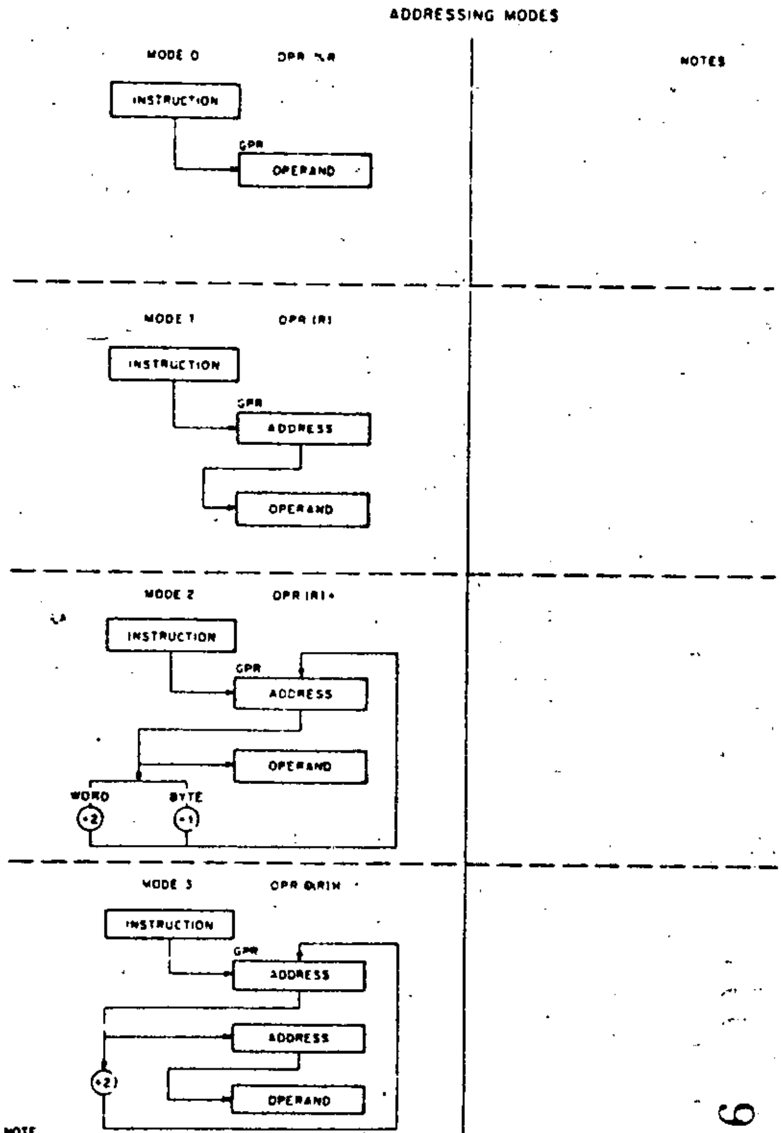
11/Ø5 11/1Ø	11/15 11/2Ø	11/35 11/4Ø	11/45 11/5Ø
T bit trap will not sequence out of WAIT.	T bit trap will sequence out of WAIT.	Same as 11/2Ø	Same as 11/Ø5
Explicit references to the PSW can set or clear the T bit. (CLR PSW(PSW=17776) will clear the T bit along with the rest of the PSW.	Same as 11/Ø5	T bit can be set or cleared only implicitly (CLR PSW (PSW=177776) will not affect T.)	Same as 11/4Ø
RESET does not clear RUN light.	RESET clears RUN light, e.g. program loops that make frequent use of RESET may not appear to be running.	Same as 11/2Ø	Same as 11/Ø5
Power fail immediately ends RESET and traps.	Power fail during RESET is not recognized until after instruction is finished (too late).	Same as 11/2Ø	Same as 11/Ø5

GENERAL REGISTER ADDRESSING

MODE	OCTAL	SYMBOLIC	OPERATION
REGISTER	Ø	R	SPECIFIED REGISTER CONTAINS OPERAND; REGISTER ADDRESS IS THE EFFECTIVE ADDRESS
REGISTER DEFERRED	1	(R)	SPECIFIED REGISTER CONTAINS EFFECTIVE ADDRESS
AUTOINCREMENT	2	(R)+	SPECIFIED REGISTER CONTAINS EFFECTIVE ADDRESS (POST-INCREMENT)
AUTOINCREMENT DEFERRED	3	@(R)+	SPECIFIED REGISTER CONTAINS THE ADDRESS OF THE EFFECTIVE ADDRESS (POST-INCREMENT)
AUTODECREMENT	4	-(R)	SPECIFIED REGISTER CONTAINS EFFECTIVE ADDRESS (PRE-DECREMENT)
AUTODECREMENT DEFERRED	5	@-(R)	SPECIFIED REGISTER CONTAINS THE ADDRESS OF THE EFFECTIVE ADDRESS (PRE-DECREMENT)
INDEXED	6	X(R)	SPECIFIED REGISTER CONTAINS INDEX VALUE; SEQUENTIAL WORD LOCATION CONTAINS BASE ADDRESS; SUM IS EFFECTIVE ADDRESS
INDEXED DEFERRED	7	@X(R)	SPECIFIED REGISTER CONTAINS INDEX VALUE; SEQUENTIAL WORD LOCATION CONTAINS BASE ADDRESS; SUM IS ADDRESS OF EFFECTIVE ADDRESS

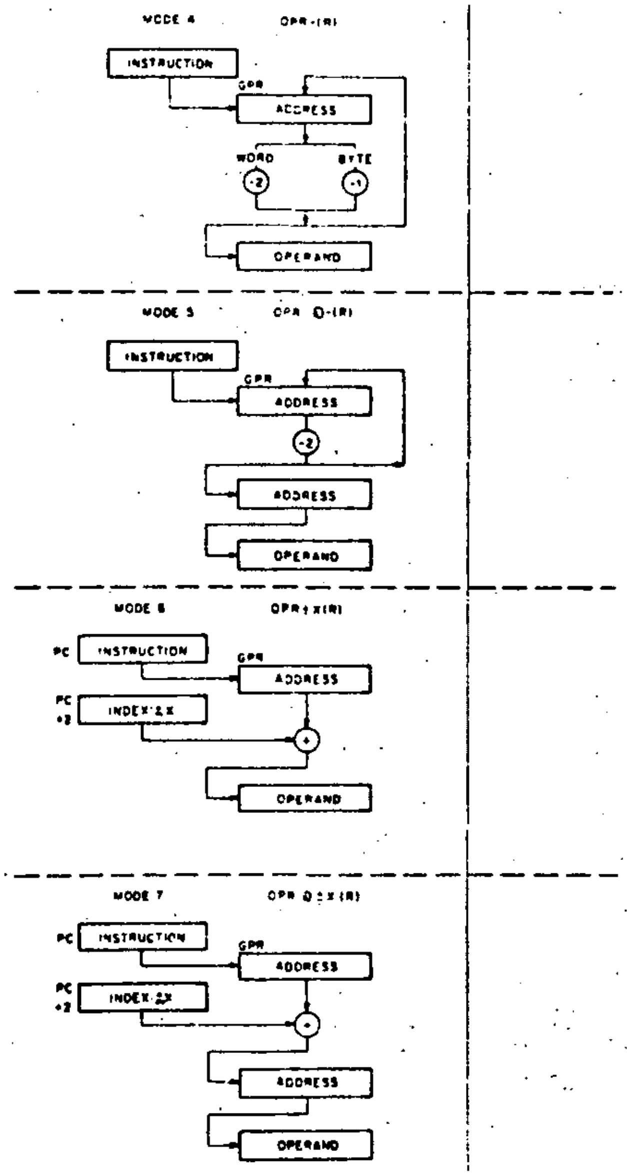
PC REGISTER ADDRESSING

MODE	OCTAL	SYMBOLIC	OPERATION
IMMEDIATE	27	#N	SEQUENTIAL WORD LOCATION CONTAINS OPERAND (N)
ABSOLUTE	37	@#A	SEQUENTIAL WORD LOCATION CONTAINS THE EFFECTIVE ADDRESS--A
RELATIVE	67	A	A IS EFFECTIVE ADDRESS; OFFSET VALUE (ENABLING ACCESS OF A FROM PRESENT LOCATION) CONTAINED IN SEQUENTIAL WORD LOCATION
RELATIVE DEFERRED	77	@A	A IS ADDRESS OF THE EFFECTIVE ADDRESS; OFFSET VALUE (ENABLING ACCESS OF A FROM PRESENT LOCATION) CONTAINED IN SEQUENTIAL WORD LOCATION

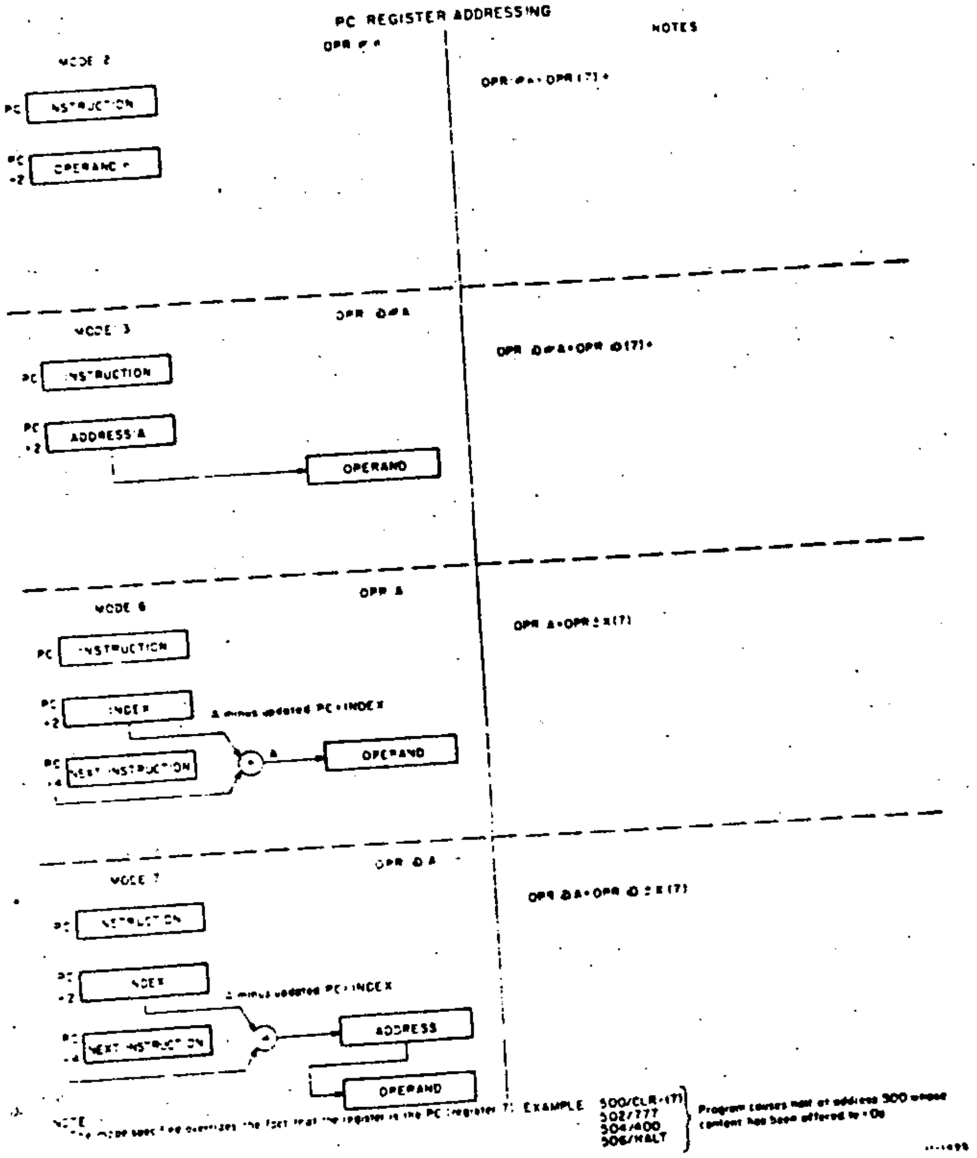


NOTE
R equals a number between 0 and 7.

Addressing Modes (sheet 1 of 3)



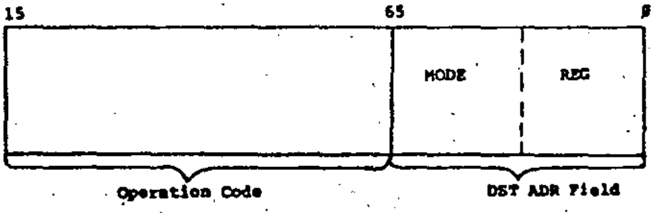
Addressing Modes (sheet 2 of 3)



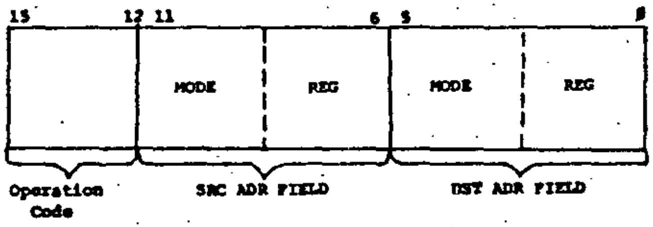
Addressing Modes (sheet 3 of 3)

INSTRUCTION FORMATS

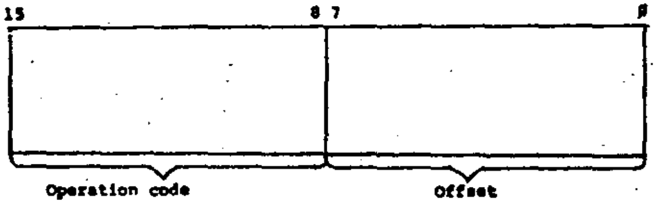
SINGLE OPERAND



DOUBLE OPERAND



BRANCH



Note: Not all formats shown.

SUM GROUP

There are four programs in this series. They offer solutions (each using a different addressing mode) to the same problem:

> Tables A, B, and C each contain five one word entries. Add corresponding entries from A and B and store the result in the corresponding entry of C. Do this without modifying tables A and B, i.e., A(I) + B(I) = C(I).
> Note: The programs do not load tables A or B.

```
;   PROGRAM SUM1
;   THIS VERSION USES AUTOINCREMENT ADDRESSING

          RØ=%Ø
          R1=%1
          R2=%2
          A=1ØØØ
          B=2ØØØ
          C=3ØØØ

          .=4ØØØ

START:    MOV #A,RØ              ;SET UP STARTING ADDRESSES
          MOV #B,R1              ;OF TABLES
          MOV #C,R2
MORE:     MOV (RØ)+,(R2)         ;GET ENTRY FROM TABLE A
          ADD (R1)+,(R2)+        ;ADD ENTRY FROM B, STORE IN C
          CMP R2,#C+12
          BEQ DONE               ;FINISHED?
          BR MORE                ;NO, GO BACK
DONE:     HALT                   ;YES, STOP

        .END START
```

R0 = 1.
R1= 2
R2 = 3

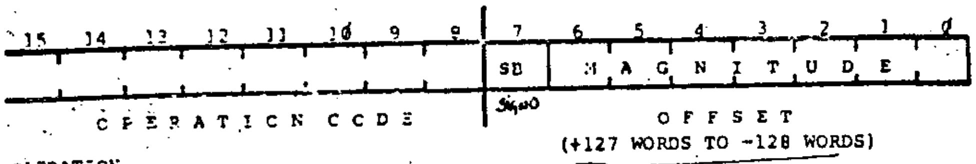
```
; PROGRAM SUM2
; VARIATION ON AUTOINCREMENT ADDRESSING

        RØ=%Ø
        R1=%1
        R2=%2
        R3=%3
        A=1ØØØ
        B=2ØØØ
        C=3ØØØ

        .=4ØØØ   — [illegible] be 1

START:  MOV #-5,R3          ; SET UP COUNTER
        MOV #A,RØ           ; SET UP STARTING ADDRESSES
        MOV #B,R1           ; OF TABLES
        MOV #C,R2
MORE:   MOV (RØ),(R2)       ; GET ENTRY FROM TABLE A
        ADD (R1),(R2)       ; ADD ENTRY FROM B, STORE IN C
        INC R3
        BEQ DONE            ; FINISHED?
        TST (RØ)+           ; NO-- INCREMENT REGISTERS
        TST (R1)+
        TST (R2)+
        BR MORE             ; GO BACK
DONE:   HALT                ; YES, STOP

        .END START
```

```
; PROGRAM SUM3
; THIS VERSION IMPLEMENTS INDEXED ADDRESSING

        RØ=%Ø
        A=1ØØØ
        B=1Ø12
        C=2ØØØ

        .=4ØØØ

START:  CLR RØ              ; SET UP RØ
MORE:   MOV A(RØ),C(RØ)     ; GET ENTRY FROM TABLE A
        ADD B(RØ),C(RØ)     ; ADD ENTRY FROM TABLE B
        CMP RØ,#B-A-2       ; FOR THIS FORMULA TO WORK, TABLE B
                            ; MUST IMMEDIATELY FOLLOW TABLE A
        BEQ DONE            ; FINISHED?
        TST (RØ)+           ; NO, INCREMENT RØ BY 2
        BR MORE             ; THEN GO BACK TO MORE NOT START

DONE:   HALT

        .END START
```

```
; PROGRAM SUM4
;THIS VERSION USES RELATIVE MODE, INCREMENTING THE
;OFFSET TO ACCESS THE TABLES OF DATA

        R1=%1
        A=1ØØØ
        B=2ØØØ
        C=3ØØØ

        .=4ØØØ

START:  MOV #-5,R1          ;SET UP COUNTER
AA:     MOV A,C             ;GET ENTRY FROM TABLE A
BB:     ADD B,C             ;ADD ENTRY FROM TABLE B
        INC R1
        BEQ DONE            ;FINISHED?
        ADD #2,AA+2         ;NO, ADD 2 TO THE OFFSETS TO
        ADD #2,AA+4         ;ACCESS NEXT ENTRIES IN TABLES
        ADD #2,BB+2         ;A, B, AND C
        ADD #2,BB+4
        BR AA               ;GO BACK
DONE:   HALT

        .END START
```

12

BRANCH INSTRUCTIONS



(+127 WORDS TO -128 WORDS)

en complemento de 2.

OPERATION

TEST THE CONDITION CODE BIT(S)
IF CONDITION(S) MET, BRANCH TO EFFECTIVE ADDRESS DEFINED BY OFFSET
IF CONDITION(S) NOT MET, EXECUTE NEXT SEQUENTIAL INSTRUCTION

OFFSET

SIGNED (TWO'S COMPLEMENT) DISPLACEMENT WITHIN 8 BITS SPECIFYING THE NUMBER OF WORDS FROM THE UPDATED PC TO THE EFFECTIVE ADDRESS

CALCULATION

THE PC IS EXPRESSING A BYTE ADDRESS, BUT THE OFFSET IS EXPRESSED IN WORDS THEREFORE, BEFORE BEING ADDED TO THE PC TO DETERMINE THE EFFECTIVE ADDRESS, THE OFFSET MUST ALSO BE EXPRESSED IN BYTES.

THE HARDWARE ACCOMPLISHES THIS BY SHIFTING THE OFFSET ONCE TO THE LEFT (MULTIPLY BY 2) AND SIGN EXTENDING (BIT 7 TO BITS 8-15) TO FORM A 16 BIT NUMBER.

RANGE (WORDS)	OFFSET (LOW 8 BITS)
-200	200
⋮	
-5	373
-4	374
-3	375
-2	376
-1	377
Ø	ØØØ
+1	ØØ1
+2	ØØ2
+3	ØØ3
+4	ØØ4
+5	ØØ5
⋮	
+177	177

EFFECTIVE ADDRESS =

$$(\text{OFFSET} * 2) + (. + 2)$$

OFFSET =

$$\frac{\text{EFFECTIVE ADDRESS} - (. + 2)}{2}$$

NOTE: .+2 IS THE UPDATED PC

SUMMARY OF BRANCH INSTRUCTIONS

UNCONDITIONAL BRANCH

	BRANCH EFFADR→PC	BR EFFADR ØØØ4ØØ+xxx	TRANSFER CONTROL TO EFFADR UNCONDITIONALLY

CONDITIONAL BRANCHES

SIMPLE	BRANCH IF MINUS IF N=1, EFFADR→PC	BMI EFFADR 1ØØ4ØØ+xx	TRANSFER CONTROL TO EFFADR IF N BIT IS SET
	BRANCH IF PLUS IF N=Ø, EFFADR→PC	BPL EFFADR 1ØØØØØ+xxx	TRANSFER CONTROL TO EFFADR IF N BIT IS CLE
	BRANCH IF EQUAL ZERO IF Z=1, EFFADR→PC	BEQ EFFADR ØØ14ØØ+xxx	TRANSFER CONTROL TO EFFADR IF Z BIT IS SET
	BRANCH IF NOT EQUAL ZERO IF Z=Ø, EFFADR→PC	BNE EFFADR ØØ1ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF Z BIT IS CLE
	BRANCH IF OVERFLOW SET IF V=1, EFFADR→PC	BVS EFFADR 1Ø24ØØ+xxx	TRANSFER CONTROL TO EFFADR IF V BIT IS SET
	BRANCH IF OVERFLOW CLEAR IF V=Ø, EFFADR→PC	BVC EFFADR 1Ø2ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF V BIT IS CLE
	BRANCH IF CARRY SET IF C=1, EFFADR→PC	BCS EFFADR 1Ø34ØØ+xxx	TRANSFER CONTROL TO EFFADR IF C BIT IS SET
	BRANCH IF CARRY CLEAR IF C=Ø, EFFADR→PC	BCC EFFADR 1Ø3ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF C BIT IS CLE
UNSIGNED	BRANCH IF LOWER IF C=1, EFFADR→PC	BLO EFFADR 1Ø34ØØ+xxx	TRANSFER CONTROL TO EFFADR IF C BIT IS SET
	BRANCH IF HIGHER OR SAME IF C=Ø, EFFADR→PC	BHIS EFFADR 1Ø3ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF C BIT IS CLE
	BRANCH IF LOWER OR SAME IF $C_{\vee}Z=1$, EFFADR→PC	BLOS EFFADR 1Ø14ØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT O C BIT IORed WITH Z BIT EQUALS ONE
	BRANCH IF HIGHER IF $C_{\vee}Z=Ø$, EFFADR→PC	BHI EFFADR 1Ø1ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT C C BIT IORed WITH Z BIT EQUALS ZERO
SIGNED	BRANCH IF LESS THAN ZERO IF $N_{\veebar}V=1$, EFFADR→PC	BLT EFFADR ØØ24ØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT (N BIT XORed WITH V BIT EQUALS ONE
	BRANCH IF GREATER OR EQUAL ZERO IF $N_{\veebar}V=Ø$, EFFADR→PC	BGE EFFADR ØØ2ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT (N BIT XORed WITH V BIT EQUALS ZERO
	BRANCH IF LESS OR EQUAL ZERO IF $Z_{\vee}(N_{\veebar}V)=1$, EFFADR→PC	BLE EFFADR ØØ34ØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT (Z BIT IORed WITH (N XORed WITH V) EQUALS
	BRANCH IF GREATER THAN ZERO IF $Z_{\vee}(N_{\veebar}V)=Ø$, EFFADR→PC	BGT EFFADR ØØ3ØØØ+xxx	TRANSFER CONTROL TO EFFADR IF THE RESULT (Z BIT IORed WITH (N XORed WITH V) EQUALS

LOOPING TECHNIQUES

PROGRAM SEGMENTS BELOW USED TO CLEAR A 5Ø. WORD TABLE

1. AUTOINCREMENT (POINTER ADDRESS IN GPR)

```
           RØ=%Ø
           MOV #TBL,RØ
  LOOP:    CLR (RØ)+
           CMP RØ,#TBL+1ØØ.
           BNE LOOP
```

2. AUTODECREMENT (POINTER AND LIMIT VALUES IN GPR)

```
           RØ=%Ø
           R1=%1
           MOV #TBL,RØ
           MOV #TBL+1ØØ.,R1
  LOOP:    CLR -(R1)
           CMP R1,RØ
           BNE LOOP
```

3. COUNTER (DECREMENTING A GPR CONTAINING COUNT)

```
           RØ=%Ø
           R1=%1
           MOV #TBL,RØ
           MOV #5Ø.,R1
  LOOP:    CLR (RØ)+
           DEC R1
           BNE LOOP
```

4. INDEX REGISTER MODIFICATION (INDEXED MODE; MODIFYING INDEX VALUE)

```
           RØ=%Ø
           CLR RØ
  LOOP:    CLR TBL(RØ)
           ADD #2,RØ
           CMP RØ,#1ØØ.
           BNE LOOP
```

5. FASTER INDEX REGISTER MODIFICATION (STORING VALUES IN GPR)

```
           RØ=%Ø
           R1=%1
           R2=%2
           MOV #2,R1
           MOV #100.,R2
           CLR RØ
  LOOP:    CLR TBL(RØ)
           ADD R1,RØ
           CMP RØ,R2
           BNE LOOP
```

6. ADDRESS MODIFICATION (INDEXED MODE; MODIFYING BASE ADDRESS)

```
           RØ=%Ø
           MOV #TBL,RØ
  LOOP:    CLR Ø(RØ)
           ADD #2,LOOP+2
           CMP LOOP+2,#1ØØ.
           BNE LOOP
```

LIGHT GROUP

There are four programs in this series. They each cause different patterns of lights to be moved through the console data lights (not so on the 11/Ø5). Each is based upon the fact that RØ is displayed when a RESET instruction is executed. The number of consecutive RESETs needed for the eye to pick up the pattern depends upon the speed of the machine (2-3 is comfortable for the 11/2Ø).

Program LIGHT1

1. This program moves a series of four lights through the data lights from right to left.

2. At some poir.ts, only three lights will show due to the use of he C bit in the ROL instruction. (Note the use of he MOV SWR,R3 instruction instead of TST SWR.)

```
; PROGRAM MOVES S   OF 4 LIGHTS THROUGH THE DATA
; LIGHTS.  MAKES USE OF THE RESET INSTRUCTION WHICH
; CAUSES THE CONTENTS OF RØ TO BE DISPLAYED IN THE
; DATA LIGHTS (NOT SO ON THE 11/Ø5).

         RØ=%Ø
         R3=%3
         SWR=17757Ø

         .=4ØØØ

START:   MOV #17,RØ             ;INITIALIZE RØ
MOVE:    ROL RØ                 ;ROTATE VALUE IN RØ
         RESET                  ;TWO RESETS OK FOR 11/2Ø
         RESET                  ;MORE NEEDED FOR 11/4Ø AND 11/4
                                ;WANT TO CONTINUE?
         MOV SWR,R3             ;CHECK THE SWR
                                ;USE MOV INSTRUCTION BECAUSE
                                ;IT SETS Z BIT WHEREAS TST
                                ;SETS THE Z BIT BUT CLEARS
                                ;THE C BIT--VALUE IN RØ WOULD
                                ;DISAPPEAR
         BEQ MOVE               ;CONTINUE IF ZERO
         HALT                   ;HALT IF NON-ZERO

         .END START
```

Program LIGHT 2

1. This program moves one light (starting with bit ∅) from right to left up through bit 15 or to just below a single bit set in the SR and then back again to bit ∅. The procedure continues (one can change the upper limit on movement simply by changing the single console switch set) back and forth until...

2. Program halts when a one is placed in the SR.

```
          R∅=%0
          R1=%1
          PC=%7
          SR=17757∅

          .=2∅∅∅

START:    MOV #1,R∅              ;START WITH R∅=1
          MOV #1,R1              ;WHEN SR=1, HALT

LP1:      CMP SR,R1
          BEQ FIN
          RESET
          RESET
          RESET
          RESET
          ROL R∅                 ;ROTATE R∅ TO LEFT
          CMP R∅,SR              ;DOES R∅ EQUAL LIMIT SET BY SR
          BEQ LP2                ;YES--START RIGHT ROTATES
          BR LP1                 ;NO--CHECK FOR HALT OR DISPLAY

LP2:      ROR R∅                 ;ROTATE R∅ TO RIGHT
          RESET                  ;DISPLAY R∅
          RESET
          RESET
          RESET
          CMP R∅,R1              ;HAS R∅ ROTATED BACK TO = 1?
          BEQ LP1                ;YES--START MOVEMENT TO LEFT
                                 ;AGAIN
          BR LP2                 ;NO--CONTINUE RIGHT ROTATES
FIN:      HALT

          .END START
```

Program LIGHT3

1. This program starts with the two middle lights lit (bits 7 and 8) and then moves these lights out in opposite directions to the extreme lights (bits 15 and ∅) and then back again to the center, so on and so forth until...

2. A non-zero value is placed in the console switches.

```
          ; PROGRAM MOVES CONSOLE LIGHTS FROM CENTER
          ; OUT TO ENDS, BACK TO CENTER, OUT AGAIN, ETC.

          R∅=%∅
          R1=%1
          R2=%2
          SWR=17757∅

          .=2∅∅∅
START:    MOV #2∅∅,R1         ; PLACE LIGHTS
          MOV #4∅∅,R2         ; IN CENTER

MOV2R∅:   MOV R1,R∅           ; R∅ BUILT FROM R1 AND R2
          ADD R2,R∅           ; COULD USE XOR AND MOV INSTEAD

DISPLY:   RESET               ; DISPLAY
          RESET
          RESET
          RESET
          TST SWR
          BEQ LIMIT           ; IF NON-ZERO IN SWITCHES, HALT
          HALT

LIMIT:    TST R2              ; HAS R2 BEEN ROTATED ALL THE WAY LEFT?
          BMI AGAIN

ROTATE:   ROR R1              ; NO, ROTATE R1 RIGHT AND R2 LEFT
          ROL R2
          BR MOV2R∅

AGAIN:    MOV #1,R2           ; SET UP OUTSIDE CONDITIONS
          MOV #1∅∅∅∅∅,R1
          JMP MOV2R∅          ; DISPLAY AND ROTATE

          .END START
```

PROGRAMMING EXAMPLES

I. PROGRAM TO COUNT NEGATIVE NUMBERS IN A TABLE

```
        ;2Ø. 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
```

```
DIFF      000544   PC       =%000007   R0     =%000000   R1     =%000001
R2       =%000002  R3       =%000003   R4     =%000004   R5     =%000005
SP       =%000006  START     000500    SUM1    000530    SUM2    000536
         * 001012



                             ;PROGRAMMING EXAMPLE
                             ;SUBTRACT CONTENTS OF LOCS 700-710
                             ;FROM CONTENTS OF LOCS 1000-1010

             000000          R0=%0
             000001          R1=%1
             000002          R2=%2
             000003          R3=%3
             000004          R4=%4
             000005          R5=%5
             000006          SP=%6
             000007          PC=%7

             000500          .=500
      000500 012706 START:   MOV  #.,SP          ;INIT STACK POINTER
             000500
      000504 012701          MOV #700,R1
             000700
      000510 012702          MOV #712,R2
             000712
      000514 012703          MOV #1000,R3
             001000
      000520 012704          MOV #1012,R4
             001012

      000524 005000          CLR R0
      000526 005005          CLR  R5

      000530 062105 SUM1:    ADD  (R1)+,R5   ;START ADDING
      000532 020102          CMP R1,R2       ;FINISHED ADDING?
      000534 001375          BNE SUM1        ;IF NOT BRANCH BACK
      000536 062300 SUM2:    ADD  (R3)+,R0    ;START ADDING
      000540 020304          CMP R3,R4       ;FINISHED ADDING?
      000542 001375          BNE SUM2        ;IF NOT BRANCH BACK

      000544 160500 DIFF:    SUB R5,R0       ;SUBTRACT RESULTS

      000546 000000          HALT            ;THAT'S ALL

             000700          .=700
      000700 000001          .WORD 1,2,3,4,5
      000702 000002
      000704 000003
      000706 000004
      000710 000005

             001000          .=1000
      001000 000004          .WORD 4,5,6,7,8.
      001002 000005
      001004 000006
      001006 000007
      001010 000010

             000500           END START
```

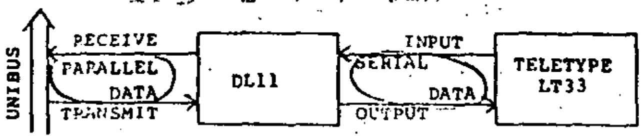
```
DIFF      000544   PC      =%000007   R0      =%000000   R1      =%000001
R2       =%000002  R3      =%000003   R4      =%000004   R5      =%000005
SP       =%000006  START    000500    SUM1     000520    SUM2     000536
         = 001012
```

```
                           ;PROGRAMMING EXAMPLE
                           ;SUBTRACT CONTENTS OF LOCS 700-710
                           ;FROM CONTENTS OF LOCS 1000-1010

        000000             R0=%0
        000001             R1=%1
        000002             R2=%2
        000003             R3=%3
        000004             R4=%4
        000005             R5=%5
        000006             SP=%6
        000007             PC=%7

        000500             .=500
000500 012706 START:  MOV   #. ,SP       ;INIT STACK POINTER
        000500
000504 012701         MOV #700,R1
        000700
000510 012702         MOV #712,R2
        000712
000514 012703         MOV #1000,R3
        001000
000520 012704         MOV #1012,R4
        001012

000524 005000         CLR R0
000526 005005         CLR  R5

000530 062105 SUM1:   ADD  (R1)+,R5  ;START ADDING
000532 020102         CMP R1,R2      ;FINISHED ADDING?
000534 001375         BNE SUM1       ;IF NOT BRANCH BACK
000536 062300 SUM2:   ADD  (R3)+,R0   ;START ADDING
000540 020304         CMP R3,R4      ;FINISHED ADDING?
000542 001375         BNE SUM2       ;IF NOT BRANCH BACK

000544 160500 DIFF:   SUB R5,R0       ;SUBTRACT RESULTS

000546 000000         HALT            ;THAT'S ALL

        000700        .=700
000700 000001         .WORD 1,2,3,4,5
000702 000002
000704 000003
000706 000004
000710 000005

        001000        .=1000
001000 000004         .WORD 4,5,6,7,8.
001002 000005
001004 000006
001006 000007
001010 000010

        000500        .END START
```

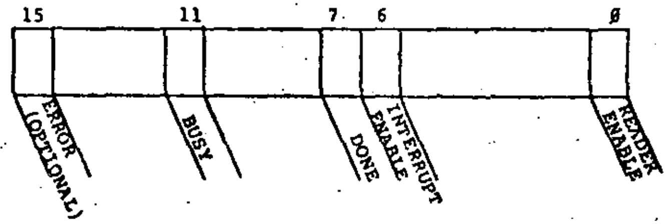
The Bootstrap Loader Program (Assumes loading into a 4K system)

```
           Octal              Assembly
Address    Code      Label    Code                   Comment

           ØØØØØ1             R1=%1                  ;USED FOR THE DEVICE ADDRESS
           ØØØØØ2             R2=%2                  ;USED FOR THE LOAD ADDRESS DISPLACEMENT
           Ø174ØØ             LOAD=174ØØ             ;DATA MAY BE LOADED NO LOWER
                                                     ;THAN THIS
           Ø17744             .=17744                ;START ADDRESS OF THE BOOTSTRAP LOADER
Ø17744     Ø167Ø1    START:   MOV DEVICE, R1         ;PICK UP DEVICE ADDRESS,
           ØØØØ26                                    ;PLACE IN R1
Ø1775Ø     Ø127Ø2    LOOP     MOV #.-LOAD+2,R2       ;PICK UP ADDRESS DISPLACEMENT,
           ØØØ352                                    ;PLACE IN R2
Ø17754     ØØ5211    ENABLE:  INC @R1                ;ENABLE THE PAPER TAPE READER
Ø17756     1Ø5711    WAIT:    TSTB @R1               ;WAIT UNTIL FRAME
Ø1776Ø     1ØØ376             BPL WAIT               ;IS AVAILABLE
Ø17762     116162             MOVB 2(R1),LOAD(R2)    ;STORE FRAME READ
           ØØØØØ2                                    ;FROM TAPE IN MEMORY BYTE
           Ø174ØØ
Ø1777Ø     ØØ5267             INC LOOP+2             ;INCREMENT LOAD ADDRESS
           177756                                    ;DISPLACEMENT
Ø17774     ØØØ765    BRNCH:   BR LOOP                ;GO BACK AND READ MORE DATA
Ø17776     ØØØØØØ    DEVICE:  Ø                      ;ADDRESS OF INPUT DEVICE STATUS
                                                     ;REGISTER
```

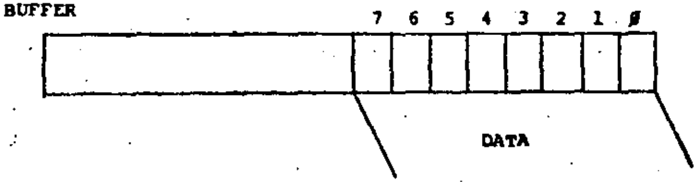
TELETYPE AND CONTROL, BLOCK DIAGRAM



INPUT STATUS REGISTER



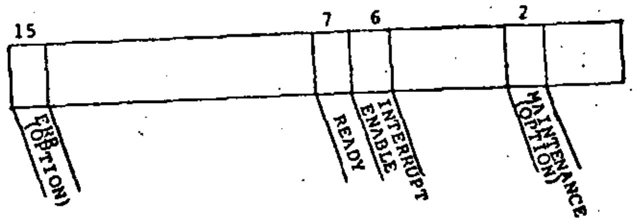
INPUT BUFFER



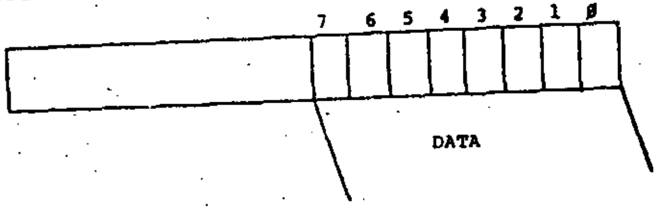
EXAMPLE INPUT (TTY)

```
        READ:  INC @#TKS        ;SET RDR ENB
        LOOP:  TSTB @#TKS       ;LOOK FOR DONE
               BPL  LOOP        ;WAIT IF DONE = Ø
               MOVB @#TKB,RØ    ;READ CHARACTER
```

OUTPUT STATUS REGISTER



OUTPUT BUFFER



EXAMPLE OUTPUT (TTY)

```
        PUNCH:  TSTB @#TPS       ;TEST FOR READY
                BPL  PUNCH       ;WAIT IF READY = Ø
                MOVB RØ,@#TPB    ;PUNCH CHARACTER
```

```
                             ; PROGRAMMING EXAMPLE
                             ; ACCEPT (WITH IMMEDIATE ECHO) AND STORE 20.
                             ; CHARS. FROM THE KYBD. OUTPUT CR AND LF
                             ; ECHO ENTIRE STRING FROM STORAGE

        000000               [illegible]
        000001               [illegible]
        000006               [illegible]
        000015               CR=15
        000012               LF=12
        177560               TKS=177560
        177562               TKB=TKS+2
        177564               TPS=TKB+2
        177566               TPB=TPS+2

        001000               [illegible]

 001000 012706 START:        ~~MOV #.,SP~~                 ; INIT SP
        001000
 001004 012700               MOV #SAVE+2,R0           ; SA OF BUFFER BEYOND CR &LF
        001100
 001010 012701               MOV #20.,R1              ; CHARACTER COUNT
        000024

 001014 105737 IN:           TSTB @#TKS               ; CHAR IN BUFFER?
        177560
 001020 100375               BPL IN                   ; IF NOT BRANCH BACK AND WAIT
 001022 105737 ECHO:         TSTB @#TPS               ; CHECK TELEPRINTER READY STATUS
        177564
 001026 100375               BPL ECHO
 001030 113737               MOVB @#TKB,@#TPB         ; ECHO CHARACTER
        177562
        177566
 001036 113720               MOVB @#TKB,(R0)+         ; STORE CHARACTER AWAY
        177562
 001042 005301               DEC R1
 001044 001363               BNE IN                   ; FINISHED INPUTTING?

 001046 012700               MOV #SAVE,R0             ; SA OF BUFFER INCLUDING CR & LF
        001076
 001052 012701               MOV #22.,R1              ; COUNTER OF BUFFER INCLUDING CR
        000026

 001056 105737 OUT:          TSTB @#TPS               ; CHECK TELEPRINTER READY STATUS
        177564
 001062 100375               BPL OUT
 001064 112037               MOVB (R0)+,@#TPB         ; OUTPUT CHARACTER
        177566
 001070 005301               DEC R1
 001072 001371               BNE OUT                  ; FINISHED OUTPUTTING?
 001074 000000               HALT

 001076    015 SAVE:         .BYTE CR,LF
 001077    012
        001124               .=.+20
        001000               .END START

                             PAGE    001

00000 ERRORS
```

The STACK

Definition

The STACK is an area of memory reserved by the programmer for subroutine/interrupt linkage or temporary storage.

It is a dynamic inverted table using the "last in, first out" concept which advances downward as items are added and retreats upward as items are removed.

Initialization

General Purpose Register 6 serves as the system STACK pointer; it will automatically keep track of "where you are" in the STACK. Hence, the first instruction in a program is usually that which initializes the STACK pointer.

Although the programmer may begin the STACK at any address, it is customarily begun at USER PROGRAM START ADDRESS-2 and will advance toward address 4ØØ (advancing below address 4ØØ will cause a STACK overflow error trap to occur).

```
                 non-PIC                         PIC

                  SP=%6                          SP=%6
                                                 PC=%7
         SETSTK: MOV #.,SP                       .=Ø
                                         SETSTK: MOV PC,SP
                                                 TST -(SP)
```

Usage

Any of the conditions below will cause data to be automatically added ("pushed") onto the STACK by the system:

Jump to SubRoutine instruction
device interrupt
software interrupt (any trap instruction)
hardware interrupt (any error trap condition)

Either of the instructions below will cause data to be automatically removed ("popped") from the STACK by the system:

ReTurn from Subroutine instruction
ReTurn from Interrupt instruction

The programmer may also use the STACK for storage and retrieval of data by simulating the automatic system operations above.

```
         To store, "PUSH"            To retrieve, "POP"

         MOV DST,-(SP)               MOV (SP)+,DST
```

Summary of Argument Handling

1. **Autoincrement** MOV (R5)+,RØ

   To access sequential arguments as operands.

   ```
   JSR R5,SUB
   1ØØ.
   1ØØØ
   6ØØØ
   ```

2. **Autoincrement Deferred** MOV @(R5)+,RØ

   To access sequential arguments as effective addresses.

   ```
   JSR R5,SUB
   FLD1
   FLD2
   ```

3. **Indexed** MOV 4(R5),RØ

   To access arguments randomly as operands.

   ```
   JSR R5,SUB
   1ØØ.
   2ØØ.
   675.
   345.
   ```

4. **Indexed Deferred** MOV @4(R5),RØ

   To access arguments randomly as effective addresses.

   ```
   JSR R5,SUB
   FLDA
   FLDB
   FLDC
   FLDD
   ```

```
        ; SUBROUTINE EXAMPLE
        ; INPUT TEN VALUES, SORT, AND
        ; OUTPUT THEM IN SMALLEST TO LARGEST ORDER

        RØ=%Ø
        R1=%1
        R2=%2
        R3=%3
        R4=%4
        R5=%5
        SP=%6
        PC=%7
        TKS=17756Ø
        TKB=TKS+2
        TPS=TKB+2
        TPB=TPS+2

        .=3ØØØ

INITSP: MOV #.,SP                 ; INITIALIZE STACK POINTER
        JSR PC,CRLF               ; GO TO CRLF SUBROUTINE
        JSR R5, OUTPUT            ; GO TO OUTPUT SUBROUTINE
        LINE1                     ; SA OF LINE 1 BUFFER
        69.                       ; NUMBER OF OUTPUTS
        JSR PC,CRLF               ; GO TO CRLF SUBROUTINE
        JSR R5,OUTPUT             ; GO TO OUTPUT SUBROUTINE
        LINE2                     ; SA OF LINE 2 BUFFER
        26.                       ; NUMBER OF OUTPUTS
        JSR PC,CRLF               ; GO TO CRLF SUBROUTINE
        JSR PC,INPUT              ; GO TO INPUT SUBROUTINE
        JSR PC,SORT               ; GO TO SORT SUBROUTINE
        JSR PC,CRLF               ; GO TO CRLF SUBROUTINE
        JSR R5,OUTPUT             ; GO TO OUTPUT SUBROUTINE
        BUFFER                    ; INPUT BUFFER AREA
        1Ø.                       ; NUMBER OF OUTPOTS
        JSR PC,CRLF
        HALT                      ; THE END!!!!

        ; SUBROUTINE TO OUTPUT A CR & LF

CRLF:   TSTB @#TPS                ; TEST TTO READY STATUS
        BPL CRLF
        MOVB #15,@#TPB            ; OUTPUT CARRIAGE RETURN
LNFD:   TSTB @#TPS                ; TEST TTO READY STATUS
        BPL LNFD
        MOVB #12,@#TPB            ; OUTPUT LINE FEED
        RTS PC                    ; EXIT

        ; SUBROUTINE TO OUTPUT A VARIABLE LENGTH MESSAGE
OUTPUT: MOV (R5)+,RØ              ; PICK UP SA OF DATA BLOCK
        MOV (R5)+,R1              ; PICK UP NUMBER OF OUTPUTS
        NEG R1                    ; NEGATE IT
AGAIN:  TSTB @#TPS                ; TEST TTO READY STATUS
        BPL AGAIN
        MOVB (RØ)+,@#TPB          ; OUTPUT CHARACTER
        INC R1                    ; BUMP COUNTER
        BNE AGAIN
        RTS R5
```

```
        ; SUBROUTINE TO INPUT TEN VALUES
INPUT:  MOV #BUFFER,RØ              ; SET UP SA OF STORAGE BUFFER
        MOV #-1Ø.,R1                ; SET UP COUNTER
IN:     TSTB @#TKS                  ; TEST KYBD READY STATUS
        BPL IN
OUT:    TSTB @#TPS                  ; TEST TTO READY STATUS
        BPL OUT
        MOVB @#TKB,@#TPB            ; ECHO CHARACTER
        MOVB @#TKB,(RØ)+            ; STORE CHARACTER
        INC R1                      ; INC COUNTER
        BNE IN
        RTS PC                      ; EXIT

        ; SUBROUTINE TO SORT TEN VALUES
SORT:   MOV #-1Ø.,R4
NEXT:   MOV COUNT,R3
        MOV #BUFFER+9.,RØ
        ADD R3,RØ
        MOVB (RØ)+,R1
LOOP:   CMPB (RØ)+,R1
        BGE GT
LT:     MOVB -(RØ),R2
        MOVB R1,(RØ)+
        MOV R2,R1
GT:     INC R3
        BNE LOOP
INSERT: MOVB R1,BUFFER+1Ø.(R4)
        INC R4
        INC COUNT
        BNE NEXT
        MOV #-9.,COUNT              ; RESTORE LOCATION COUNT
        RTS PC                      ; EXIT

COUNT:  .WORD -9.
LINE1:  .ASCII /INPUT ANY TEN SINGLE DIGIT VALUES (Ø-9); I'LL/
        .ASCII /SORT AND OUTPUT THEM IN/
LINE2:  .ASCII /SMALLEST TO LARGEST ORDER./
BUFFER: .=.+1Ø.

        .END INITSP                 ; FINISHED!!!
```

```
                        ;EXAMPLE OF INTERRUPT LEVEL I/O
                        ;BACKGROUND RINGS TTY BELL
                        ;FOREGROUND ACCEPTS CHARACTERS UNTIL A LINE FEED
                        ;THEN PRINTS ENTIRE MESSAGE


        RØ=%Ø
        R1=%1
        R2=%2
        R3=%3
        R4=%4
        R5=%5
        SP=%6
        PC=%7
        TPS=177564
        TPB=177566
        TKS=17756Ø
        TKB=177562
        .=6Ø
        .WORD TTKSVC,Ø,TTPSVC,Ø
        .=5ØØ
START:  MOV #.,SP               ;SET STACK POINTER
        MOV #NBUF,R1            ;I/O BUFFER ADDRESS
        MOV R1,R2
        CLR RØ
        CLR HLTFLG              ;FLAG TO TELL WHEN TO HALT
        BIS #1ØØ,@#TKS          ;SET READER ENABLE + INTERR. ENABLE
BACK:   INC R5                  ;DELAY COUNTER
        BNE BACK                ;NOT READY TO SEND BELL YET
CK:     TSTB @#TPS              ;IS PRINTER READY
        BPL CK                  ;NOT YET
        MOVB #7,@#TPB           ;MOVE BELL CODE TO PRINT BUFFER
        TST HLTFLG              ;TIME TO HALT?
        BEQ BACK                ;NOT IF ZERO
        HALT                    ;STOP NOW ... RØ CONTAINS  NBR OF
                                ;CHARS IN NAME
        BR START                ;TO RESTART

TTKSVC: INC RØ                  ;BUMP CHAR COUNT
        MOVB @#TKB,R3           ;MOVE CHAR JUST TYPED TO R3
        BIC #2ØØ,R3             ;CLEAR BIT 8
        MOVB R3,(R1)+           ;MOVE CHAR TO MY BUFFER
        CMPB R3,#Ø12            ;WAS IT A LINE FEED TERMINATOR?
        BNE TTKSRT              ;NO ... CONTINUE
        BIC #1ØØ,@#TKS          ;CLEAR KEYBOARD INTERRUPT ENABLE
        BIS #1ØØ,@#TPS          ;SET PRINTER INTERRUPT ENABLE
TTKSRT: RTI                     ;RETURN
TTPSVC: MOVB (R2)+,@#TPB        ;MOVE BYTE FROM BUFFER TO PRINTER BUFF
        CMP R2,R1               ;SENT ALL YET?
        BNE TTPSRT              ;NOT YET
        BIC #1ØØ,@#TPS          ;CLEAR PRINTER INTERRUPT ENABLE
        INC HLTFLG              ;SET HALT FLAG NOW
TTPSRT: RTI
HLTFLG: .WORD Ø
NBUF:   .=.+16Ø.                ;BUFFER FOR 16Ø CHARACTERS
        .END START
```

```
                 ;EXAMPLE OF INTERRUPT LEVEL I/O
                 ;BACKGROUND RINGS TTY BELL
                 ;FOREGROUND ACCEPTS CHARACTERS UNTIL A LINE FEED
                 ;THEN PRINTS ENTIRE MESSAGE


        ~~Ø~~
        ~~1~~
        ~~2~~
        ~~3~~
        ~~4~~
        ~~5~~
        ~~S 6~~
        ~~PC=%7~~
        TPS=177564
        TPB=177566
        TKS=17756Ø
        TKB=177562
        .=6Ø
        .WORD TTKSVC,Ø,TTPSVC,Ø
        .=5ØØ
START:  MOV #.,SP               ;SET STACK POINTER
        MOV #NBUF,R1            ;I/O BUFFER ADDRESS
        MOV R1,R2
        CLR RØ
        CLR HLTFLG              ;FLAG TO TELL WHEN TO HALT
        BIS #1ØØ,@#TKS          ;SET READER ENABLE + INTERR. ENABLE
BACK:   INC R5                  ;DELAY COUNTER
        BNE BACK                ;NOT READY TO SEND BELL YET
CK:     TSTB @#TPS              ;IS PRINTER READY
        BPL CK                  ;NOT YET
        MOVB #7,@#TPB           ;MOVE BELL CODE TO PRINT BUFFER
        TST HLTFLG              ;TIME TO HALT?
        BEQ BACK                ;NOT IF ZERO
        HALT                    ;STOP NOW ... RØ CONTAINS  NBR OF
                                ;CHARS IN NAME
        BR START                ;TO RESTART

TTKSVC: INC RØ                  ;BUMP CHAR COUNT
        MOVB @#TKB,R3           ;MOVE CHAR JUST TYPED TO R3
        BIC #2ØØ,R3             ;CLEAR BIT 8
        MOVB R3,(R1)+           ;MOVE CHAR TO MY BUFFER
        CMPB R3,#Ø12            ;WAS IT A LINE FEED TERMINATOR?
        BNE TTKSRT              ;NO ... CONTINUE
        BIC #1ØØ,@#TKS          ;CLEAR KEYBOARD INTERRUPT ENABLE
        BIS #1ØØ,@#TPS          ;SET PRINTER INTERRUPT ENABLE
TTKSRT: RTI                     ;RETURN
TTPSVC: MOVB (R2)+,@#TPB        ;MOVE BYTE FROM BUFFER TO PRINTER BUFF
        CMP R2,R1               ;SENT ALL YET?
        BNE TTPSRT              ;NOT YET
        BIC #1ØØ,@#TPS          ;CLEAR PRINTER INTERRUPT ENABLE
        INC HLTFLG              ;SET HALT FLAG NOW
TTPSRT: RTI
HLTFLG: .WORD Ø
NBUF:   .=.+16Ø.                ;BUFFER FOR 16Ø CHARACTERS
        .END START
```

```
                  ;EXAMPLE OF INTERRUPT LEVEL I/O
                  ;BACKGROUND RINGS TTY BELL
                  ;FOREGROUND ACCEPTS CHARACTERS FROM READER UNTIL
                  ;A LINE FEED THEN PRINTS ENTIRE MESSAGE

         RØ=%Ø
         R1=%1
         R2=%2
         R3=%3
         R4=%4
         R5=%5
         SP=%6
         PS=177776
         TKS=17756Ø
         TKB=TKS+2
         TPS=TKB+2
         TPB=TPS+2
         .=5ØØ
INIT:    MOV #.,SP              ; SET STACK POINTER
         MOV #BUFFER,R1         ; I/O BUFFER ADDRESS
         MOV R1,R2
         MOV #TKYSUB,@#6Ø       ; SET UP VECTORS
         CLR @#62
         MOV #TPRSUB,@#64
         CLR @#66
         CLR RØ                 ; CHARACTER COUNTER
         BIS #1Ø1,@#TKS         ;SET TTY INPUT ENABLE, INTERRUPT ENABLE
BACKGD:  INC R3                 ; DELAY LOOP
         BNE BACKGD
         BIS #34Ø,@#PS          ; RAISE CP PRIORITY TO PREVENT INTERRUPT
WAIT:    TSTB @#TPS             ; TEST TTO READY STATUS
         BPL WAIT
         MOVB #7,@#TPB          ; OUTPUT BELL
         BIC #34Ø,@#PS          ; LOWER CP PRIORITY TO ALLOW INTERRUPT
         BR BACKGD
TKYSUB:  CMP #212,@#TKB         ;TEST FOR LF
         BEQ EOM                ;YES, PREPARE FOR OUTPUT
         MOVB @#TKB,(R1)+       ; NO, PUT CHARACTER IN BUFFER
         INC RØ                 ; COUNTER
         INC @#TKS              ;SET TTY ENABLE
         RTI
EOM:     BIC #1ØØ,@#TKS         ;CLEAR KYBD INTERRUPT ENABLE
         BIS #1ØØ,@#TPS         ; SET TTO INTERRUPT ENABLE
         RTI


TPRSUB:  MOVB (R2)+,@#TPB       ;OUTPUT A CHARACTER
         CMP R2,R1              ; DONE?
         BEQ FINISH             ; NO
         RTI                    ; YES
STOP:    HALT                   ;ERROR HALT
FINISH:  HALT                   ;CORRECT HALT
BUFFER:  Ø
         .END INIT
```

22

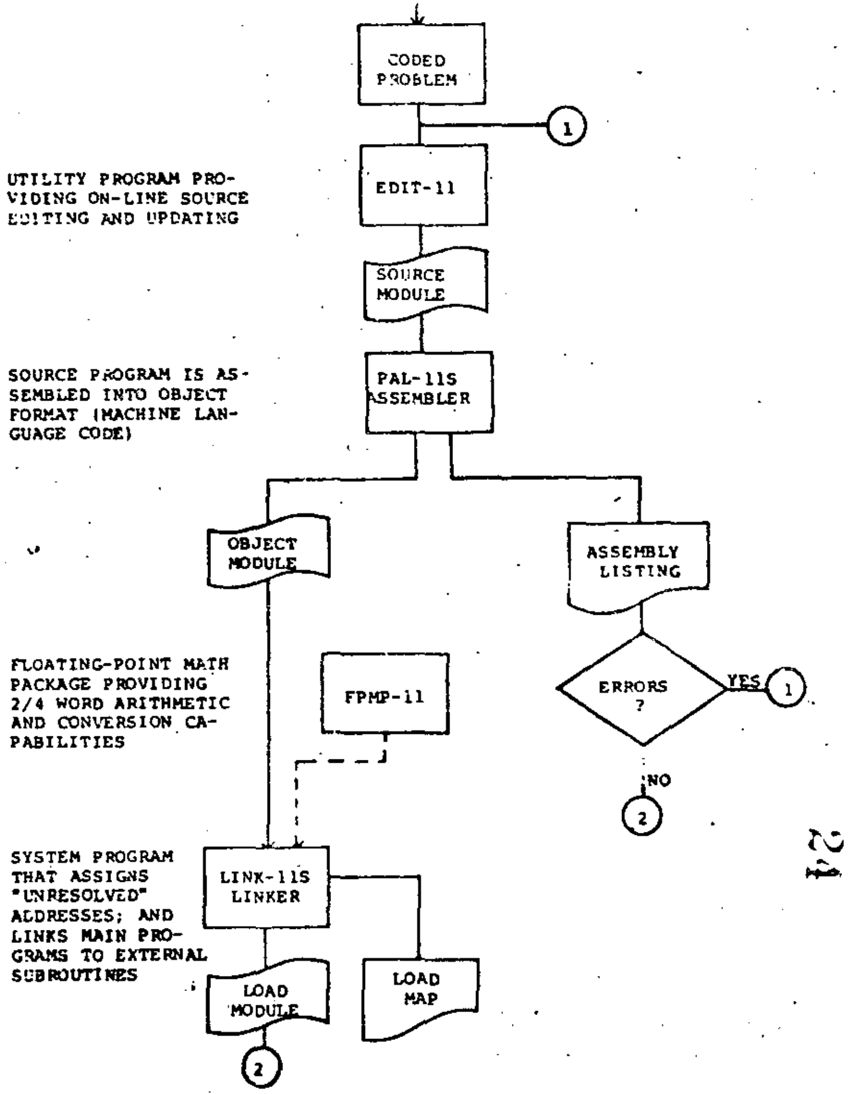
```
                ;EXAMPLE OF INTERRUPT LEVEL I/O
                ;TAPE DUPLICATOR PROGRAM (HSR/HSP)

        RØ=%Ø
        R1=%1
        R2=%2
        SP=%6
        PRS=17755Ø
        PRB=PRS+2
        PPS=PRB+2
        PPB=PPS+2
        TPS=177564
        TPB=TPS+2
        .=1ØØØ
INIT:   MOV #.,SP               ;SET UP STACK
        MOV #BUFFER,R1          ;BUFFER ADDRESS POINTER TO R1 (INPUT)
        MOV R1,R2               ;BUFFER ADDRESS POINTER TO R2 (OUTPUT)
        MOV #HSRSRV,@#7Ø        ;SA HSR SERVICE ROUTINE TO VECTOR
        CLR @#72                ;NO NEED TO SPECIFY NEW PRIORITY LEVEL
        MOV #HSPSRV,@#74        ;SA HSP SERVICE ROUTINE TO VECTOR
        CLR @#76                ;NO NEED TO SPECIFY NEW PRIORITY LEVEL

        BIS #1Ø1,@#PRS          ;SET READER ENABLE, INTERRUPT ENABLE

BACKG:  INC RØ
        BNE BACKG               ;EXAMPLE BACKGROUND PROGRAM TO
CK:     TSTB @#TPS              ;CONTINUOUSLY RING TTYP BELL
        BPL CK
        MOVB #7,@#TPB
        BR BACKG

HSRSRV: TST @#PRS               ;CHECK ERROR BIT (15)
        BMI EOM                 ;EOM MEANS INPUT DONE
        MOVB @#PRB,(R1)+        ;STORE CHARACTER
        INC @#PRS               ;SET READER ENABLE BIT
        RTI                     ;RETURN TO BACKGROUND PROGRAM
EOM:    BIC #1ØØ,@#PRS          ;INPUT DONE--CLEAR HSR INTERRUPT
        BIS #1ØØ,@#PPS          ;SET INTERRUPT ENABLE FOR HSP
        RTI                     ;RETURN TO BACKGROUND PROGRAM

HSPSRV: TST @#PPS               ;CHECK ERROR BIT (15)
        BMI STOP                ;PHYS ERROR--STOP PROGRAM
        MOVB (R2)+,@#PPB        ;PUNCH CHARACTER
        CMP R2,R1               ;DONE?
        BEQ CLRTN               ;YES--CLEAR INTERRUPT AND RETURN
        RTI                     ;NO--KEEP INTERRUPT AND RETURN
CLRTN:  BIC #1ØØ,@#PPS
        RTI
STOP:   HALT                    ;ERROR CONDITION--HALT

BUFFER: Ø                       ;REST OF CORE IS BUFFER AREA

        .END
```

```
                          EXAMPLE

                        TRAP HANDLER

THNDLR:  BIC #ØØØ17,2(SP)    ;CLEAR USER PS CC BITS
         MOV R5,-(SP)        ;SAVE
         MOV R4,-(SP)        ;ALL
         MOV R3,-(SP)        ;GPR
         MOV R2,-(SP)        ;ON
         MOV R1,-(SP)        ;THE
         MOV RØ,-(SP)        ;STACK
         MOV 14(SP),RØ       ;PICK UP COPY OF MAIN PROGRAM PC
         MOV -(RØ),R1        ;USE IT TO GET TRAP INSTRUCTION
         BIC #1777ØØ,R1      ;EXTRACT USER CODE
         ASL R1              ;TO BE ADDRESS VALUE--MAKE IT EVEN
         JMP @TTABLE(R1)     ;GO TO INDICATED ROUTINE

RETURN:  BIS @#PS,16(SP)     ;SET USER PS CC BITS TO REFLECT ROUTINE
         MOV (SP)+,RØ        ;RESTORE
         MOV (SP)+,R1        ;ALL
         MOV (SP)+,R2        ;GPR
         MOV (SP)+,R3        ;FROM
         MOV (SP)+,R4        ;THE
         MOV (SP)+,R5        ;STACK
         RTI                 ;RETURN TO MAIN PROGRAM

TTABLE:  TASKØ               ;DISPATCH
         TASK1               ;TABLE
         TASK2               ;CONTAINING
           .                 ;ALL
           .                 ;ROUTINE
         TASK77              ;ADDRESSES
----------------------------------------------------------------
TASKØ:   OPR
           .
           .
         JMP RETURN
TASK1:   OPR
           .
           .
         JMP RETURN
TASK2:   OPR
           .
           .
         JMP RETURN
           .
           .
           .
           .
           .
TASK77:  OPR
           .
           .
         JMP RETURN
```

23

```
                ;EXAMPLE OF INTERRUPT LEVEL I/O
                ;TAPE DUPLICATOR PROGRAM (HSR/HSP)
                ;POSITION INDEPENDENT CODE

        RØ=%Ø
        R1=%1
        R2=%2
        SP=%6
        PC=%7
        PRS=17755Ø
        PRB=PRS+2
        PPS=PRB+2
        PPB=PPS+2
        TPS=177564
        TPB=TPS+2
        .=1ØØØ
INIT:   MOV PC,SP               ;SET UP STACK FOR
        TST -(SP)               ;PIC PROGRAM
AC1:    MOV PC,R1               ;CALCULATE BUFFER ADDRESS POINTER FOR
        ADD #BUFFER-AC1-2,R1    ;PIC PROGRAM
        MOV R1,R2               ;SA OF BUFFER TO R1 (INPUT)--R2 (OUTPUT)
AC2:    MOV PC,RØ               ;CALCULATE SA OF
        ADD #HSRSRV-AC2-2,RØ    ;HSR SERVICE ROUTINE
        MOV, RØ,@#7Ø            ;LOAD IN VECTOR ADDRESS
        CLR @#72                ;NO NEED TO SPECIFY NEW PRIORITY LEVEL
AC3:    MOV PC,RØ               ;CALCULATE SA OF
        ADD #HSPSRV-AC3-2,RØ    ;HSP SERVICE ROUTINE
        MOV RØ,@#74             ;LOAD IN VECTOR ADDRESS
        CLR @#76                ;NO NEED TO SPECIFY NEW PRIORITY LEVEL

        BIS #1Ø1,@#PRS          ;SET READER ENABLE, INTERRUPT ENABLE

BACKG:  INC RØ
        BNE BACKG               ;EXAMPLE BACKGROUND PROGRAM
CK:     TSTB @#TPS              ;CONTINUOUSLY RING TTYP BELL
        BPL CK
        MOVB #7,@#TPB
        BR BACKG

HSRSRV: TST @#PRS               ;CHECK ERROR BIT (15)
        BMI EOM                 ;EOM MEANS INPUT DONE
        MOVB @#PRB,(R1)+        ;STORE CHARACTER
        INC @#PRS               ;SET READER ENABLE BIT
        RTI                     ;RETURN TO BACKGROUND PROGRAM
EOM:    BIC #1ØØ,@#PRS          ;INPUT DONE--CLEAR HSR INTERRUPT
        BIS #1ØØ,@#PPS          ;SET INTERRUPT ENABLE FOR HSP
        RTI                     ;RETURN TO BACKGROUND PROGRAM

HSPSRV: TST @#PPS               ;CHECK ERROR BIT (15)
        BMI STOP                ;PHYS ERROR--STOP PROGRAM
        MOVB (R2)+,@#PPB        ;PUNCH CHARACTER
        CMP R2,R1               ;DONE?
        BEQ CLRTN               ;YES--CLEAR INTERRUPT AND RETURN
        RTI                     ;NO--KEEP INTERRUPT AND RETURN
CLRTN:  BIC #1ØØ,@#PPS
        RTI
STOP:   HALT                    ;ERROR CONDITION--HALT

BUFFER: Ø                       ;REST OF CORE IS BUFFER AREA

        .END
```

PDP-11 PAPER TAPE SOFTWARE OVERVIEW



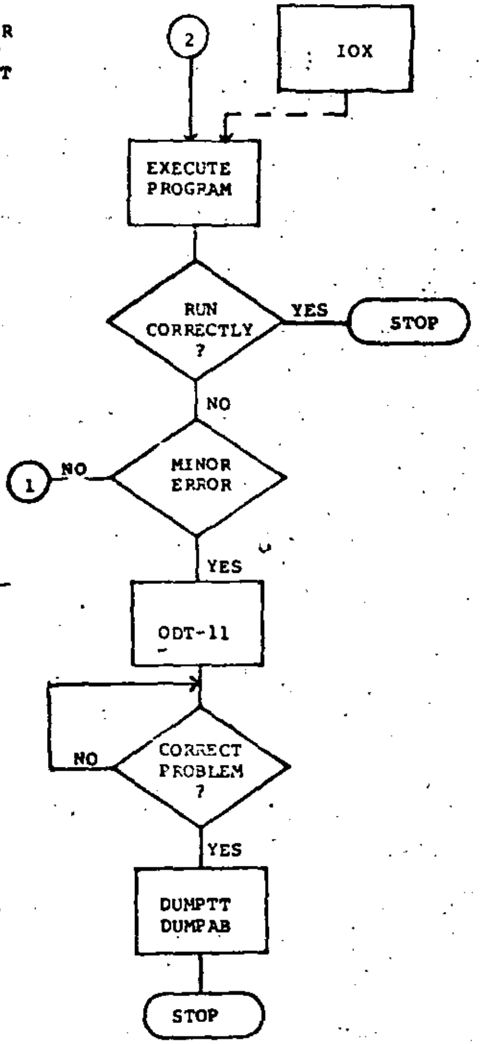
UTILITY PROGRAM PROVIDING ON-LINE SOURCE EDITING AND UPDATING

SOURCE PROGRAM IS ASSEMBLED INTO OBJECT FORMAT (MACHINE LANGUAGE CODE)

FLOATING-POINT MATH PACKAGE PROVIDING 2/4 WORD ARITHMETIC AND CONVERSION CAPABILITIES

SYSTEM PROGRAM THAT ASSIGNS "UNRESOLVED" ADDRESSES; AND LINKS MAIN PROGRAMS TO EXTERNAL SUBROUTINES

INPUT-OUTPUT HANDLER PROVIDING INTERRUPT DRIVEN DATA XFERS AT "READ-WRITE" LEVEL



UTILITY PROGRAM PROVIDING "DYNAMIC ON-LINE DEBUGGING" CAPABILITIES

CORE MEMORY DUMP PROGRAMS ALLOWING "SNAPSHOT" DUMPS OF SELECTED AREAS OF CORE ONTO PERIPHERAL DEVICES

CONSOLE OPERATION

TO EXAMINE MEMORY:

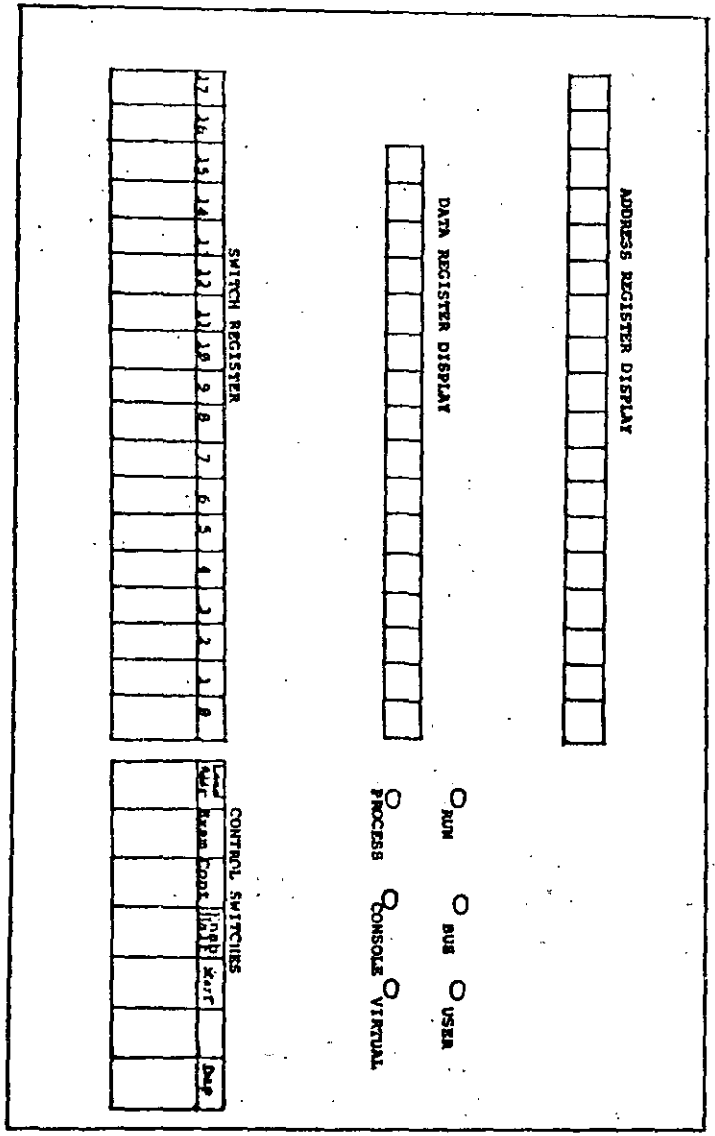
1. HALT the processor.
2. Set SR for the desired address.
3. Press the LOAD ADDRESS key.
4. Press the EXAMINE key.

TO DEPOSIT IN MEMORY:

1. HALT the processor.
2. Set SR for the desired address.
3. Press the LOAD ADDRESS key.
4. Set SR for the desired content.
5. Raise the DEPOSIT key.

TO RUN A PROGRAM:

1. HALT the processor.
2. Set SR for starting address of program.
3. Press the LOAD ADDRESS key.
4. Set ENABLE/HALT switch to ENABLE.
5. Press the START key.

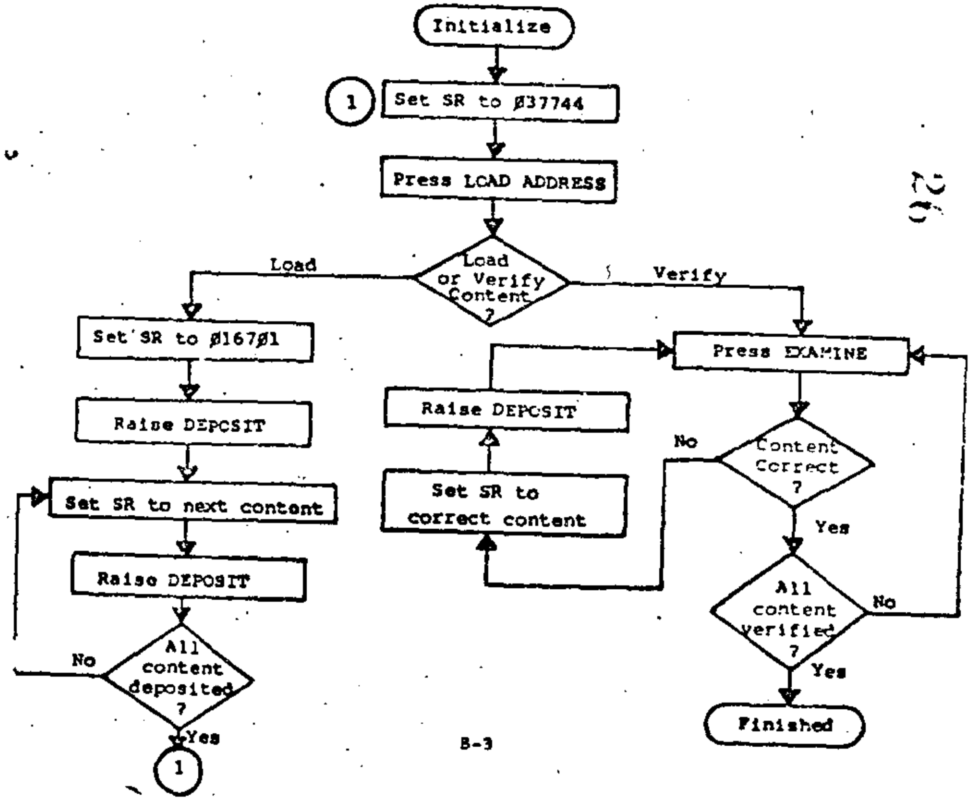


PDP-11 CONSOLE

LOADING AND VERIFYING THE BOOTSTRAP LOADER

(8K System---High Speed Paper Tape Reader)

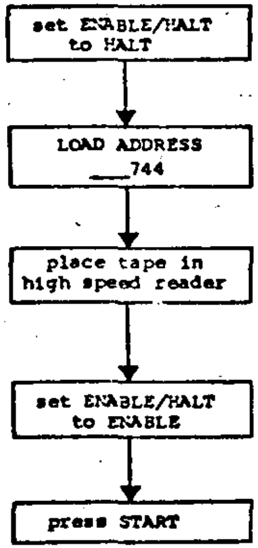
Location	Content
Ø37744	Ø167Ø1
Ø37746	ØØØØ26
Ø3775Ø	Ø127Ø2
Ø37752	ØØØ352
Ø37754	ØØ5211
Ø37756	1Ø5711
Ø3776Ø	1ØØ376
Ø37762	116162
Ø37764	ØØØØØ2
Ø37766	Ø374ØØ
Ø3777Ø	ØØ5267
Ø37772	177756
Ø37774	ØØØ765
Ø37776	17755Ø



LOADING WITH THE BOOTSTRAP LOADER

The BOOTSTRAP LOADER program is designed to load <u>any</u> tape in bootstrap format directly beneath itself (see allocation diagram on 6-12). Presently, only the ABSOLUTE LOADER program and the core dump programs (DUMPTT/DUMPAB) are provided in this format--as they are short enough to fit in the space allotted.

Generally, the absolute formatted core dump programs are used, and the <u>sole purpose of the BOOTSTRAP LOADER is to load the ABSOLUTE LOADER</u>.



starting address of BOOTSTRAP LOADER

position tape (using tape feed button) so that the "special leader" (351 code) is over the sensors

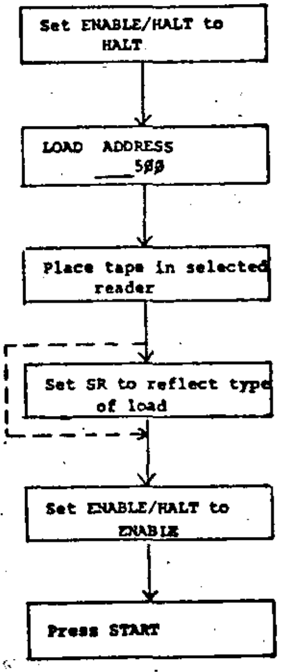
overlook the above is a vam

ABSOLUTE LOADER will be read into memory beginning at address <u>XXX</u>5ØØ.

LOADING WITH THE ABSOLUTE LOADER

The ABSOLUTE LOADER program is designed to load <u>any</u> tape in absolute format -- the majority of the system software (PAL- 11S, ED- 11, LINK- 11S, ODT- 11, IOX, PAL- 11A, etc.) and your user programs which have been assembled and processed by PAL- 11S and LINK- 11S or assembled by the absolute assembler PAL- 11A.

In most cases, the load address is on the binary tape. Realize, however, that the program may be written in Position Independent Code (PIC) and that in this case the user may express any desired load address at load time.


Set ENABLE/HALT to HALT
LOAD ADDRESS ___5ØØ
Place tape in selected reader
Set SR to reflect type of load
Set ENABLE/HALT to ENABLE
Press START

Starting address of ABSOLUTE LOADER

Blank tape positioned over sensors

Bit Ø ⇒ Ø ⇒ normal
SR = ØØØØØ1 ⇒ continuous
SR = N ⇒ load bias

Binary tape will be read into core memory beginning at: address on tape (non-PIC)
address in SR (PIC)

27

USING THE TEXT EDITOR

The text editor is used to generate source tapes of the user's program. The editor is loaded using the ABSOLUTE LOADER and is self-starting.

To input text, type:

```
            XXX  C/T      TEXT LINE    ↲
```

where:

```
            XXX = octal line number
            C/T = CONTROL TAB
             ↲  = RETURN
```

To change a line of text, retype the line correctly using the same line number.

To delete a line, type the line number, CONTROL TAB, then RETURN.

Commands have the format:

```
            X   ↲
```

where:

```
            X = L, R, or P
```

The R command reads a tape from LSR or HSR. It clears the buffer before the read.

The P command punches the text in the buffer to LSP or HSP. It does not clear the buffer.

An L command lists the entire buffer on the TTY.

To clear the buffer, type R with no tape in the reader.

To resequence a program, Punch the program, clear the buffer, and then Read the program back into the buffer.

Never use line feed or rubout.

USING THE ASSEMBLER (PAL-11A)

PAL-11A is used to assemble symbolic code into binary code--to create from the symbolic tape of your program a binary tape of your program which can subsequently be loaded into core memory and executed.

This is normally accomplished in two passes, with an optional third pass for a listing of your program (the latest version of the 8K assembler will give both binary tape and listing on the second pass).

After you have loaded the PAL-11A program (using the ABSOLUTE LOADER), it will start itself automatically and begin the INITIAL DIALOGUE---

PAL-11A types	you respond	this indicates
*S	H↲ or L↲	Symbolic tape to be read from HSR or LSR
*B	H↲ or L↲	Binary tape to be output on HSP or LSP
*L	T↲	Listing to be output on Teleprinter
*T	T↲	user symbol Table to be output on Teleprinter *be certain your symbolic tape is in the proper reader before you respond

ASSEMBLY DIALOGUE

Pass 1: (symbolic tape read in and symbol table output on teleprinter)

END?	↲	pass 1 over; put symbolic tape back in HSR and type CR for pass 2

Pass 2: (symbolic tape read in and binary tape output on HSP)

END?	↲	pass 2 over; put symbolic tape back in HSR and type CR for pass 3

Pass 3: (symbolic tape read in and assembly listing output on teleprinter)

*S	ignore	PAL-11A ready for another assembly

Push the feed button to generate some TRAILER for the binary tape of your program, and remove it from the HSP.

Note: The response to EOF? is E↲ indicating a missing .END statement.

PAL-11A SPECIAL CHARACTERS

Character	Function
form feed	Terminates a line of source code.
line feed	Terminates a line of source code.
carriage return	Terminates the source statement.
:	Label terminator.
=	Direct assignment indicator.
%	Register term indicator.
tab	Terminates an item or field.
space	Terminates an item or field.
#	Immediate expression indicator (mode 27)
@	Deferred addressing indicator.
(	Initial register indicator.
)	Terminal register indicator.
,	Operand field separator.
;	Comment field indicator.
+	Arithmetic addition operator.
-	Arithmetic subtraction operator.
&	Logical AND operator.
!	Logical INCLUSIVE OR operator.
"	Double ASCII character indicator.
'	Single ASCII character indicator.
.	Assembly current location counter.

PAL - 11A ERR CODES

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

Error Code	Meaning
A	Addressing error. An address within the instruction is incorrect.
B	Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.
L	Line buffer overflow. Extra chacters on a line (more than 72_{10}) are ignored.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.
N	Number containing 8 or 9 has no decimal point.
P	Phase error. A label's definition or value varies from one pass to another.
Q	Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.
R	Register-type error. An invalid use of or reference to a register has been made.
S	Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the initial dialogue.
T	Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.

LOADING YOUR BINARY TAPE

The paper tape output of the PAL-11A Assembler is in absolute binary format and is therefore loaded by the ABSOLUTE LOADER. Reference the handout entitled LOADING WITH THE ABSOLUTE LOADER.

RUNNING YOUR PROGRAM

After you have loaded your program into memory using the **ABSOLUTE LOADER**, you are ready to run it. The procedure is **as follows:**

1. Set the ENABLE/HALT switch to HALT.
2. Set switch register to the starting address of your program.
3. Press LOAD ADDRESS.
4. Set the ENABLE/HALT switch to ENABLE.
5. Press START.

WHERE DID I GO WRONG?

Hopefully, you will have no need to reference this section! But occasionally programs do not run as intended--halting without giving the desired result or failing to halt at all. If this has happened to you, take the following remedial steps:

1. Repeat the above sequence (try LOADING YOUR BINARY TAPE and RUNNING YOUR PROGRAM again).
2. EXAMINE your program in memory; compare it with the assembly listing.
3. Check your program THOROUGHLY; determine whether or not the correct instructions have been used.
4. **CALL FOR HELP from your instructor!!!!**

CHALLENGE PROGRAM #1

Two tables of numerical data is created in memory. Three tasks are to be performed on this data. The results are to be left in General Purpose Registers.

R3 = the number of negative values (16 bit, signed, twos compliment) in both tables.

R4 = the number of corresponding matches between entries of both tables.

R5 = the number of total matches between all entries of each table.

TABLE A	TABLE B
035353	100001
007436	007736
165004	055561
165005	100001
071332	071332
176332	060075
000424	060076
010001	060077
100001	100001
177753	177776
177776	000424
035353	035353
060076	077776
164551	164550

Note: each task should be coded separately.

CHALLENGE PROGRAM #2

This program will recognize two ASCII characters within the context of simple operation interaction. The program will request the operator to type a "Y" on a "N". If "Y", the program will print "ES", if "N", the program will print "O!" if other than "N" or "Y" the program will respond "TRY AGAIN".

SAMPLE RUN

```
PLEASE TYPE A "Y" OR "N"  YES
PLEASE TYPE A "Y" OR "N"  NO!
PLEASE TYPE A "Y" OR "N"  G
TRY AGAIN
PLEASE TYPE A "Y" OR "N"
```

CHALLENGE PROGRAM #3

Five lines of text are to be printed out on the console terminal. Each line of text is a different length. The program should use a subroutine to do the data transfers. If one line of text exceeds 64 characters, the subroutine will insert a "CR" carriage return and "LF" line feed.

Each line of text should be a sentence, two of which exceed 64 characters to test the CP/LF specification.

REVIEW SHEET (Day 1)

BINARY > DECIMAL CONVERSION

```
1. 000 000 000 001 100 110=

2. 000 000 000 000 110 110=
```

DECIMAL > BINARY CONVERSION

```
1. 100=

2. 235=
```

OCTAL > DECIMAL CONVERSION

```
1. 000742=

2. 001000=
```

DECIMAL > OCTAL CONVERSION

```
1. 580=

2. 1000=
```

BINARY > OCTAL CONVERSION

```
1. 000 000 001 010 011 100=

2. 000 000 000 101 111 110=
```

OCTAL > BINARY CONVERSION

```
1. 000735=

2. 005224=
```

BINARY ADDITION

```
1.  000 000 100 110 011 100
   +000 000 110 110 111 011
   ------------------------

2.  000 000 011 101 100 101
   +000 000 110 111 111 110
   ------------------------
```

BINARY SUPTRACTION

```
1.  000 000 000 011 110 000
   -000 000 000 001 111 101
   ------------------------

2.  000 000 000 100 001 101
   -000 000 000 011 110 111
   ------------------------
```

OCTAL ADDITION (OPTIONAL)

```
1.  054362          2.  003321
    073441              004407
   +067750             +005622
   -------             -------
```

OCTAL SUDTRACTION (OPTIONAL)

```
1.  013431          2.  011234
   -012054             -010567
   -------             -------
```

LOGICAL AND

```
1.  000 001 010 011 100 101
   &001 010 011 100 101 110
   ------------------------
```

INCLUSIVE OR

```
1.  001 010 011 100 101 110
   !010 011 100 101 110 111
   ------------------------
```

EXCLUSIVE OR

```
1.  010 011 100 101 110 111
   ⊕011 100 101 110 111 000
   ------------------------
```

Review Sheet (Day 2)

Directions: Please select the best possible answer.

1) Given the assembler code CLR R2, the Octal Code is:

   A) 105002
   B) 005002
   C) 050012
   D) None of the Above

2) The starting address for the Absolute Loader on a 12K PDP-11 System is

   A) X03744
   B) 057500
   C) 057744
   D) 005744

3) The Unibus is not capable of bidirectional transfers.

   True or False

4) The Instruction MOVB ØØ1ØØ2 will MOVE the low byte of a 16 bit word.

   True or False

5) Mode 67 is called

   A) relative
   B) absolute
   C) index
   D) relative deferred

6) The maximum amount of true memory that can be used in the basic PDP-11 (I/O Page excluded) is

   A) 4K words
   B) 28K words
   C) 32K words
   D) 24K words

7) In a 16 bit word, bit 15, (the most significant bit) is called the

   A) the positive bit
   B) the leading bit
   C) the negative bit
   D) the signed bit

(continued on back)

8) Given the assembler code MOV RØ, R1 the octal code is:

A) 01 00 10
B) 00 10 02
C) 01 00 01
D) 10 00 10

9) The process of subtraction is accomplished in the PDP-11 by

A) signed arithmetic
B) complementary addition
C) subtract and carry
D) complementary subtraction

10) In loading paper tape software programs, the Editor Produces ________________ to be read by the PAL 11A Assembler.

A) Binary Tape
B) Object Tape
C) Magnetic Tape
D) ASCII Source Tape

REVIEW SHEET (Day 3)

I. CODING. CONSIDER EACH INSTRUCTION TO BE THE INITIAL INSTRUCTION.

GIVEN: ALL CONDITION CODE BITS = Ø FOR EACH INSTRUCTION

(R1)=1030	(1000)=100	(100)=10	(10)=1	(76)=153436
(R2)=2000	(2000)=200	(200)=20	(20)=2	(776)=76
(R3)=3000	(3000)=300	(300)=30	(30)=3	(1776)=300
(R4)=4000	(4000)=400	(400)=40	(40)=4	(2776)=400
(R6)=6000	(6000)=600	(600)=60	(60)=6	(3776)=500
(R7)=5000		(500)=50	(50)=5	(4776)=600

					RESULT↓				
	OCTAL CODE	ASSEMBLER CODE	SEA	DEA	(DEA)	N	Z	V	C
1.	112702 020202								
2.	012252								
3.	006233								
4.	066771 174772 002000								
5.	005252								
6.		SUB #1234,#5000							
7.		ADD R1,-(R2)							
8.		SWAB @-(R1)							
9.		CLR @#177776							
10.		ROLB (R6)							

REVIEW SHEET (Day 4)

Hardware failure does not occur very often.... Before calling DEC Field Service try the following. If every attempt to run fails - including:

1. Leave text mode and try command mode.
2. ABORT or CTRL/C and restart program.
3. Rebootstrap the system, then retry.
4. Mount a fresh disk and rebootstrap. (however, never mount a MASTER (issue) disk until the drive has been checked out.)
5. Start from scratch. Build or SYSGEN onto a clean formatted disk.

TRY THE FOLLOWING

from the front panel you can enter a few instructions to find out if the processor, memory, or the console terminal are dead.

- Toggle into the last memory location

```
MOV  -(R7), -(R7)
```

should load 014747 everywhere in memory.

```
ZERO:  CMP #014747, (R0)+ ; (R0)=10
       BEQ ZERO
       HALT
```

should detect an obvious memory failure.

- Toggle at location zero

```
0/ 012700
2/ 000010
4/ 005020
6/ 000776
```

should load zeroes everywhere in memory.

```
zero: CMP #Ø, (RØ)+ ; RØ=1Ø
      BEQ ZERO
      HALT
```

should detect an obvious memory failure.

- Try this to see if you're hooked up.

```
500/105737
   /177564
   /100375
   /010037
   /177566
   /005200
   /000137
   /000500
```

PDP-11 ASSEMBLY LANGUAGE PROGRAMMING COMPREHENSIVE FINAL EXAMINATION

A Programmer has just received the attached PAL-11A assembly listing for his system and is attempting to analyze it. He has a PDP-11/40 with a console teleprinter and line frequency clock as standard equipment.

You have been asked to help him answer some questions about this program in order to demonstrate your ability to analyze PDP-11 programs. You may use any written references available to answer his questions. If you run into difficulty in the classroom, you may use the PDP-11 laboratory computers to experiment with this program.

You have one-half day and as many attempts as necessary to solve his problems. When you have finished, please ask the instructor to certify your results. You are expected to answer 36 of the 45 questions, correctly. Please work individually, directing any questions to the instructor.

34

Note:
This program has been assembled with a special printout format in order to automatically sequence number each statement. The line number to the left of each statement will serve as a reference for all of the following questions.

```
                                              ;11 ASSEMBLY LANGUAGE PROGRAMMING FINAL EXAMINATION
        000000                    R0=%0
        000001                    R1=%1
        000002                    R2=%2
        000003                    R3=%3
        000004                    R4=%4
        000005                    R5=%5
        000006                    SP=%6
        000007                    PC=%7
        000100'                           .=.+100
000100  001160'  000300                   .WORD   LKINT,300       ;HANDLER ADDRESS, PRIORITY
        001000'                           .=.+674
        177546                            LKCSR=177546
001000  012706   001000'          START:  MOV     #START,SP       ;INITIALIZE STACK POINTER
001004  004567   000202                   JSR     R5,MESAGE       ;REQUEST 4 DIGIT TIME IN SECONDS
001010  001110'                           .WORD   PROMPT          ;TRAILING ARGUMENT, POINTER
001012  004767   000214                   JSR     PC,INPUT        ;INPUT TIME IN SECONDS TO R4
001016  010467   000064                   MOV     R4,DELAY        ;SAVE TIME DELAY INTERVAL
001022  012767   177704   000052          MOV     #-60.,TICK      ;INITIALIZE CLOCK HANDLER COUNT
001030  005067   000050                   CLR     SECOND
001034  052737   000100   177546          BIS     #100,@#LKCSR    ;STARTUP LINE CLOCK
001042  012700   000077                   MOV     #77,R0          ;DATA DISPLAY BIT PATTERN
001046  026767   000032   000032  LOOP:   CMP     SECOND,DELAY    ;DELAY TIME UP?
001054  002005                            BGE     DONE            ;YES
001056  000241                            CLC                     ;NO
001060  006100                            ROL     R0              ;ROTATE LIGHTS (11-45,-70 ONLY)
001062  005500                            ADC     R0              ;WRAP AROUND
001064  000001                            WAIT                    ;DISPLAY R0, SLEEP UNTIL INTERRUPT
001066  000767                            BR      LOOP
001070  042737   000100   177546  DONE:   BIC     #100,@#LKCSR    ;STOP CLOCK
001076  000000                            HALT                    ;REPLACE TIS WITH "NOP" AFTER TEST
001100  000737                            BR      START           ;RE-START IF HIT CONTINUE SWITCH
--------------------------------------------------------------------------------------------------
                                  ;DATA FOR MAIN PROGRAM
001102  000000                    TICKS:  .WORD   0               ;CLOCK 1/60 SEC. COUNTER
001104  000000                    SECOND: .WORD   0               ;CLOCK SECONDS
001106  000000                    DELAY:  .WORD   0               ;INTERVAL
001110     015      012           PROMPT: .BYTE   015,012         ;CR-LF
001112     124      131      120          .ASCII  /TYPE A 4 DIGIT DECIMAL NUMBER PLEASE /
001157     000                            .BYTE   0               ;TERMINATON NULL
                                          .EVEN
--------------------------------------------------------------------------------------------------
                                  ;CLOCK INTERRUPT SUBROUTINE
001160  005267   177716           LKINT:  INC     TICK            ;COUNT ON TICKS PER SECOND
001164  001401                            BEQ     LKSEC           ;ANOTHER SECOND?
001166  000002                            RTI                     ;NO, KEEP TICKING
001170  012767   177704   177704  LKSEC:  MOV     #-60.,TICK      ;RE-INITIALIZE TICKS
001176  005267   177702                   INC     SECOND
001202  112737   000007   177566          MOVB    #7,@#TPB        ;BEEP
001210  000002                            RTI
--------------------------------------------------------------------------------------------------
                                  ;SUBROUTINE TO PRINT A MESSAGE
        177564                            TPS=177564              ;CONSOLE TELEPRINTER
        177566                            TPB=177566
001212  012501                    MESAGE: MOV     (R5)+,R1        ;ADDRESS OF MESSAGE PARAMETER
001214  105737   177564           MLOOP:  TSTB    @#TPS           ;PRINTER READY?
001220  100375                            BPL     MLOOP           ;NO, WAIT
001222  112137   177566                   MOVB    (R1)+,@#TPB     ;YES, PRINT A CHARACTER
001226  001372                            BNE     MLOOP           ;DONE IF NULL, ELSE CONTINUE
001230  000205                            RTS     R5
--------------------------------------------------------------------------------------------------
                                  ;SUBROUTINE TO INPUT A DECIMAL NUMBER 0000 TO 9999
        177560                            TKS=177560
        177562                            TKB=177562              ;CONSOLE TELEPRINTER KEYBOARD
001232  005004                    INPUT:  CLR     R4              ;RESULT
001234  012703   000004                   MOV     #4,R3           ;NUMBER OF CHARACTERS
001240  032737   100200   177560  ILOOP:  BIT     #100200,@#TKS   ;DONE OR ERROR?
001246  001774                            BEQ     ILOOP           ;NO, WAIT
001250  100417                            BMI     ERROR
001252  006304                            ASL     R4              ;CURRENT RESULT TIMES TEN
001254  010402                            MOV     R4,R2           ;*2
001256  006304                            ASL     R4              ;*4
001260  006304                            ASL     R4              ;*8
001262  060204                            ADD     R2,R4           ;*10.
001264  113702   177562                   MOVB    @#TKB,R2        ;READ A CHARACTER
001270  110237   177566                   MOVB    R2,@#TPB        ;ECHO IT
001274  042702   177760                   BIC     #177760,R2      ;STRIP BINARY CODES FOR 0-9
001300  060204                            ADD     R2,R4           ;ACCUMULATE TO TOTAL
001302  005303                            DEC     R3              ;TALLY
001304  001355                            BNE     ILOOP           ;FINISHED?
001306  000207                            RTS     PC              ;YES
001310  000000                    ERROR:  HALT                    ;NO RECOVERY
        001000'                           .END    START
```

ANSWER SHEET: PDP-11 ASSEMBLY LANGUAGE PROGRAMMING FINAL EXAMINATIO

1. ____________8
2. ____________8
3. ____________8
4. ____________8
5. ____________8
6. A B C D
7. A B C D
8. ____________8 ____________8
9. A B C D
10. T F
11. A B C D
12. A B C D
13. ____________8 ____________8
14. A B C D
15. ____________8
16. ____________8 ____________8
17. A B C D
18. ____________8
19. A B C D
20. A B C D
21. A B C D
22. A B C D

23. A B C D
24. A B C D
25. A B C D
26. A B C D
27. ____________8
28. ____________8
29. ____________8
30. ____________8
31. ____________
32. ____________8
33. A B C D
34. ____________8
35. ____________8
36. ____________8
37. ____________8
38. ____________8
39. A B C D
40. T F
41. T F
42. T F
43. T F
44. T F
45. T F

No se hizo [illegible]
[illegible]
[illegible]

Identify the number of the addressing modes of the instructions on the following lines:

1. Line 46 (destination)
2. Line 55 (source)
3. Line 61 (destination)
4. Line 62 (source)
5. Line 63 (destination)

MODE	GPR	
6	7	relativo
2	1	autoincremento
0	4	registro
[illegible]	[illegible]	inmediato
3	7	absoluto

6. After line 19 is executed

A. the symbol TICK = 177704_8.

B. the contents of location 1102_8=177704_8.

C. overflow occurs since a negative number is created.

D. the contents of location 100 = $-6Ø_{10}$.

7. The instruction at location 1022 MOV #-60., TICK places what number into memory location 1102?

A. 000052

B. 001022

C. 012767

D. 177704

8. For the instruction at line 20, the DEA is calculated as follows:

```
(R7)  = 001034
+X    = + 0050____8     (twos compliment form)
DEA   = + [illegible]____8     (both must be correct)
```

9. The instruction at location 1042 MOV #77, RØ will display what in the front panel data lights? ●=On O=Off

A. ● 0●0 ●0● 0●0 ●0● 0●0
B. 0 000 00● 000 ●00 0●0
C. 0 000 000 000 ●●● ●●●
D. ● ●●● ●●● ●●● 000 000

10. Given second = 000000 and DELAY = 00012, the next executed instruction (after line 23, 24) is taken from line 3∅.
(T/F) ____F____

11. Given that the contents of R∅=176000 before lines 25 thru 27 are executed. What are the contents of R∅ after execution?

A. 077000_8

B. 174000_8

[✓ C. 174001_8]

D. Indeterminate, as the console data lights are rotating cyclically every time an asynchronous clock interrupt occurs.

12. The "background" program will cause the bits presented in the data lights to "move". They will appear to

[✓ A. move to the left.]

B. move to the right.

C. flicker (random on/off).

D. alternate (bits on - then bits off).

13. For the instruction at line 32, the DEA is calculated as follows:

```
   (PC)         =       001102
+2x offset      =     + __177676__8  (twos compliment form)
   DEA                  __[illegible]__8  (both must be correct)
```

14. What purpose does line 34 serve?

A. As a HALT instruction (OPCODE=000000) after the main program
B. Creates a symbol at assembly time and equates it to zero
[✓ C. Defines a label at assembly time and equates it to location 1102]
D. Initializes memory location 1102_8 to 000000 every time the program is executed.

15. For this program, what is the upper stack limit?

____1000 see line 4____

16. After the Jump to Subroutine instruction at line 15 is executed,

1. the contents of R5: [R5] = __1010__ $_8$.

2. the contents of SP: [R6] = __776__ $_8$.

17. In the subroutine to print a message, after R1 receives the number at memory location 1010, R1 is then used as

A. an accumulator (operand register).

B. a counter.

C. an index.

[✓ D. a pointer.]

18. During program initialization, a message is printed out by the "MESAGE" subroutine (line #49). One argument is passed. What is the value of the argument?

____1110____$_8$

19. In the message subroutine (line #49), which instruction picks up the argument?

A. TPB = 177566

[✓ B. MOV (R5)+, R1]

C. MOVB (R1)+, @# TPB

D. RTS R5

20. If the contents of R1 = 001110_8, the instructions on lines 53 through 56 will

A. cause a teleprinter/punch interrupt to occur.
B. output a carriage return on the teleprinter/punch.
C. output a 4 digit decimal number on the teleprinter/punch.
[✓ D. Output a carriage return, line feed, and prompting message on the teleprinter/punch.]

21. The return from subroutine instruction on line 57 will

A. trap through location 4.

B. return to main program at line 15.

C. return to main program at line 16.

[✓ D. return to main program at line 17.]

22. Given that line 63 has just been executed and nothing has been typed on the terminal, which line will be executed next?

A. 63

[✓ B. 64]

C. 65

D. 66

23. During program initialization, a decimal number is accepted by the "INPUT" subroutine. How many arguments are passed when the subroutine is called?

A. none

B. one

C. two

D. four

24. In the subroutine to input a decimal number, GPR R3 is used as

A. an accumulator (operand register).

[✓ B. a counter.]

C. an index.

D. a pointer.

25. In the subroutine to input a decimal number, R4 is used as

[✓ A. an accumulator.]

B. a counter.

C. an index.

D. a pointer.

26. After the numeral 1 has been typed on the console keyboard and lines 71 through 73 have been executed, what will R2 contain?

[✓ A. 000001_8]

B. 000061_8

C. 000261_8

D. 177761_8

After the numeral 1 has been typed on the console keyboard and lines 61-76 have been executed, what will be the contents of the following registers:

27. R2 ___1___$_8$

28. R3 ___2___$_8$

29. R4 ___1___$_8$

38

30. If R4 = 000001 and the numeral Ø has been typed, what will be the contents of R4 after lines 63-76 have been executed?

R4 ___1?___$_8$

31. LKCSR (line #13) is the 16 bit register for which device, option or CPU function?

[illegible]

32. Interrupts from the device at UNIBUS address 177546 (line #17) will trap to what low memory address?

100 ₈

33. Line 21 will

A. enable clock interrupt.

B. disable clock interrupt.

C. reset the line clock to time 000000.

D. cause a line clock to occur.

If a clock interrupt request is granted by the CPU and an interrupt sequence is executed after the instruction at line 28, then what are the contents of

34. PC = 1160 ₈

35. SP = 774 ₈

36. PSW = 300 ₈

37. "1st item on the stack" 1044 ₈
-(SP)

38. When an interrupt sequence occurs, causing a vector to the LKINT (line #42) interrupt service routine (I.S.R.), a new PSW will be supplied from absolute memory location 102. What level will the processor priority be raised to during the I.S.R.?

6 ₈

39. The clock interrupt (line #41) subroutine will output which character once each second?

A. TPB
B. TICK
C. Bz
✓D. BELL

Provided this program has been started from the beginning and the operator typed 0010 on the console keyboard, indicate which of the following are T-True, or F-False.

40. F The computer halts immediately at location 1076 (immediately = within 1 millisecond).

41. F The clock handler executes once per second.

42. V The computer halts at location 1076_8 after 10 seconds.

43. V The terminal beeps every second for 10 seconds.

44. V The data lights (11/45-70) appear to rotate left after every clock tick.

45. F The elapsed time is printed when any key is struck on the console keyboard.

33

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

PRACTICAS PARA EL LABORATORIO

ELABORADAS POR:
ALBERTO TEMPLOS CARBAJAL

OCTUBRE, 1984

PRACTICA # 1

INTRODUCCION A LA MINICOMPUTADORA PDP11/40

PRACTICA # 1

INTRODUCCION A LA PDP11/40

OBJETIVO:

El alumno conocera de manera general el diagrama de bloques basico, de la configuracion de la PDP11/40, asi como algunas caracteristicas del equipo en forma individual.

Ademas el alumno conocera y aprendera a manejar algunos programas importantes del sistema, como son: el sistema de ayuda en linea SOS, el editor de lineas EDI, el programa de intercambio periferico PIP, la interfase para comunicacion con el sistema operativo MCR, y la secuencia a seguir para la ejecucion de un programa en la PDP11/40 bajo el sistema operativo RSX-11M(sistema en tiempo real multiusuario).

DESARROLLO:

1) Explicacion del equipo y algunas de sus caracteristicas.
2) Formas de entrar y salir de sesion.
3) Introduccion a la interfase de comunicacon MCR.
4) Explicacion y manejo del sistema de ayuda en linea SOS.
5) Explicacion y manejo del programa editor de lineas EDI.
6) Explicacion y manejo del programa de intercambio periferico PIP.
7) Explicacion de la secuencia a seguir para la ejecucion de un programa.
8) Ejecucion de un programa.

" EQUIPO Y CARACTERISTICAS "

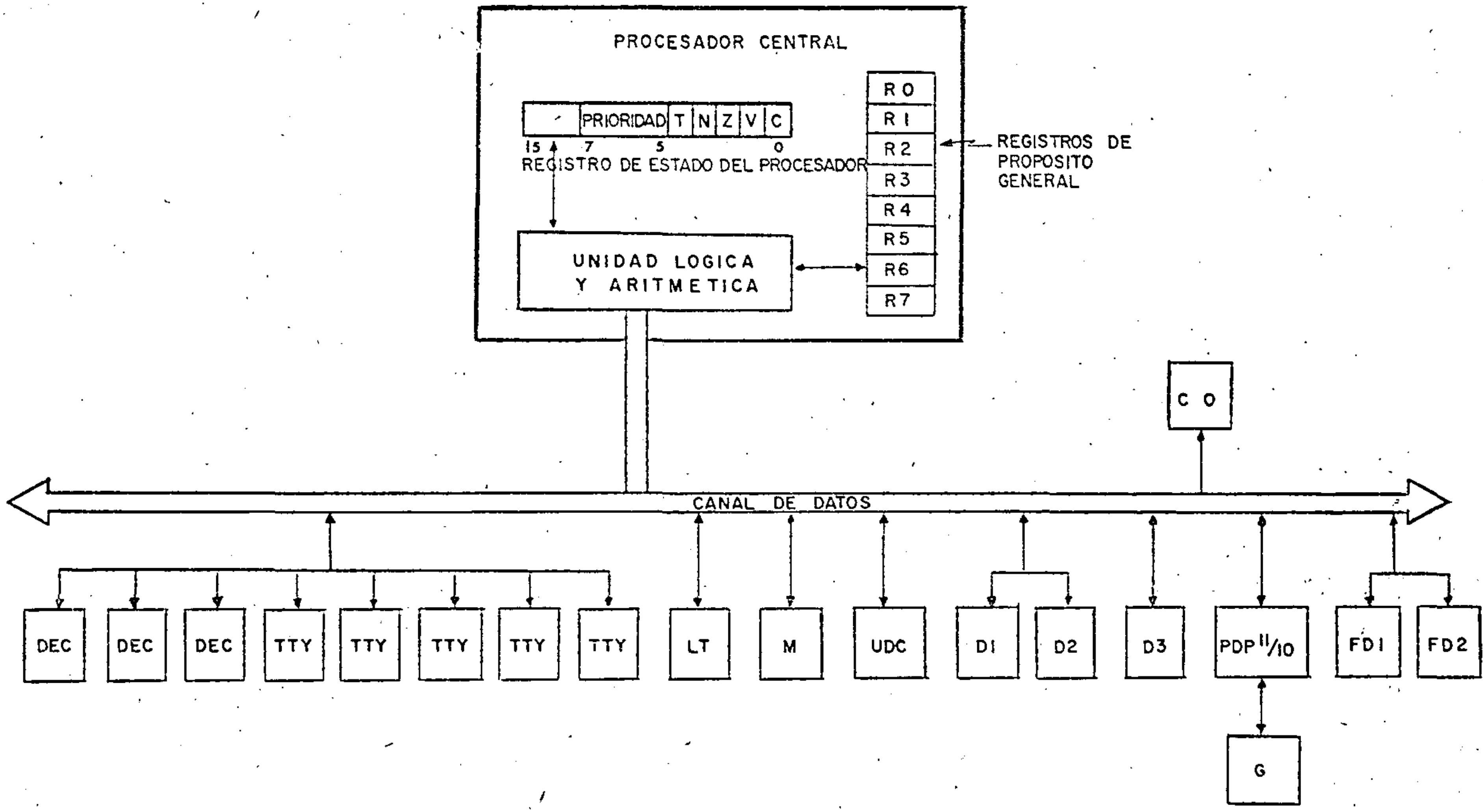


DIAGRAMA DE BLOQUES DEL SISTEMA

LISTA DE DISPOSITIVOS

DEC	- Decwriter LA36
TTY	- Terminal de video Hazeltine 1421
LT	- Lectora de Tarjetas CR-11
M	- Memoria principal
UDC	- Controlador Universal digital UDC11
D1,D2	- Discos RK05, unidad dual
D3	- Disco RK07
FD1,FD2	- Floppy Disk, unidad dual RX01
CO	- Consola del operador, decwriter LA36
G	- Graficador de pantalla VT11

Procesador central	PDP11/10	PDP11/40
Mercado principal	Usuario final	Usuario final
- Memoria	Ferritas	Ferritas
- Transferencia registro a registro	2.7 µs	0.9 µs
- Tamano maximo de memoria (palabras)	28K	124K
- Espacio maximo de direccionamiento	32K	128K
- Registros de proposito general	8	8
- Procesamiento de stack	si	si
- Microprogramado	si	si
- Instrucciones	Conjunto basico	Conjunto basico + XOR, SOB, MARK, SXT, RTT
- Aritmetica extendida (hardware)	Opcional (externa)	Opcional (interna) MUL, DIV, ASH, ASHC
- Punto flotante	Unicamente software	Opcion de hardware palabras de 32 bits
- Direccionamiento limite del stack	400 (fijo)	400 o programable (opcion)
- Administrador de memoria	No disponible	Opcion MFPI, MTPI
- Modos	1	1 std; 2 opt
- Prioridad de interrupcion automatica	4-lineas multi-nivel	4-lineas multi-nivel
- Autorestauracion del sistema cuando ocurre una falla de alimentacion	standard	standard

CARACTERISTICAS DEL DISCO RK05

CARACTERISTICAS	ESPECIFICACIONES
Cabezas magneticas	2
Densidad de grabado y formateo	
Densidad	2200 bpi maximo
Pistas	406
Cilindros	203(de 2 pistas c/u)
Sectores (registros)	4872(12 por revolucion)/ 6496(16 por revolucion)
Capacidades en bits(no formateado)	
Por disco	25 millones
Por pulgada	2040
Por cilindro	115200
Por pista	57600
Por sector	4800/3844
Tiempo de acceso	
Rotacion del disco	1500+-30 rpm
Retardo promedio	20ms(rotacion media)
Posicionamiento de la cabeza (incluyendo tiempo de asentamiento)	10ms-para pistas adyacentes 50ms-promedio 85ms-para el movimiento de 200 pistas
Transferencia de bits	
Codigo de transferencia	doble frecuencia(codigo NRZ)
Promedio de transferencia	1.44Mbits por segundo
Temperatura ambiente	De 10 a 43 grados centigrados

CARACTERISTICAS DEL DISCO RK07

CARACTERISTICAS	ESPECIFICACIONES	
Cabezas magneticas	3 de lectura/escritura y una de servo.	
Capacidad de grabado (formateado)	Palabra de 18bits	Palabra de 16bits
Cilindros/Cartucho	815	815
Pistas/Cilindro	3	3
Pistas/Cartucho	2445	2445
Sectores/Pista	20	22
Palabras/Sector	256	256
Bits/Palabra	18	16
Bits/Sector	4608	4096
Bits/Superficie	73.43M	75.11M
Bits/Pack	220.32M	225.33M
Bits/Pulgada	4040	4040
Pistas/Pulgada	384.6	384.6
Promedio de transferencia de bits	4.30 M/s	
Bit cell width	232.5 ns	
Frecuencia rotacional	2400 rpm+-2.5%	
Valor promedio	12.5ms(rotacion media)+-2.5%	
Valor maximo	25.0ms	
Tiempo de busqueda		
Valor promedio	36.5 ms	
Valor maximo	71.0 ms	
Temperatura ambiente	De 16 a 49 grados centigrados	

CARACTERISTICAS DEL FLOPPY DISK RX01

CARACTERISTICAS	ESPECIFICACIONES	
Capacidad	8-bit(byte)	12-bits(palabra)
Por diskette	256256	128128
Por pista	3328	1664
Por sector	128	64
Promedio de transferencia de datos		
De diskette a buffer del controlador	4 μs/data bit(250Kbps)	
De buffer a interfase del CPU	2 μs/bit (500Kbps)	
Interfase del CPU al bus de I/O	18 μs/byte(>50Kbytes/s)	
Movimiento pista a pista	10ms/pista maxima	
Tiempo de asentamiento de la cabeza	20ms maximo	
Velocidad rotacional	360 rpm+-2.5%;166ms/rev nom.	
Grabado de superficies por disco	1	
Pistas por disco	77 (0-76)	
Sectores por pista	26 (1-26)	
Tecnica de grabado	doble frecuencia	
Densidad en bits	3200 bpi	
Densidad en pistas	48pistas/pulgada	
Tiempo promedio de acceso	488 ms,calculado como sigue:	

$$\overset{\text{busqueda}}{(77\text{pistas}/2) \times 10\text{ms}} + \overset{\text{asentamiento}}{20\text{ms}} + \overset{\text{rotacion}}{(\ 166\text{ms}/2\ )} = 488\text{ms}$$

Temperatura ambiente de 15 a 32 grados centigrados

CARACTERISTICAS DEL DECWRITER LA36

- Impresion
  Se tienen switches seleccionables para 10,15,o 30 caracteres/seg

- Longitud de la linea
  132 caracteres maximo

- Espaciamiento
  10 caracteres/pulgada (horizontal)
  6 lineas/pulgada (vertical)

- Caracteres
  96 caracteres ASCII
  Matriz de puntos de 7x5 (0.07x0.10")(1.77x2.54mm)

- Teclado
  Standard ANSI

- Interfase
  EIA/CCITT

- Modos de transmision
  halfduplex o full duplex

- Porcentaje de transmision y recepcion de caracteres

SWITCH	PORCENTAJE DE CARACTERES
110	10 (caracteres/segundo)
300	30 (caracteres/segundo)
110 y 300	15 (caracteres/segundo)

- Temperatura ambiente
  De 10 a 40 grados centigrados

CARACTERISTICAS DE LA LECTORA DE TARJETAS CR-11

- Medio de entrada
  Tarjetas perforadas de 80 columnas

- Velocidad
  285 tarjetas/minuto

- Capacidad del "hopper"
  550 tarjetas

- Temperatura ambiente
  15 a 32 grados centigrados

CARACTERISTICAS DE LA TERMINAL HAZELTINE 1421

- Tamano de la pantalla
  diagonal de 30.5cm, es de fosforo

- Capacidad
  80 caracteres/linea x 24 lineas (1920 caracteres)

- Formato del caracter
  Matriz de puntos de 5x8 en una ventana de puntos de 7x10

- ConJunto de caracteres
  95 caracteres ASCII desplegables.
  Los 128 caracteres ASCII pueden ser tecleados y transmitidos.

- Display
  Blanco sobre fondo negro, dos intensidades

- PorcentaJe de refresco
  60 Hz

- Standard TV
  260 lineas/marco, 240 lineas desplegadas

- Memoria
  Memoria de acceso aleatorio(RAM) de 2048x8

- Interfase
  EIA RS-232C

- Modos de transmision
  half duplex y full duplex

- Temperatura de operacion
  de 10 a 40 grados centigrados

SISTEMA UDC

INTRODUCCION

El uso de las computadoras digitales ha permitido la automatizacion de los procesos industriales, en mayor o menor escala, dependiendo de las necesidades y los recursos de cada usuario.

La familia de computadoras PDP11 presenta un sistema periferico que se encarga de interconectar los dispositivos de las instalaciones con la computadora; este sistema periferico recibe el nombre de subsistema de control digital universal de captura de datos industriales (UDC).

El UDC tiene una construccion modular que permite la rapida reacomodacion o reconfiguracion de todo el sistema.

Basicamente el UDC es un control de trafico que permite la utilizacion de programas de control o de monitoreo, instalados en la memoria de la computadora.

El UDC tiene la capacidad de tener entradas combinadas ya sean digitales o bien analogicas, debido a la construccion de los modulos que constituyen el subsistema, presenta una alta inmunidad al ruido, las conexiones que hacen falta se basan principalmente en conectores de dos terminales, lo cual puede facilitar el cableado; posee ademas la caracteristica de tener niveles de respuesta inmediatos, o bien diferidos. Ademas posee un soporte de software que permite la rapida implementacion de todos los programas que se requieran para las actividades a que se tengan destinadas las operaciones del sistema.

Otra de las caracteristicas que se consideran importantes es la de suprecion de circuitos especiales de tierra de los aparatos de campo con la computadora, estacaracteristica da mas flexibilidad a el equipo instalado.

Como en algunas practicas haremos uso de los convertidores A/D y D/A, a continuacion daremos algunas de sus caracteristicas. Sin embargo no son los unicos componentes del sistema UDC.

SISTEMA DE ENTRADAS ANALOGICAS ADU01

(convertidores analogico/digital)

Especificaciones funcionales

- Resolucion	11 bits + signo	
- Canales multiplexados	se pueden seleccionar 8 canales	
- Ancho de banda del canal	1.5 Hz	
- Promedio de muestreo incluyendo conversion A/D	4K muestras/s (maximo) 20 mustras/s en el mismo canal	
- Rechazo modo comun	DC a 60 Hz, 100 dB minimo	
- Rechazo modo normal	50 dB a 60 Hz	
- Exactitud total del sistema	peor caso <+-(0.11% de la escala completa + 15 v)	
- Rangos de las senales de entrada analogica	senal de entrada	ganancia programada
	+-1 mA o +-10 mV	1000
	+-5 mA o +-50 MV	200
	+-10 mA o +-100 mV	100
	+-20 mA o +-200 mV	50
	+-50 mA o +-500 mV	20
	+-62 mA o +-1.0 v	10
	+-62 mA o +-5.0v	2
	+-62 mA o +-10v	1
- Voltaje maximo de entrada	senal +-10v + modo comun +-12v	
- Salida digital	0.0v= 0000 escala completa + = 77760 escala completa - = 100000	
- Interrupcion	inmediata, diferida, o sin interrupcion	
- Requerimientos de potencia	+5v con 1.75A	

GRAFICADOR VT11

Instalaciones de este tipo tienen un gran rango de aplicaciones, entre las cuales se incluyen estudios de simulacion, diseno con ayuda de computadora y adquisicion de datos en tiempo real.

Consiste de un procesador de desplegados. El procesador puede estar conectado al UNIBUS y funcionar como un procesador autonomo para manejar las instrucciones de graficacion. Otra alternativa es usar este subsistema como una terminal inteligente en un sistema de graficacion con multiprocesamiento. La primera forma es llamada Stand-Alone y la segunda Host-Satellite.

Se pueden desplegar diferentes tipos de elementos: puntos, segmentos de lineas, caracteres y graficos. Estos elementos estan normalmente definidos en posiciones de coordenadas relativas a la posicion actual del cursor, aunque tambien pueden ser definidos en posiciones absolutas. Los segmentos de lineas o vectores pueden ser dibujados en cualquiera de los siguientes formatos:

- linea continua

- linea larga discontinua

- linea corta discontinua

- linea discontinua con punto

La capacidad del area principal de desplegados es de 73 caracteres de tamano normal por linea y 31 lineas por pantalla.

Esta area principal es de 9 1/4"x9 1/4" y direcciona 1024x1024 puntos identificados cada uno por sus respectivas coordenas (x,y).

Se puede variar la brillantes de los dibujos o parte de estos, para lo cual existen 8 niveles de intensidad. Tambien puede especificarse que un desplegado o parte de este flashee.

El conjunto de caracteres disponibles son 96 caracteres ASCII convencionales y ademas otros 31 caracteres que incluyen letras del alfabeto griego y simbolos matematicos. Estos pueden desplegarse con el tipo normal o en tipo italico.

Uno de los principales soportes del sistema de graficacion es la pluma electronica, la cual permite interactuar con el procesador de desplegados para seleccionar opciones de un menu determinado, identificar imagenes a ser movidas; o manipular desplegados en la pantalla. Ademas de estas caracteristicas existen otras que hacen muy potente al graficador.

* FORMAS DE ENTRAR Y SALIR DE SESION *

Para entrar en sesion en la computadora PDP11/40 se debera hacer lo siguiente:

1) Dar:

>HELLO <cr> ; tambien se acepta la forma abreviada HEL
la computadora responde:

ACCOUNT OR NAME: xxx <cr> ; contestamos xxx y <cr>
ahora la computadora nos pide una contrasena

PASSWORD: yyy <cr> ; se contesta la contrasena(esta no aparece en la terminal) y se da <cr>

por ultimo la computadora responde con un prompt ">", indicandonos que esta lista para aceptar cualquier comando de MCR.

xxx - representa la cuenta(clave), o bien un nombre.
formas validas de xxx son:

a) [006,001] o [6,1] ; el uso de los parentesis es opcional
b) 6,1 o 6/1
c) un nombre valido(como clave) de 1 a 9 caracteres

yyy - representa un password o contrasena(valida en la computadora) compuesta de 1 a 6 caracteres.

Generalmente cada cuenta o clave tiene asociados uno o dos nombres(puede no tenerlos), que permiten la misma posibilidad de acceso, y una sola contrasena.

2) O bien en forma abreviada dar:

>HELLO xxx/yyy

o bien

>HEL xxx/yyy

donde xxx y yyy aceptan las formas explicadas anteriormente.

La diferencia entre las formas 6,1 y 6/1 es la siguiente: con 6,1 se entra en sesion y se despliega en la terminal del usuario el archivo LOGIN.TXT, que contiene normalmente alguna informacion util para el usuario; con 6/1 se entra en sesion y el archivo LOGIN.TXT no es desplegado. Esta forma es utilizada para abreviar el tiempo de entrada a una sesion de computadora.

Para salir y terminar una sesion se da el comando BYE.

```
>BYE
y la computadora responde:
HAVE A GOOD AFTERNOON
21-SEP-84 16:48 TT3: LOGGED OFF
```

NOTA: Existe un comando que es aceptado sin entrar en sesion, este es, HELP, con el que se proporciona informacion para que el usuario pueda entrar en sesion.

* INTERFASE DE COMUNICACION MCR *

(MONITOR CONTROL ROUTINE)

INTERFASE MCR (MONITOR CONTROL ROUTINE)

La interfase MCR es la que permite la comunicacion con el sistema operativo RSX-11M, el usuario a traves de la terminal introduce los comandos que seran interpretados y ejecutados por el MCR.

Funciones que permite el MCR:

- Inicializar el sistema.

- Manejo de dispositivos perifericos.

- Control de ejecucion de tareas

- Obtener informacion del sistema y de las tareas.

- Etc.

Algunos de los comandos mas usados de MCR son los siguientes:

ABORT - permite abortar una tarea y terminar con la ejecucion de esta.

ACTIVE - despliega los nombres de las tareas activas en la terminal del usuario.

BROADCAST - por medio de este comando podemos enviar mensajes a una o mas terminales.

BOOT - permite cargar el sistema y transferirle el control.

BYE - termina la sesion con el usuario.

DEVICES - despliega los nombres simbolicos de todos los dispositivos reconocidos por el sistema.

HELLO - permite iniciar una sesion con el usuario.

HELP - despliega el contenido del archivo HELP.TXT. Regularmente es una ayuda para el usuario.

INSTALL - hace que una tarea sea conocida por el sistema y la pone en estado "dormat" hasta que el ejecutivo reciba una peticion para ejecutar esta.

LOAD - permite cargar el manejador de un dispositivo no residente.

PARTITIONS - despliega en la terminal del usuario una descripcion de cada una de las particiones de memoria en el sistema.

REMOVE - borra del directorio de tareas el nombre de la tarea para hacerla desconocida al sistema.

RUN - permite la ejecucion de una tarea.

SET - este comando permite alterar algunas condiciones del sistema y ademas las caracteristicas locales de una terminal.

TASK-LIST - despliega el nombre de todas las tareas instaladas en el sistema.

UFD - Este comando crea un User File Directory en un volumen de files-11 e introduce su nombre en el directorio maestro de archivos.

UNLOAD - Descarga el manejador de un dispositivo.

Todos los comandos descritos, a excepcion de HELP pueden ser invocados escribiendo solo sus 3 primeros caracteres.

Ademas se tienen algunas teclas de control, estas son: (para las teclas de control es necesario oprimir la tleca de la letra al mismo tiempo que la tecla "CTRL")

CTRL/C - Se obtiene la atencion del MCR, el cual es el - encargado de interpretar los comandos al sistema operativo, responde al CTRL/C de la siguiente manera: MCR>

CTRL/O - Descarta las salidas que se envian a la terminal, el sistema descarta dichas salidas hasta que se de CTRL/O por segunda vez.

CTRL/Q - Cuando en la terminal se desea detener algun
CTRL/S listado(salida) momentaneamente se da CTRL/S. El listado proseguira al dar CTRL/Q.

CTRL/R - Ejecuta el retorno de carro y reimprime la ultima linea para verificar si las correcciones hechas con DELETE estuvieron bien hechas.

CTRL/U - Borra la linea actual (donde se esta trabajando) y se ejecuta un <cr>.

CTRL/Z - Se usa en programas para regresarle el control a MCR.

Vease los ejemplos anexos.

Ejemplos de comandos de MCR

```
>HELLO  ; comando para entrar en sesion
ACCOUNT OR NAME: 6/1  ; esta es la clave
PASSWORD:

        RSX-11M BL22   MULTI-USER SYSTEM

GOOD MORNING
09-OCT-84 09:03 LOGGED ON TERMINAL TT11:

>ACT  ; comando para listar las tareas activas en la terminal
...MCR
...SYS
>


>TAS  ; COMANDO PARA LISTAR  LAS TAREAS ACTIVAS DEL  SISTEMA

. LDR.         LDR    248. 000000 LB0:-00000000 FIXED
TKTN   03.7    TKTPAR 248. 010000 LB0:-00003561
...DMO 03.1    GEN    160. 040000 LB0:-00006334
...MCR 02      SYSPAR 160. 010000 LB0:-00006537
...MOU 03.02   GEN    160. 040000 LB0:-00006045
...SYS 01      GEN    160. 012000 LB0:-00006561
F11ACP M0235   FCPPAR 149. 030000 LB0:-00006373
SHF... 03      SHFPAR 105. 010000 LB0:-00006233
...INI 03      GEN    100. 040000 LB0:-00006435
...INS 03      GEN    100. 040000 LB0:-00006472
...UFD V0407   GEN    100. 040000 LB0:-00006073
...AT. 04.17   GEN     65. 040000 LB0:-00005751
...EDI M11     GEN     65. 040000 LB0:-00004662
...SAV 03.9    SAVPAR  64. 040000 LB0:-00006736
...ERF 01      GEN     61. 040000 LB0:-00007152
...ACS 01      GEN     50. 040000 LB0:-00006205
...BOO 03.2    GEN     50. 040000 LB0:-00006310
...LOA 03      GEN     50. 040000 LB0:-00006767
...UNL 02      GEN     50. 040000 LB0:-00007023
...HEL 01.15   GEN     50. 040000 LB0:-00006111
...BRO V02.3   GEN     50. 040000 LB0:-00006144
...MAC M1110   GEN     50. 070000 LB0:-00004330
...TKB M29     GEN     50. 070000 LB0:-00004437
...PIP M1331   GEN     50. 040000 LB0:-00005174
...FOR M03     GEN     50. 070000 DM0:-00113771
...SAT 0736    GEN     50. 126000 DM0:-00123676
...SOS 0736    GEN     50. 020400 DM0:-00066110
...TTY 0736    GEN     50. 014400 DM0:-00010066
...BYE 01.6    GEN     50. 040000 DK0:-00006102
...PAS V1.1F   GEN     50. 121400 DK0:-00010364
ERRLOG 01      SAVPAR  40. 040000 LB0:-00007156
>
```

```
>DEV  ; comando para listar los dispositivos del sistema
UD0:
CR0:    UNLOADED
DK0:    MOUNTED
DK1:    MOUNTED
DM0:    MOUNTED LOADED
DX0:    LOADED
DX1:    LOADED
TT0:    [6,1]    - LOGGED ON
TT1:
TT2:    [6,1]    - LOGGED ON
TT3:
TT4:
TT5:
TT6:
TT7:
TT10:
TT11:   [6,1]    - LOGGED ON
NL0:
TI0:
CO0:    TT0:
CL0:    TT0:
LB0:    DK0:
SY0:    DK0:
>
```

```
>PAR  ; comando para listar las particiones de memoria en el sistema
LDR    000000 000000 MAIN TASK
DRVPAR 100000 010000 MAIN SYS
       100000 002700 SUB  DRIVER -DM:
       102700 001200 SUB  DRIVER -DX:
FCPPAR 110000 030000 MAIN SYS
       110000 030000 SUB  (F11ACP)
SYSPAR 140000 010000 MAIN TASK
TKTPAR 150000 010000 MAIN TASK
SHFPAR 160000 010000 MAIN TASK
FCSRES 170000 020000 MAIN COM
PAR20K 210000 100000 MAIN TASK
SAVPAR 210000 040000 SUB  TASK
PAR4K  250000 020000 SUB  TASK
PAR2K  270000 010000 SUB  TASK
TOTCOM 310000 020000 MAIN SYS
GEN    330000 310000 MAIN SYS
       330000 012000 SUB  (...SYS)
UDCOM  771000 001000 MAIN DEV
>
```

```
>TIM  ; despliega el dia, la hora, minutos y segundos
09:15:31 09-OCT-84
>


>SET /UIC  ; nos indica en que clave estamos
UIC=[6,1]
>


>SET /BUF=TT6:132.  ; asigna un buffer de 132 caracteres a la terminal tt6
>


>LOA CR:  ; CARGA LA LECTORA DE TARJETAS
LOA -- SYNTAX ERROR
>


>LOA CR:  ; carga la lectora de tarjetas
>


>INS [6,6]GREP/TASK=...GRP  ; instala la tarea GREP en la cuenta [6,6] con
                               el nombre GRP(nombre de ejecucion), para que
                               todos los usuarios la utilicen.
>


>BYE  ; comando para salir de sesion
>
HAVE A GOOD MORNING
09-OCT-84 09:22 TT11: LOGGED OFF
>
```

" MANEJO DEL SISTEMA DE AYUDA EN LINEA SOS "

SISTEMA DE AYUDA EN LINEA "SOS"

SOS es un sistema de ayuda en linea, que permite al usuario investigar el funcionamiento, uso, sintaxis, etc. De algunos comandos y subsistemas del sistema operativo RSX-11M.

Existen dos modos de operacion del SOS:

1) Modo Prompt.

2) Modo Inmediato.

1) El modo prompt se establece cuando se proporciona el siguiente comando:

```
SOS <cr>
```

Donde <cr> significa oprimir la tecla de RETURN. Inmediatamente el sistema respondera con SOS>, en espera de algun comando. Una vez ejecutado el comando, regresa a pedir otro. Si ya no se desea consultar alguna otra informacion se da un <cr> como comando, para regresar el control al MCR.

2) El modo inmediato se establece de la siguiente forma:

```
SOS comando <cr>
```

SOS regresa el control al MCR cuando termina de ejecutar el comando.

Un comando esta compuesto de:

```
--------------------------------------
|     TOPICO     |    SUBTOPICO      |
--------------------------------------
```

Donde cualquiera de los dos puede sustituirse por un signo de interrogacion. Por ejemplo:

```
>SOS ?
```

En este caso, se desplegara una lista de los topicos que pueden ser consultados.

```
>SOS MCR ?
```

En este caso, MCR es el topico y la interrogacion permite ver la lista de los subtopicos de MCR.

```
>SOS MCR RUN
```

En este caso, se despliega la informacion del subtopico RUN del comando MCR. Ver ejemplos anexos.

Ejemplos del sistema SOS

```
>SOS ?
SOS                    PIP                    C
MCR                    QIO                    EDI
TOOLS                  TOT                    MAC
NOT                    PASCAL                 FORTRAN
AYUDATOT               CC
sos>
>


>SOS EDI ?

        ADD             BOTTOM          CHANGE          DELETE
        EXIT            INSERT          LOCATE          NEXT
        RENEW           RETYPE          TOP             PASTE
        MACRO           SAVE            UNSAVE          SIZE
        READ            PRINT           TYPE            UPPER
        VERIFY          ------          ------          ------


 SOS NOT

----------------------------------------------------------------------

    La persona que haga uso del equipo y no se anote en la lista,
    que para tal efecto se encuentra en el laboratorio, le sera
    suspendido el servicio.
                              ATTE. Lab. de Computacion...

----------------------------------------------------------------------
>


>SOS MCR RUN

                 RUN     - permite la ejecucion de una tarea

>
```

* MANEJO DEL PROGRAMA EDITOR DE LINEAS EDI *

EDITOR DE TEXTOS POR LINEA (EDI)

El editor de textos es un programa del sistema operativo que tiene por objeto la creacion y/o modificacion de archivos de caracteres alfanumericos.

Esto se logra mediante comandos que el usuario introduce desde su terminal, permitiendo insertar, borrar o corregir a dicho archivo.

Los comandos de EDI actuan sobre un apuntador que se mueve por lineas, dependiendo de los comandos que se introduzcan.

Con el editor EDI se lee un bloque del archivo de entrada en un buffer(area de almacenamiento temporal) y el apuntador se mueve en las lineas de dicho bloque.

Cuando se termina la sesion de edicion o se lee otro bloque del archivo de entrada, se lleva a cabo automaticamente la escritura al archivo de salida.

Un bloque contiene 38 lineas, para referirse a otro bloque se tiene que leer este del archivo de entrada con algun comando del editor.

A la linea donde se encuentra el apuntador se le llama "linea actual", analogamente, al bloque en el cual estamos trabajando se le llama "bloque actual".

Caracteristicas para el uso eficiente del editor:

1) El editor trabaja en dos modos de operacion:

a) Modo de entrada (input mode). En este modo se puede escribir el texto que debera contener el archivo. Este modo se establece automaticamente al solicitar un archivo que no existe.

b) Modo de edicion (edit mode). En este modo se pueden dar comandos de control al editor y se identifica por un asterisco que aparece en la pantalla solicitando un comando. Este modo se establece automaticamente al editar un archivo que ya existe.

Para pasar del modo de entrada al modo de edicion se oprime dos veces la tecla <cr>.

Para pasar del modo de edicion al modo de entrada se teclea en seguida del asterisco una letra "I" (para insertar) y se da un <cr>.

2) El editor tiene dos modos de accesar textos, estos son:

a) Por lineas.

b) Por bloques.

De estos dos metodos es mas conveniente trabajar con un buffer que almacena un cierto numero de lineas, sobre las cuales se pueden hacer modificaciones y en caso de ser un nuevo archivo, agregar las lineas que este contendra, por default el buffer contiene 38 lineas, pero puede ser modificado. Se entra automaticamente al modo por bloques al editar cualquier archivo.

3) El editor trabaja con archivos de entrada y archivos de salida.

El archivo de entrada es aquel que toma el editor como fuente de texto, pasando un numero de lineas de este al buffer.

El archivo de salida es aquel que usa el editor para almacenar el contenido del buffer y este pueda ser utilizado nuevamente al finalizar una sesion de edicion , este archivo es el mismo que el de entrada y contiene todas las modificaciones hechas a este; es guardado como una nueva version, por lo cual es importante eliminarla con "KILL" al finalizar la edicion, o bien, depurar aquellas versiones que ya no sean utiles.

- Como llamar al editor ?

Existen dos formas de llamar al editor:

1) Modo prompt.
2) Modo inmediato.

Para editar un programa deberemos hacerlo de la siguiente forma:

```
>EDI {dev:[uic]}nombre.extension{;version}
```

donde:

dev: - es el dispositivo donde se encuentra el archivo.
[uic] - es el directorio del usuario.
nombre - es el nombre con el cual llamaremos a nuestro archivo, y debera estar formado por caracteres alfanumericos(de 1 hasta 9).

extension - esta compuesta por 3 caracteres alfanumericos y representa el tipo de algun archivo.

Los tipos comunmente usados son:

FTN - fortran
MAC - macro o ensamblador
DBG - para definir esquemas a la base de datos
PAS - pascal
TXT - archivo de texto
DAT - archivo de datos

Sin embargo podemos tener cualquier extension que contenga de 1 a 3 caracteres alfanumericos.

version - es un numero que le asigna el sistema, si se crea por vez primera se le asigna el numero 1 y luego este se ira incrementando en forma octal, si el archivo es editado.

Las partes entre {} son opcionales.

Si el archivo especificado es un nuevo archivo(es decir, el archivo no fue encontrado en el dispositivo especificado), el editor asumira que se quiere crear un archivo con ese nombre y entonces EDI imprimira:

```
[CREATING NEW FILE]
INPUT
```

con lo cual estaremos en modo de entrada.

Si se especifica un nombre de archivo ya existente, entonces EDI respondera de la siguiente forma:

```
[000nn LINES READ IN]
[PAGE p]
*
```

El asterisco indica que estamos en modo editor, en el cual se podran usar los comandos de EDI. Estas lineas indican que automaticamente EDI leyo un bloque del archivo de entrada colocandolo en el buffer.

Las "nn" nos indican el numero de lineas leidas y la "p" el numero de pagina en que se encuentra.

- Como salir del editor y salvar el programa ?

Existen 3 formas de llevar a cabo esto:

Estando en modo de edicion en seguida del asterisco teclear:

1) EXIT o bien EX

Con lo cual se cierran los archivos de entrada y salida, salvando el bloque actual; se termina la sesion de edicion y se crea una nueva version con las modificaciones hechas.

2) Oprimir simultaneamente las teclas CTRL y Z

Funciona en forma igual a EXIT

3) EDX o bien ED -- EXIT DELETE

Funciona en la misma forma que EXIT y CTRL/Z solo que la version anterior es borrada.

- Como recorrer las lineas de texto para localizar una en especial y hacer correcciones ?

Caso 1) Si se esta editando un nuevo texto y se desea regresar una o mas lineas.

Primero pasar a modo de edicion(dar 2 <cr>), despues por cada vez que se oprima la tecla <esc> se retrocedera una linea de texto y por cada vez que se oprima la tecla de <cr> se avanzara una linea de texto.

nota: el comando END posiciona el apuntador en la ultima linea de texto, que se encuentra en el buffer.

Caso 2) Teniendose un archivo ya creado se desea localizar una linea especifica.

Primero verificar que se este en modo de edicion. Despues teclear el comando PL CADENA; donde PL (PAGE LOCATE) es el comando para localizar por pagina o bloque, y CADENA es el conjunto de caracteres que se desea localizar. Despues de esto el apuntador se ha posicionado en el bloque en donde se encuentra la linea, siendo esta la actual, asi como el bloque en el buffer es el actual.

- Que hacer cuando se recibe el mensaje de que se ha llenado el buffer EOB ?

- Cuando este ocurre es necesario pasar el contenido del buffer al archivo de salida, para que quede listo para recibir mas texto, para lo cual basta con teclear el comando siguiente en modo de edicion:

*RENEW o bien REN

nota: cuando se trata de un archivo nuevo se recibe el mensaje de que no hay archivo de entrada abierto, ya que el comando, una vez limpio el buffer busca un archivo de entrada para traer el bloque siguiente.

- Como incrementar la longitud del buffer ?

Basta con teclear el comando siguiente, despues del asterisco en modo de edicion:

```
*SIZE n
```

donde "n" es el numero de lineas que se desea contenga el buffer.

OTROS COMANDOS DE EDITOR SON:

ADD

Este comando agrega al final de la linea una cadena de caracteres, ejemplo:

```
* linea de prueba
*A del editor.  <cr>
*
```

esta linea quedara como: linea de prueba del editor "pero no sera desplegada"

ADD and PRINT

La funcion es la misma que la del ADD excepto que la linea resultante si es desplegada, ejemplo:

```
* linea de prueba
*AP del editor.  <cr>
linea de  prueba del editor.
*
```

BOTTOM

Mueve el apuntador al final del bloque.

```
*BO <cr>
```

Si la opcion de VERIFY ON esta activa, se desplegara el contenido de la ultima linea del bloque.

```
*V ON <cr>
*BO <cr>
ultima linea
*
```

CHANGE

Cambia la CADENA1 por la CADENA2 en una linea, si la CADENA1 aparece en la linea. Si la CADENA1 es nula, la CADENA2 es insertada al inicio de la linea. Si la CADENA2 es nula la CADENA1 es borrada de la linea. Para localizar la CADENA1 se barre la linea desde el principio, hasta encontrarla.

Los caracteres para delimitar las cadenas son normalmente caracteres especiales que no aparecen en la linea. Es muy usual que se use una diagonal como delimitador. Ejemplos:

```
* hola como estas
*C/hola/que <cr>
que como estas
*
```

Es posible realizar "n" cambios sobre una linea con:

```
*nC/CADENA1/CADENA2
```

Ejemplo:

```
*
quee pasa, quee te ocurre
*2C/quee/que
que pasa, que te ocurre
*
```

DELETE

Elimina lineas de texto de la siguiente manera:

a) Si damos Dn, la linea actual y las siguientes n-1 son borradas del texto, siendo la linea nueva, la siguiente de la ultima borrada.

b) Si damos D-n, la linea actual no es borrada, pero si las n lineas que le preceden. La linea actual es la misma.

c) Si n es nulo la linea actual es borrada y la linea siguiente se vuelve la actual. Ejemplos:

```
*D-5 ; borra las 5 lineas anteriores a la actual

*D2  ; borra la linea actual y la siguiente

*D   ; borra la linea actual
```

DELETE and PRINT

Realiza la misma funcion que DELETE pero imprime la linea actual.

```
*DP-5 ; borra las 5 lineas anteriores a la actual,
        permaneciendo esta.

*DP3  ;  borra la linea actual y las dos siguientes,
         quedando la siguiente como la actual

*DP   ; borra la linea actual e imprime la siguiente, esta
        es ahora la actual
```

EXIT

Este comando transfiere todo el resto de lineas que existen en el buffer y el resto de lineas del archivo de entrada, al archivo de salida; cierra los archivos, y nombra al archivo de salida con una nueva version y termina la sesion de edicion. Ejemplos:

```
*EXIT nombre <cr> ; renombra al archivo de salida, este
           es el que contiene las modificaciones hechas.

*EXIT <cr> ; deja el mismo nombre con otra version
```

INSERT

Con este comando se insertan lineas inmediatamente despues de la linea actual. La linea insertada se vuelve la linea actual. Ejemplos:

```
*I CADENA DE CARACTERES <cr>
* ; inserta la linea y regresa a modo de edicion
*I <cr> ; pone al editor en modo de entrada
CADENA 1
CADENA 2
ETC. <cr>
<cr>
```

LOCATE

Localiza una cadena dentro del buffer, empezando la busqueda en la linea siguiente a la actual. El apuntador es colocado en la linea que contiene a la cadena y la despliega si VERIFY ON esta activo. Ejemplo:

```
*L CADENA <cr>
```

NEXT

Este comando mueve el apuntador hacia arriba o hacia abajo dentro del bloque.(no se considera la linea actual). Ejemplos:

```
*N-5 <cr> ; mueve el apuntador 5 lineas hacia arriba

*N5 <cr>  ; mueve el apuntador 5 lineas hacia abajo
```

NEXT and PRINT

Mueve el apuntador e imprime la nueva linea actual.(si considera la linea actual).Ejemplo:

```
*NP3  <cr>  ;  imprime  la  tercera  linea(despues de la
                actual) como actual
```

TYPE

Escribe desde la linea actual, hasta la linea n-1, sin mover el apuntador, o sea que la linea actual permanece siendo la misma despues de que se han desplegado las n-1 lineas. Ejemplo:

```
*ty 5 <cr>
```

RENEW

Este comando nos permite escribir el bloque actual al archivo de salida, y leer un nuevo bloque del archivo de entrada. Ejemplos:

```
*REN <cr> ; salva el bloque actual y lee uno nuevo

*REN 3 <cr> ; salva el bloque actual y lee 3 bloques del
    archivo de entrada, dejando en el buffer el ultimo leido.
```

RETYPE

Remplaza la linea actual por una nueva linea. Ejemplos:

```
*R NUEVA LINEA <cr> ; escribe linea nueva  en  la  linea
                      actual

*R <cr> ; borra la linea actual
```

TOP

Coloca el apuntador al inicio del bloque. Cuando se usa TOP se pueden agregar lineas que precedan a la primera del bloque. Ejemplo:

```
*T <cr> ; coloca el apuntador al inicio del bloque
```

PASTE

Busca en todo el bloque a partir de la linea actual la CADENA1 y la remplaza por la CADENA2. Ejemplo:

```
*PA/CADENA1/CADENA2 <cr>
```

MACRO

Este comando es usado para definir macros, existe espacio para definir 3 macros, llamadas 1, 2 y 3, y pueden contener cualquier comando legal del editor.

Definicion de una macro:

```
*MACRO 1 PA/ABC/XYZ/&REN <cr>
```

Con esto estamos definiendo una macro que cambie en todo el bloque ABC por XYZ y traiga un nuevo bloque.

Llamada a la macro:

```
*2M1 <cr> ; realiza la funcion macro 1 en dos bloques
```

SAVE

Con este comando se escriben a un archivo n lineas a partir de la linea actual, el archivo y el numero de lineas se especifican en el comando. Ejemplo:

```
*SAVE 3 TEMPORAL.DAT <cr>
```

Escribe en el archivo TEMPORAL.DAT 3 lineas a partir de la linea actual, permaneciendo las lineas en el buffer. Si no se especifica el archivo se crea un archivo llamado SAVE.TMP .

UNSAVE

Recupera todas las lineas guardadas en el archivo SAVE.TMP o el que se especifique y las copia despues de la linea actual. Ejemplos:

*UN <cr> ; inserta despues de la linea actual el contenido de SAVE.TMP

*UN TEMPORAL.DAT <cr> ; inserta despues de la linea actual el contenido del archivo TEMPORAL.DAT

READ

Lee n bloques al buffer, si ya existe un bloque en el buffer le agrega los n especificados. Ejemplo:

*READ 2 <cr> ; lee 2 bloques al buffer

PRINT

Escribe o despliega desde la linea actual hasta la linea n-1, la ultima linea desplegada se convierte en la linea actual. Ejemplo:

*P5 <cr>

UPPER CASE ON/OFF

Es posible con esta opcion escribir en el archivo editado con letras mayusculas o minusculas. Ejemplos:

*UC OFF <cr>

Los caracteres son aceptados tal y como son tecleados, minusculas o mayusculas, y se escriben en el archivo de salida.

*UC ON <cr>

Los caracteres son aceptados, ya sean mayusculas o minusculas, pero son escritos en mayusculas al archivo de salida.

VERIFY ON/OFF

Controla el despliegue de lineas para los comandos de LOCATE y CHANGE, si:

```
*V ON <cr>
```

es tecleado LOCATE y CHANGE despliegan la linea localizada o la linea cambiada respectivamente.

RESUMEN DE COMANDOS DE EDI

COMANDO	FORMATO	DESCRIPCION
ADD	A cadena	Agrega la cadena al final de la linea actual
ADD and PRINT	AP cadena	Igual que ADD, y ademas imprime
BLOCK ON/OFF	BL ON BL OFF	Switch de modos de acceso de textos, modo de bloque y modo de lineas
BOTTOM	BO	Mueve el apuntador al final del bloque actual
CLOSE	CL [archivo]	Transfiere el bloque actual y el archivo de entrada al archivo de salida, cerrando ambos archivos. El archivo de salida es renombrado con el nombre "archivo"
CLOSE	CD [archivo]	Igual que CLOSE, solo que el archivo de entrada es borrado.
CLOSE SECONDARY	CLOSES	Cierra un archivo secundario
CHANGE	[n]C/cadena1/cadena2	Remplaza la cadena1 con la cadena2 "n" veces en la linea actual
CONCATENATION CHARACTER	CC [letra]	Cambia la concatenacion de caracteres a un caracter especifico
CTRL/Z	Z	Cierra los archivos y termina la sesion de edicion. El bloque actual es salvado
DELETE	D [n] D [-n]	Borra la linea actual y las n-1 siguientes si "n" es positiva. Borra las "n" anteriores a la linea actual si "n" es negativa
DELETE and PRINT	DP [n] DP [-n]	Igual que DELETE solo que ahora imprimira la linea actual
EXIT	EX	Igual que CTRL/Z
EXIT DELETE	ED	Igual que EXIT, solo que ahora se borra el archivo de entrada, es decir, no crea otra version
FILE	FIL [archivo]	Transfiere lineas del archivo de entrada al de salida, y al archivo especifico
INSERT	I cadena	Inserta la cadena en la siguiente linea despues de la linea actual. Entra en modo de texto si la cadena es omitida
KILL	KILL	Cierra los archivos de entrada y salida. Y ademas borra el archivo de salida
LINE CHANGE	[n]LC/cadena1/cadena2	Cambia la cadena1 por la cadena2 a partir de la linea actual "n" lineas
LIST ON TERMINAL	LI	Lista todas las lineas restantes del bloque actual
LOCATE	L cadena	Igual a FIND, apartir de la linea actual

MACRO X	MACRO X	Define el macro x(1,2,3)
MACRO EXECUTE	[n]MX	Ejecuta el macro X n veces
NEXT	N [n] N [-n]	Establece una nueva linea actual "n" lineas despues de la linea actual
NEXT PRINT	NP [n] NP [-n]	Igual que NEXT, solo que ahora se imprime la linea actual
OPEN SECUNDARY	OP archivo	Abre un archivo secundario especifico
OUTPUT ON/OFF	OU ON	Continua o descontinua la transferencia al archivo de salida
OVERLAY	O [n]	Borra "n" lineas, y entra en modo editor
PAGE	PAGE [n] PAGE [-n]	Lee el n-esimo bloque del archivo de entrada, el cual sera ahora el nuevo bloque actual
PAGE FIND	[n]PF cadena	Busca en bloques sucesivos la n-esima ocurrencia de cadena
PAGE LOCATE	[n]PL cadena	Igual que PAGE FIND
PASTE	PA/cadena1/ cadena2	Remplaza cadena1 por cadena2 a partir de la linea actual en el bloque actual
PRINT	P [n]	Imprime "n" lineas y la ultima es ahora la linea actual, a partir de la linea actual
READ	REA [n]	Lee los siguientes "n" bloques dentro del bloque actual
RENEW	REN [n]	Escribe el bloque actual al archivo de salida y lee el siguiente bloque. Esta operacion se repite "n" ve#es
RETURN	<cr>	I-priee la sigu)e.te linea, haciendo a esta la linea actual
RETYPE	RE cadena	Remplaza la linea actual por la cadena
SAVE	SA[n]archivo	Salva la linea actual y las n-1 lineas siguientes en el archivo especificado
SELECT PRIMARY	SP	Restablece al archivo primario como archivo de entrada
SELECT SECONDARY	SS	Selecciona al archivo secundario abierto como archivo de entrada
SEARCH	SC/cadena1/ cadena2	Localiza la cadena1 y la sustituye por la cadena2
SIZE	SIZE [n]	Especifica el numero maximo de lineas para ser leidas dentro del archivo de bloques
TAB	TA ON/OFF	TA/ON habilita 8 espacios al principio de la linea de entrada, el default es TA/OFF
TOP	T	Coloca el apuntador al inicio del bloque actual
TOP of FILE	TOF	Coloca el apuntador al inicio del del archivo
TYPE		Igual a PRINT, pero el apuntador no es modificado
UNSAVE	UNS [archivo]	Inserta todas las lineas del archivo especificado despues de la linea actual
VERIFY ON/OFF	V ON/OFF	Controla la impresion de la linea actual en los comandos L y C
WRITE	W	Escribe el bloque actual al archivo de salida y borra el bloque actual

Para ejemplificar el uso del editor elaboraremos un programa en FORTRAN.

```
>EDI EJEMPLO.FTN
[CREATING NEW FILE]
INPUT
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
      DIMENSION I(5)
      TYPE 10 ; TYPE es una instruccion para escribir
      ACCEPT *,(I(J),J=1,5) ; ACCEPT es una instruccion para leer
      DATO=1
      ISUMA=0
      DO 15 J=1,5
      ISUMA=ISUMA+I(J)
15    CONTINUE
      IPROM=ISUMA/5
      TYPE 20,IPROM
10    FORMAT(1X,'DAME 5 DATOS ENTEROS, UNO POR RENGLON')
20    FORNAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
      CAII EXIT
      END

*ED
[EXIT]

>
```

Como se podra apreciar en el programa existen algunos errores, para corregirlos volveremos a editar el programa EJEMPLO.FTN
La informacion que viene despues de un ";" son comentarios.

```
>EDI EJEMPLO.FTN
[00016 LINES READ IN]
[PAGE    1]
* <cr> ; el asterisco nos indica que estamos en modo editor
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
* <cr> ; cr = return (para avanzar un renglon)
      DIMENSION I(5)
* <CR>
      TYPE 10 ; TYPE esuna instruccion para escribir
* <CR>
      ACCEPT *,(I(J),J=1,5) ; ACCEPT es una instruccion para leer
* <CR>
      DATO=1
*DP ; borramos esta linea que no tiene ninguna funcion
      ISUMA=0
*L CAII ; localizamos la palabra CAII
      CAII EXIT
*2C/II/LL ; cambiamos las dos I por dos L
      CALL EXIT
```

```
*T ; mandamos el apuntador al inicio del bloque actual
* <cr>
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
* <CR>
      DIMENSION I(5)
* <CR>
      TYPE 10 ; TYPE es una instruccion para escribir
* <cr>
      ACCEPT *,(I(J),J=1,5) ; ACCEPT es una instruccion para leer
* <esc> ; se oprime la tecla ESC para regresar una posicion el apuntador
      TYPE 10 ; TYPE es una instruccion para escribir
*P ; ahora damos este comando para corroborar que esta es la linea actual
      TYPE 10 ; TYPE es una instruccion para escribir
*NP A ; aqui probamos un comando incorrecto
[ILL CMD]
*NP 3 ; posicionamos al apuntador 3 lineas despues de la actual
      DO 15 J=1,5
*C/0/O ; cambiamos el 0(cero) por la letra O
      DO 15 J=1,15
*BO ; posicionamos el apuntador en la ultima linea del bloque actual
      END
* <cr>
[*EOB*]
* ; End Of Buffer nos indica que ya no hay mas lineas en este bloque
[*EOB*]
*REN ; este comando traera un nuevo bloque, que sera ahora el actual
[*EOF*]
[PAGE    2]
* ; estos avisos nos indican que ya no hay mas lineas en el archivo
  ; EOF significa End Of File(fin de archivo)
  ; en seguida damos un comando para tener el apuntador al inicio del archiv
*TOF ; Top Of File
[00015 LINES READ IN]
[PAGE    1]
* <cr>
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
*I <CR> ; entramos a modo input, para insertar nuevas lineas
C  ESTE EJEMPLO SOLO NOS MUESTRA ALGUNOS COMANDOS, <cr>
C  RECOMENDAMOS AL ALUMNO LA PRACTICA Y USO DE TODOS  <cr>
C  LOS COMANDOS QUE SE MUESTRAN EN EL RESUMEN.  <cr>

*T <cr>
*TY 4 <cr> ; desplegamos las 4 primeras lineas, sin mover el apuntador

C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
C  ESTE EJEMPLO SOLO NOS MUESTRA ALGUNOS COMANDOS,
C  RECOMENDAMOS AL ALUMNO LA PARCTICA Y USO DE TODOS
[*BOB*]  ; nos indica el inicio de bloque
* <cr>
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
*T <cr>
*LI <cr>;lista las lineas del bloque actual, dejando el apuntador al inicio
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
```

```
C  ESTE EJEMPLO SOLO NOS MUESTRA ALGUNOS COMANDOS,
C  RECOMENDAMOS AL ALUMNO LA PRACTICA Y USO DE TODOS
C  LOS COMANDOS QUE SE MUESTRAN EN EL RESUMEN.
      DIMENSION I(5)
      TYPE 10 ; TYPE es una instruccion para escribir
      ACCEPT *,(I(J),J=1,5) ; ACCEPT es una instruccion para leer
      ISUMA=0
      DO 15 J=1,5
      ISUMA=ISUMA+I(J)
15    CONTINUE
      IPROM=ISUMA/5
      TYPE 20,IPROM
10    FORMAT(1X,'DAME 5 DATOS ENTEROS, UNO POR RENGLON')
20    FORNAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
      CALL EXIT
      END
*ED ; comando para salvar el archivo y regresar el control a MCR
[EXIT]
```

* MANEJO DEL PROGRAMA DE INTERCAMBIO PERIFERICO PIP *

P I P (Peripherical Interchange Program)

(Programa de intercambio periferico)

PIP es un programa de utileria que permite realizar funciones como:

- Copiar archivos de un dispositivo a otro.
- Borrar archivos.
- Purgar archivos.
- Renombrar archivos.
- Listar los contenidos de los directorios.
- Etc.

Existen dos maneras de invocar a PIP:

1) En un solo comando:

```
EJ:
>PIP linea de comando,linea de comando,etc/switch<cr>
```

Se ejecuta el comando y el control de la terminal regresa al MCR.

2) Comandos multiples:

```
EJ: >PIP <cr>
    PIP>linea de comando/switch  <cr>
    PIP>linea de comando1/switch <cr>
     ----
    PIP>CTRL/Z
```

Para especificar completamente un archivo se deben indicar las siguientes partes:

```
dev:[uic]nombre.extension;version
```

Ejemplo:

```
DK1:[23,1]TAREA.PAS;3
```

User File Directory -- UFD
User Identification Code -- UIC

Un disco esta organizado en la PDP11 bajo directorio y la informacion (nombre de archivo) contenida en cada directorio, esta registrada en un archivo llamado UFD, cuyo nombre esta formado por el UIC seguido de .DIR y version 1.

Ejemplo:

UIC=[30,2]

el nombre del archivo UFD seria - 030002.DIR;1

Estos UFD's se encuentran en la cuenta [000,000] del sistema.

Nombres comunes de dispositivos;

Cinta de papel	DTn:
Cinta magnetica	MMn:
Disco	DBn: DKn: DMn: Dpn: DXn:
Dispositivo del sitema	SY:
Impresora	LPn:
Lectora de Tarjetas	CR:
Pseudo terminal	TI:
Terminal	TTn:

Cuando se omite alguna parte de la especificacion del archivo, PIP asume lo siguiente:

* dev: - la unidad en donde el sistema es montado o la unidad especificada por el switch /DF (default) o la ultima especificacion hecha. (para nuestro caso el default es DM0:)

* [uic] - la cuenta donde el usuario inicio su sesion o la cuenta de la ultima especificacion hecha.

* nombre - no hay default para la primera especificacion los subsecuentes toman la especificacion anterior (puede llegar a aparecer un asterisco)

* extension - igual que para 'nombre'

* version - por default se tiene la version mas reciente en caso de ser una copia del archivo, el numero siguiente de la ultima version, si se desea borrar, si se requiere de una version explicita(puede aceptar un asterisco)

CONVENCIONES DEL USO DEL ASTERISCO

CASO(ejemplo)	SIGNIFICADO
.;*	todas las versiones de todos los archivos de cualquier tipo.
.TXT;	todas las versiones de todos los archivos de tipo TXT.
FIBO.*;*	todas las versiones y todos los tipos de archivo, cuyo nombre sea FIBO.
FIBO.PAS;*	todas las versiones del archivo FIBO.PAS .
.	las mas recientes versiones de todos los archivos.
*.FTN	las mas recientes versiones de los archivos cuyo tipo es FTN.
FACT.*	las mas recientes versiones de todos los tipos de archivos cuyo nombre es FACT.
FACT.MAC	las mas recientes versiones de FACT.MAC

COMANDOS MAS UTILIZADOS DE PIP

Para copiar un archivo basta con invocar a PIP y especificar los archivos de entrada y de salida respectivamente.

En general:

```
>PIP archivo de salida = archivo de entrada
```

Ejemplo:

```
>PIP DK1:PRUEBA2.FTN=DK0:PRUEBA1.FTN
```

Copia la ultima version del archivo PRUEBA1.FTN del disco DK0 al disco DK1 con el nombre PRUEBA2.FTN

```
>PIP DK1:[*,*]=[11,*]
```

Copia todos los archivos de todos los miembros del grupo 11 del dispositivo de default a DK1 preservando el uic

MERGE SWITCH /ME

Para efectuar la concatenacion de dos o mas archivos. Ejemplo:

```
>PIP TRES.DAT=UNO.DAT,DOS.DAT;3/ME
```

Crea un archivo llamado TRES.DAT que contiene la concatenacion de UNO.DAT y DOS.DAT

APPEND SWITCH /AP

Para agregar archivos a un archivo ya existente. Ejemplo:

```
>PIP DK1:DIA1.FTN;1=DIA2.FTN;1,DIA3.FTN;1/AP
```

Al archivo existente DIA1.FTN en el disco DK1 le agrega los archivos DIA2.FTN;1 y DIA3.FTN;1

DELETE SWITCH /DE

Para borrar archivos de disco. Ejemplos:

```
PIP>DATOS.FTN;n/DE
```

Aqui borra la version n-esima del archivo DATOS.FTN

```
PIP>DATOS.FTN;-1/DE
```

Borra la ultima version del archivo DATOS.FTN

```
>PIP *.OBJ;*/DE
```

Borra todas las versiones de todos los archivos de un tipo determinado(OBJ)

```
>PIP FIBO.*;*/DE
```

Borra todas las versiones de todos los tipos del archivo FIBO

```
>PIP *.*;*/DE
```

Borra todas las versiones de todos los archivos de cualquier tipo de la cuenta actual

DEFAULT SWITCH /DF

Cambia el dispositivo o el UFD de default. Ejemplos:

```
>PIP [100,100]/DF
```

Cambia el UFD de default a la cuenta [100,100]

```
>PIP DK1:/DF
```

Cambia el dispositivo de default al dispositivo DK1

PROTECTION SWITCH /PR

SUBSWITCHES /SY /OW /GR /WO

Proteccion de archivos y privilegios

Cada archivo tiene asociada una palabra de 16 bits cuyo formato es el siguiente:

```
15        12  11        8  7       4  3        0
world
              group
                           owner
                                      system
```

Es a traves de esta palabra que se asignan privilegios y protecciones sobre un archivo.
Los bits al estar prendidos significan que no hay acceso permitido, en cada campo hay 4 bits que significan:

```
     D          E        W        R
   ----------------------------------
    delete
              extend
                       write
                               read
```

donde:

System : grupos y miembros <= 10 (base 8)

Owner : el de la cuenta

Group : los del mismo grupo

World : todos los demas

Ejemplos:

```
>PIP PRUEBA.FTN;3/PR/OW:RWE/GR:RWE/WO:
```

Se asignan privilegios al owner y al grupo para realizar escritura, lectura y extensiones (no pueden borrar), el world no tiene privilegios y los del sistema permanecen sin cambios.

```
>PIP PRUEBA.FTN;3/PR:3
```

Se asignan los privilegios quitados al archivo PRUEBA.FTN;3

PURGE SWITCH /PU

Borra un rango especifico de archivos, cuyas versiones ya resultan obsoletas. Ejemplos:

```
>PIP *.FTN/PU
```

Borra todas las versiones, excepto la ultima de todos los archivos cuyo tipo es FTN

```
>PIP *.*/PU
```

Nos deja solo las ultimas versiones de todos los archivos con cualquier tipo.

RENAME SWITCH /RE

Cambia el nombre de un archivo. Ejemplo:

```
>PIP DESPUES.PAS=ANTES.PAS/RE
```

El archivo ANTES.PAS es renombrado como DESPUES.PAS

LIST SWITCH /LI

Lista el contenido del directorio del dispositivo y cuenta de default. Ejemplo:

```
>PIP /LI
```

Si ademas de la informacion proporcionada por la opcion /LI se desea conocer la proteccion de cada archivo, la fecha y la hora de la ultima actualizacion, asi como el numero de revisiones, se utiliza el switch /FU (full). Ejemplo:

```
>PIP /FU
```

La opcion /BR(brief), es una forma breve (y mas rapida) de "LI". Ejemplo:

```
>PIP /BR
```

NOTA: Existen otros switches, para su consulta ver el manual:

El programa PIP tambien permite listar el contenido de un archivo, esto se logra mediante los siguientes comandos:

```
>PIP TI:= especificacion del archivo
```

Despliega el contenido del archivo especificado por el usuario en la pantalla de su terminal.

Nota: Como el listado se efectua en forma muy rapida, es necesario detener la transmision, esto se logra oprimiendo simultaneamente las teclas de CTRL y S; para restaurar la transmision se oprime CTRL y Q.

```
>PIP TTn:= especificacion del archivo
```

Despliega el contenido del archivo especificado por la terminal "n".

Ver ejemplos anexos.

RESUMEN DE LOS SWITCHES IMPORTANTES DE PIP

SWITCH	FUNCION
/AP	Agrega archivos al final de un archivo ya existente.
/BR	Lista el directorio en forma breve.
/CO	Especifica que el archivo de salida debe ser contiguo.
/DE	Borra uno o mas archivos.
/DF	Cambia la cuenta o dispositivo de dafault.
/EN	Entra un sinonimo para un archivo del directorio.
/FI	Accesa un archivo por su numero de identificacion.
/FR	Despliega el total de espacio libre sobre un volumen especifico.
/FU	Lista el directorio en un formato mas completo.
/ID	Identifica la version que ha sido usada.
/LI	Lista el directorio.
/PR	Cambia las protecciones de un archivo.
/PU	Borra todas las versiones, dejando solo la ultima.
/RE	Cambia el nombre de un archivo.
/TB	Proporciona el tamano del directorio, dando el total de lineas en el.
/UN	Abre un archivo.

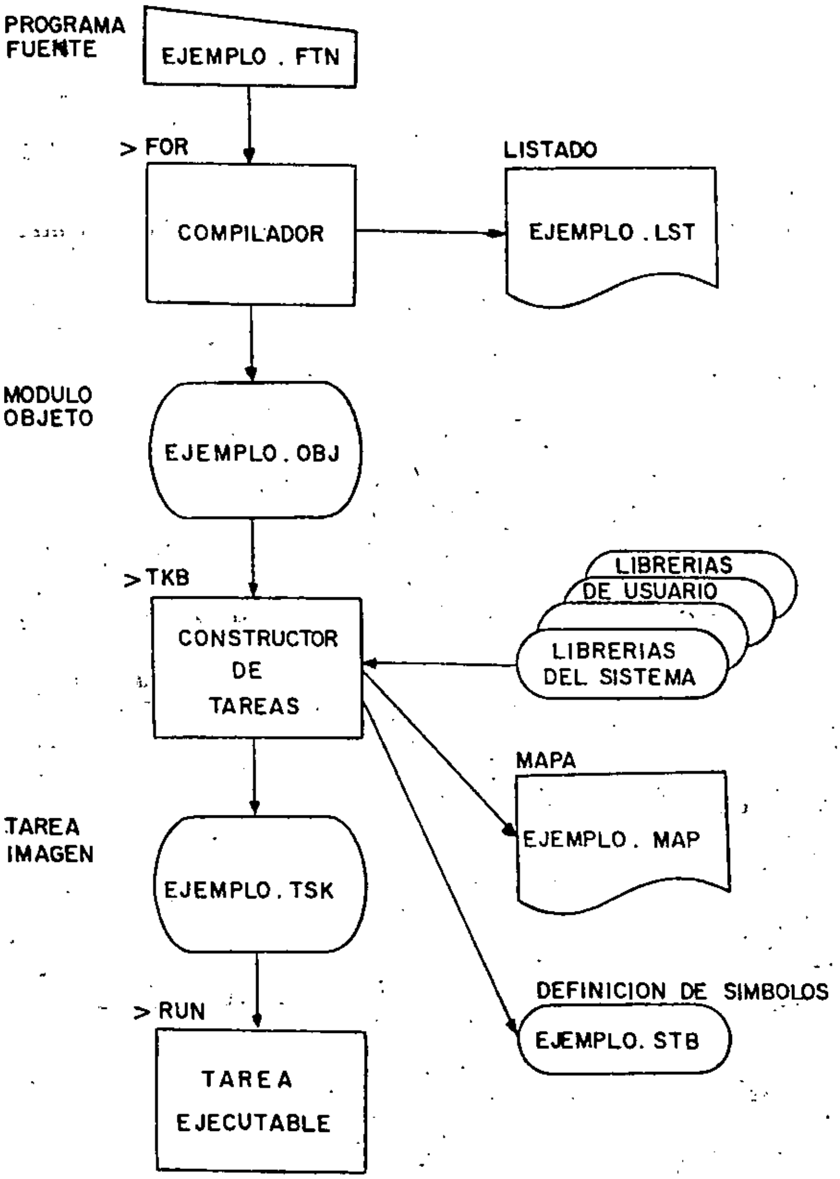
EJemplos usando el programa PIP

```
>PIP /LI   ; nos muestra el directorio

DIRECTORY DM0:[206,11]
9-OCT-84 09:46

EJEMPLO.FTN;3          2.          14-SEP-84 13:21
EJEMPLO.OBJ;3          3.          14-SEP-84 14:57
EJEMPLO.OBJ;4          3.          09-OCT-84 09:43
DATOS.TXT;1            0.          14-SEP-84 13:22

TOTAL OF 8./10. BLOCKS IN 4. FILES


>PIP /FU   ; nos muestra el directorio en una forma mas completa

DIRECTORY DM0:[206,11]
9-OCT-84 09:47

EJEMPLO.FTN;3          (2474,71)          2./2.               14-SEP-84 13:21
  [6,1]     [RWED,RWED,RWED,R]    04-OCT-84 09:12(3.)
EJEMPLO.OBJ;3          (3434,56)          3./3.               14-SEP-84 14:57
  [206,11] [RWED,RWED,RWED,R]    04-OCT-84 09:12(4.)
EJEMPLO.OBJ;4          (3463,63)          3./5.               09-OCT-84 09:43
  [206,11] [RWED,RWED,RWED,R]    09-OCT-84 09:43(2.)
DATOS.TXT;1            (3263,74)          0./0.               14-SEP-84 13:22
  [206,11] [RWED,RWED,RWED,R]    04-OCT-84 09:12(3.)

TOTAL OF 8./10. BLOCKS IN 4. FILES


>PIP /BR   ; nos muestra el directorio en forma breve


DIRECTORY DM0:[206,11]

EJEMPLO.FTN;3
EJEMPLO.OBJ;3
EJEMPLO.OBJ;4
DATOS.TXT;1


>PIP /FR   ; nos indica la cantidad de memoria libre

DM0: HAS 3524. BLOCKS FREE, 50266. BLOCKS USED OUT OF 53790.

>
```

```
>PIP *.OBJ;*/LI ; lista todas las versiones de todos los archivos del tipo
                                                                        OBJ
 IRECTORY DMO:[206,11]
 -OCT-84 09:51

EJEMPLO.OBJ;3         3.          14-SEP-84 14:57
EJEMPLO.OBJ;4         3.          09-OCT-84 09:43

TOTAL OF 6./8. BLOCKS IN 2. FILES


>PIP EJEMPLO.*;*/LI ; lista todos los tipos y versiones del archivo EJEMPLO

DIRECTORY DMO:[206,11]
9-OCT-84 09:52

EJEMPLO.FTN;3         2.          14-SEP-84 13:21
EJEMPLO.OBJ;3         3.          14-SEP-84 14:57
EJEMPLO.OBJ;4         3.          09-OCT-84 09:43

TOTAL OF 8./10. BLOCKS IN 3. FILES


>PIP *.*/PU  ; deja las ultimas versiones de todos los archivos
>PIP /LI  ; checamos el comando anterior

DIRECTORY DMO:[206,11]
9-OCT-84 09:54

EJEMPLO.FTN;3         2.          14-SEP-84 13:21
EJEMPLO.OBJ;4         3.          09-OCT-84 09:43
DATOS.TXT;1           0.          14-SEP-84 13:22

TOTAL OF 5./7. BLOCKS IN 3. FILES


>PIP TI:=EJEMPLO.FTN  ; listamos por la terminal el archivo EJEMPLO.FTN
C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
C ESTE EJEMPLO SOLO NOS NUESTRA ALGUNOS COMANDOS,
C RECOMENDAMOS AL ALUMNO LA PRACTICA Y USO DE TODOS
C LOS COMANDOS QUE SE MUESTRAN EN EL SUMARIO.
      DIMENSION I(5)
      TYPE 10
      ACCEPT *,(I(J),J=1,5)
      ISUMA=0
      DO 15 J=1,5
      ISUMA=ISUMA+I(J)
15    CONTINUE
      IPROM=ISUMA/5
      TYPE 20,IPROM
10    FORMAT(1X,'DAME 5 DATOS ENTEROS, UNO POR RENGLON')
20    FORNAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
      CALL EXIT
      END
>
```

* SECUENCIA A SEGUIR PARA LA EJECUCION DE UN PROGRAMA *

Diagrama de flujo para la ejecucion de un programa.



PASOS PARA LA EJECUCION DE UN PROGRAMA

1) Edicion del archivo fuente.

Este paso se logra haciendo un uso eficiente del editor y sujetandose a las reglas que marque cada lenguaje, para la estructuracion del programa.

Ejemplos:

a) >EDI EJ1.FTN ; archivo para el compilador fortran

b) >EDI EJ2.PAS ; archivo para el compilador pascal

c) >EDI EJ3.MAC ; archivo para el ensamblador macro-11

d) >EDI EJ4.C ; archivo para el compilador c

e) >EDI EJ5.TXT ; archivo que contiene solo texto, no es procesable

f) >EDI EJ6.DAT ; archivo de datos

g) Etc.

2) Compilacion del archivo fuente.

En esta fase se lleva a cabo la traduccion de programas escritos en lenguaje fuente (lenguaje de alto nivel o lenguaje ensamblador) a sus equivalentes en lenguaje maquina (sistema octal para nuestro caso PDP11/40).

Daremos algunos ejemplos usando el compilador fortran, sin embargo, el resultado es analogo para los otros compiladores.

a) >FOR SALIDA.OBJ=ENTRADA.FTN

Dado el archivo fuente ENTRADA.FTN el compilador producira un archivo de salida llamado SALIDA.OBJ conteniendo el codigo objeto generado por el compilador. Con FOR estamos llamando al compilador fortran.

b) >FOR SALIDA=ENTRADA

Tiene el mismo efecto que el ejemplo anterior, ya que si no se escriben los tipos de los archivos el compilador FOR asume por default que el archivo a la derecha del signo igual tiene una extension FTN y el archivo de salida (a la izquierda del signo igual) se le asignara la extension OBJ.

```
c) >FOR SALIDA2.OBJ,SALIDA1.LST=ENTRADA.FTN
```

Comando similar al ejemplo a), solo que ahora se creara un archivo mas, SALIDA1.LST, el cual contendra informacion adicional, usada para un analisis del mismo.

```
d) >FOR SALIDA2,SALIDA1=ENTRADA
```

Comando identico al anterior.

```
e) >FOR ,TI:=ENTRADA
```

Este comando produce el archivo del listado (.LST) por la pantalla del usuario, y ademas no sera creado un archivo con codigo objeto. Este caso es muy usado cuando se tiene un programa con errores, ya que por medio de este comando los errores apareceran en la linea donde ocurran, asi como el tipo de error. Si se usara por ejemplo el comando del inciso b), al final de la compilacion solo se mostraria el numero de linea donde ocurrio el error y una descripcion del mismo.

Existen otras opciones para la compilacion, para mayor informacion consultar el manual:

FORTRAN IV USER's GUIDE

3) Construccion de una tarea ejecutable.

En esta fase el Task Builder (TKB) es el encargado de producir una tarea ejecutable.

Los siguientes tipos de archivos son los aceptados y producidos por el TKB.

```
      ARCHIVO DE             EXTENSION
    --------------          -----------
      mapa                     MAP <----
      biblioteca               OLB      |
      archivo.obj              OBJ      >-- estos dos archivos son
      tarea imagen             TSK <----|   producidos por el TKB
      tabla de simbolos        STB
```

Ejemplos:

```
a) >TKB SALIDA4.TSK,SALIDA3.MAP=SALIDA2.OBJ
```

En este comando se da como archivo de entrada el programa ya compilado, es decir, el codigo objeto. La salida consiste de dos archivos: SALIDA4.TSK en donde se encuentra la tarea imagen que va a ser ejecutada y SALIDA3.MAP que es el mapa de la tarea imagen. Con TKB estamos llamando al Task Builder.

```
b) >TKB SALIDA4,SALIDA3=SALIDA2
```

Comando equivalente al anterior.

```
c) >TKB SALIDA=ENTRADA1,ENTRADA2,etc
```

Con este comando se crea una tarea ejecutable llamada SALIDA.TSK, que esta compuesta de los modulos objetos de los archivos ENTRADA1, ENTRADA2, etc.

Tambien es valido el siguiente comando:

```
d) >TKB SALIDA,TI:=ENTRADA
```

Con este comando creamos una tarea ejecutable, y ademas se despliega informacion sobre estadisticas del programa.

4) Ejecucion de la tarea.

En esta fase estamos listos para ejecutar el programa.

Ejemplos:

```
a) >RUN SALIDA4

b) >RUN SALIDA

c) etc.
```

Es muy usual que todos los archivos tengan el mismo nombre para facilitar el manejo de los mismos(en el purgado, borrado, etc). Por ejemplo la siguiente secuencia es valida:

```
>EDI CORRE.FTN

     -----
      ---
       -

>FOR CORRE=CORRE

>TKB CORRE=CORRE

>RUN CORRE
```

NOTA: Los comados EDI, FOR, TKB, RUN pueden ser utilizados en dos formas: modo inmediato y modo prompt (en forma analoga como se explico el SOS, solo que para regresar el control a MCR se deben oprimir simultaneamente las teclas CTRL/Z).

" EJECUCION DE UN PROGRAMA "

Ejemplo de la ejecucion de un programa

```
>EDI EJEMPLO.FTN
[00018 LINES READ IN]
[PAGE     1]
*LI

C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
C ESTE EJEMPLO SOLO NOS NUESTRA ALGUNOS COMANDOS,
C RECOMENDAMOS AL ALUMNO LA PRACTICA Y USO DE TODOS
C LOS COMANDOS QUE SE MUESTRAN EN EL SUMARIO.
      DIMENSION I(5)
      TYPE 10
      ACCEPT *,(I(J),J=1,5)
      ISUMA=0
      DO 15 J=1,5
      ISUMA=ISUMA+I(J)
15    CONTINUE
      IPROM=ISUMA/5
      TYPE 20,IPROM
10    FORMAT(1X,'DAME 5 DATOS ENTEROS, UNO POR RENGLON')
20    FORNAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
      CALL EXIT
      END
*ED
[EXIT]

>FOR EJEMPLO=EJEMPLO
.MAIN.




FORTRAN IV DIAGNOSTICS FOR PROGRAM UNIT .MAIN.

IN LINE 0009,  ERROR:   INVALID FORMAT SPECIFIER
IN LINE 0011,  ERROR:   [SEE SOURCE LISTING]

FOR -- [.MAIN.] ERRORS: 2, WARNINGS: 0
>
```

Para saber donde esta especificamente el error vamos a compilar de nuevo con otra opcion

```
>FOR ,TI:=EJEMPLO/LI:1




FORTRAN IV      V02.04          TUE 09-OCT-84 10:01:13             PAGE 001
CORE=14K, UIC=[206,11]                                     ,TI:=EJEMPLO/LI:1

      C  PROGRAMA PARA CALCULAR EL PROMEDIO DE 5 DATOS
      C ESTE EJEMPLO SOLO NOS NUESTRA ALGUNOS COMANDOS,
      C RECOMENDAMOS AL ALUMNO LA PRACTICA Y USO DE TODOS
      C LOS COMANDOS QUE SE MUESTRAN EN EL SUMARIO.
0001            DIMENSION I(5)
0002            TYPE 10
0003            ACCEPT *,(I(J),J=1,5)
0004            ISUMA=0
0005            DO 15 J=1,5
0006            ISUMA=ISUMA+I(J)
0007  15        CONTINUE
0008            IPROM=ISUMA/5
0009            TYPE 20,IPROM
0010  10        FORMAT(1X,'DAME 5 DATOS ENTEROS, UNO POR RENGLON')
0011  20        FORNAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
***** U
0012            CALL EXIT
0013            END

.MAIN.




FORTRAN IV DIAGNOSTICS FOR PROGRAM UNIT .MAIN.

IN LINE 0009,  ERROR:   INVALID FORMAT SPECIFIER
IN LINE 0011,  ERROR:   [SEE SOURCE LISTING]

FOR -- [.MAIN.] ERRORS: 2, WARNINGS: 0
>
```

Para corregir el error debemos entrar nuevamente al editor

```
>EDI EJEMPLO.FTN
[00018 LINES READ IN]
[PAGE     1]
*SC/FORNAT/FORMAT
20     FORMAT(1X,'ESTE ES EL PROMEDIO',2X,I5)
*ED
[EXIT]

>FOR EJEMPLO=EJEMPLO
.MAIN.
>
```

```
; Como ya no tenemos errores podemos pasar al ligado
```

```
>TKB EJEMPLO=EJEMPLO
>
```

```
; La fase del ligado no tuvo errores por tanto, pasamos a la ejecucion
```

```
>RUN EJEMPLO
DAME 5 DATOS ENTEROS, UNO POR RENGLON
1
2
3
4
5
ESTE ES EL PROMEDIO      3
>
```

BIBLIOGRAFIA

- RSX-11M
  Beginner's Guide

- RSX-11M
  FORTRAN IV User's Guide

- IAS/RSX -11
  Utilities Procedures Manual
    - Line Text Editor (EDI)
    - Peripheral Interchange Program (PIP)

- RSX-11M
  Operator's Procedures Manual
    - MCR Commands

- RSX-11M
  Task Builder Reference Manual

- Processor handbook
  pdp11/05/10/35/40

- Hazeltine 1421
  Video display terminal reference manual

- UDC11
  Universal digital control subsystem maintenance manual

- DECGRAPHIC-11
  FORTRAN Programming Manual

- RX8/RX11
  floppy disk system user's manual

- RK05
  disk drive maintenance manual

- RK06/RK07 Disk Drive
  User's Manual

- LA36/LA35 DECwriter II
  User's Manual

PRACTICA # 2

USO DEL LENGUAJE FORTRAN PARA EL PASO DE PARAMETROS A SUBRUTINAS EN MACRO-11 Y PROGRAMACION EN ENSAMBLADOR MACRO-11.

PRACTICA # 2

USO DEL LENGUAJE FORTRAN PARA EL PASO DE PARAMETROS A SUBRUTINAS EN MACRO-11 Y PROGRAMACION EN ENSAMBLADOR MACRO-11.

OBJETIVO:

El alumno aprendera la tecnica empleada para realizar programas en MACRO-11 utilizando al lenguaje FORTRAN para ejecutar las operaciones de lectura y escritura. Ademas reafirmara los conocimientos del ensamblador MACRO-11 obtenidos en la clase de teoria.

DESARROLLO:

Para lograr nuestro objetivo realizaremos diferentes programas.

Programa # 1 - Suma de dos numeros enteros.
Programa # 2 - Suma de dos numeros enteros. (PASCAL)
Programa # 3 - Suma de dos vectores.
Programa # 4 - Simulacion de una multiplicacion por sumas sucesivas.
Programa # 5 - Ordenamiento de letras y numeros.
Programa # 6 - Busqueda de un dato dentro de un vector.

LIGADO DE PROGRAMAS ESCRITOS EN FORTRAN CON PROGRAMAS EN MACRO-11, PASANDO PARAMETROS.

Como el lenguaje ensamblador no tiene instrucciones explicitas de lectura o escritura de datos del o al exterior, es necesario usar un programa en FORTRAN que realize estas operaciones.

Nuestro programa en ensamblador va a ser manejado como una subrutina en el programa FORTRAN y los parametros en la llamada seran los de entrada y salida al programa ensamblador.

Si el compilador FORTRAN encuentra una proposicion como la siguiente:

```
  CALL RUTINA(A1,A2,A3,...,AN)
```

crea un bloque con las siguientes caracteristicas:

```
                   ----------------------
      R5 -----> | No. de argumentos |
                   ----------------------
                   |  direccion de A1  |
                   ----------------------
                   |  direccion de A2  |
                   ----------------------
                   |         :         |
                   ----------------------
                   |  direccion de AN  |
                   ----------------------
```

colocando un apuntador a dicho bloque en el registro R5.
Si una subrutina en MACRO-11 desea obtener el numero de argumentos, lo puede hacer mediante la instruccion:

```
    MOV (R5)+,R0
```

y ahora R0 contendra dicho numero. Y R5 estara apuntando a la direccion del primer argumento.

```
                   ----------------------
                   | No. de argumentos |
                   ----------------------
      R5 -----> |  direccion de A1  |
                   ----------------------
                   |  direccion de A2  |
                   ----------------------
                   |         :         |
                   ----------------------
                   |  direccion de AN  |
                   ----------------------
```

Para obtener el regreso al programa escrito en FORTRAN se utilizan sub- la subrutina de MACRO-11 las instrucciones:

```
        RTS PC
        .END
```

PROGRAMA # 1

```
>EDI SUMAF.FTN

C  Programa en FORTRAN para realizar las operaciones de lectura y escritura
C  del programa SUMAM.MAC
C  Suma de dos numeros enteros.
C
C  NOTA:
C  En este programa, como en los demas se hara uso de las instrucciones
C  ACCEPT * y TYPE * para las operaciones de lectura y escritura con
C  formato libre, respectivamente.
C
5      TYPE *,'DAME LOS DATOS A Y B, CON FORMATO ENTERO'
       ACCEPT *,IA,IB
       CALL SUM(IA,IB,IC)   ! llamada a la subrutina en MACRO-11
       TYPE 10,IA,IB,IC
10     FORMAT(1X,I5,' + ',I5,' = ',I6)
       TYPE *,'DESEAS REALIZAR OTRA SUMA [Y/N] ?'
       ACCEPT 15,IR
15     FORMAT(A1)
       IF (IR.EQ.'Y') GOTO 5
       CALL EXIT
       END
```

```
>EDI SUMAM.MAC

; Programa en MACRO-11 para realizar la suma de dos numeros enteros
; Este programa sera ligado a SUMAF.FTN
;
SUM::                   ; nombre de la subrutina en MACRO-11
       MOV @2(R5),R0    ; R0 <-- IA
       MOV @4(R5),R1    ; R1 <-- IB
       ADD R0,R1        ; R1 <-- R1+R0
       MOV R1,@6(R5)    ; IC <-- R1
       RTS PC           ; regresa al programa en FORTRAN
       .END
```

```
>FOR SUMAF=SUMAF

>MAC SUMAM=SUMAM

>TKB SUMAF=SUMAF,SUMAM

>RUN SUMAF
```

PROGRAMA # 2

```
>EDI SUMAP.PAS

 * Programa en lenguaje PASCAL que realiza las operaciones de lectura y
   escritura, para obtener la suma de dos numeros enteros, realizando
   esta en ensamblador MACRO-11 *)

{ Para crear lineas en lenguaje ensamblador dentro de un programa en PASCAL }
{ se hace uso del concepto para un comentario }  (* este es un comentario *)
(* Para insertar lineas en ensamblador deberemos colocar la opcion $C despues
   de los caracteres " (* " que indican inicio de un comentario.
   El compilador examina el macro fuente para encontrar las referencias a va-
   riables en el programa en PASCAL. Para accesar una variable a nivel global
   llamada VAR1, se usa VAR1(%5), y para accesar una variable local o un argu-
   mento de un procedure llamada VAR2, usamos VAR2(%6)   *)

PROGRAM SUMA(INPUT,OUTPUT);
VAR
IA,IB,IC:INTEGER;
RES:CHAR;
BEGIN
  RES:='Y';
  WHILE RES<>'N' DO
    BEGIN
      WRITELN('DAME LOS DATOS A Y B, CON FORMATO ENTERO');
      READLN(IA,IB);
      (*$C
            MOV IA(%5),R0   ; R0 <-- IA
            ADD IB(%5),R0   ; R0 <-- IA+IB
            MOV R0,IC(%5)   ; IC <-- R0
      *)
      WRITELN(IA:5,' + ',IB:5,' = ',IC:6);
      WRITELN('DESEAS REALIZAR OTRA SUMA [Y/N] ?');
      READLN(RES)
    END
END.



>PAS SUMAP=SUMAP

>MAC SUMAP=SUMAP

>TKB SUMAP=SUMAP,[1,1]PASLIB/LB

>RUN SUMAP
```

PROGRAMA # 3

```
>EDI VECTF.FTN

C  Programa en FORTRAN para realizar la operacion de escritura del
   programa VECTM.MAC
C  En este programa se compartira un bloque comun de datos entre
C  el programa de FORTRAN y la subrutina de MACRO-11, se usara para
C  ello la declaracion COMMON de FORTRAN y la directiva .PSECT de
C  MACRO-11.
C
      COMMON/AREA/IA(10),IB(10),IC(10)
      DO 5 I=1,10
      IA(I)=I
      IB(I)=10
      IC(I)=0
5     CONTINUE
      CALL SUMA     ! llamada a la subrutina de MACRO-11
C                   ! vease que la llamada no contiene argumentos
C                   ! ya que se hara uso de variables globales
      TYPE 10,IC
10    FORMAT(10(2X,I4))
      CALL EXIT
      END
```

```
>EDI VECTM.MAC

; Programa en MACRO-11 para realizar la suma de dos vectores
; IC(I)=IA(I)+IB(I), haciendo uso de variables globales.
; Este programa sera ligado a VECTF.FTN
;
      .PSECT AREA,RW,D,GBL,REL,OVR
;
; AREA - nombre para identificar el bloque comun
; RW   - se tiene acceso para leer/escribir
; D    - indica la clase de informacion a manejar ( D=datos)
; GBL  - como el bloque contiene datos, se define al bloque como global
; REL  - se establece que el bloque es relocalizable
; OVR  - define los requirimientos de memoria asignada al bloque.
;         las secciones de datos son "overlaiadas".
;
IA:   .BLKW 12      ; se reservan 10 palabras para el vector IA
IB:   .BLKW 12      ; se reservan 10 palabras para el vector IB
IC:   .BLKW 12      ; se reservan 10 palabras para el vector IC
SUMA::              ; nombre de la subrutina en MACRO-11
      MOV #12,R0    ; R0 <-- 10. se inicializa un contador
      MOV #IA,R4    ; R4 contiene la direccion de IA(1)
      MOV #IB,R3    ; R2 contiene la direccion de IB(1)
      MOV #IC,R2    ; R2 contiene la direccion de IC(1)
ETI1: MOV (R3)+,R1  ; R1 <-- (R3)
      ADD (R4)+,R1  ; R1 <-- R1+(R4)
      MOV R1,(R2)+  ; (R2) <-- R1 , se almacenan las sumas
      SOB R0,ETI1   ; R0 <-- R0-1 , si R0<>0 ve a ETI1
      RTS PC        ; en caso contrario regresa al programa en FORTRAN
      .END
```

```
>FOR VECTF=VECTF

>MAC VECTM=VECTM

>TKB VECTF=VECTF,VECTM

>RUN VECTF
```

70

PROGRAMA # 4

```
>EDI MULTF.FTN

C  Programa en FORTRAN para realizar las operaciones de lectura y escritura
C  del programa MULTM.MAC
C  Simulacion de una multiplicacion por sumas sucesivas.
C
      INTEGER A,B,C
1     TYPE *,'DAME LOS FACTORES A Y B, CON FORMATO ENTERO'
      ACCEPT *,A,B
      CALL MULT(A,B,C)   ! llamada a la subrutina en MACRO-11
      TYPE 10,A,B,C
10    FORMAT(1X,I4,' X ',I4,' = ',I6)
      TYPE *,'DESEAS REALIZAR OTRA MULTIPLICACION [Y/N] ?'
      ACCEPT 15,IR
15    FORMAT(A1)
      IF (IR.EQ.'Y') GOTO 1
      CALL EXIT
      END

>EDI MULTM.MAC

; Programa en MACRO-11 que simula una multiplicacion de 2 factores,
; realizando el menor numero de sumas posible, los factores se en--
; cuentran en los registros R0 y R1 y el resultado en R3.
; Este programa sera ligado a MULTF.FTN
;
MULT::                    ; nombre de la subrutina en MACRO-11
       MOV @2(R5),R0      ; R0 <-- A
       MOV @4(R5),R1      ; R1 <-- B
       MOV #0,R3          ; R3 <-- 0
       TST R0             ; si R0=0
       BEQ ETI3           ; ve a ETI3
       TST R1             ; si R1=0
       BEQ ETI3           ; ve a ETI3
       CMP R0,R1          ; si R0<R1
       BMI ETI2           ; ve a ETI2
       MOV R1,R2          ; R2 contiene el # de sumas a ejecutar, R0>R1
       MOV R0,R4          ; R4 contiene el # a sumar R2 veces
ETI1:  ADD R4,R3          ; R3 <-- R3+R4 se efectuan las sumas sucesivas
       SOB R2,ETI1        ; R2 <-- R2-1 , si R2<>0 ve a ETI1
       BR ETI3            ; en caso contrario ve a ETI3
ETI2:  MOV R0,R2          ; R2 contiene el # de sumas a ejecutar, R0<R1
       MOV R1,R4          ; R4 contiene el # a sumar R2 veces
       BR ETI1            ; ve a ETI1
ETI3:  MOV R3,@6(R5)      ; C <-- R3 , contiene el resultado
       RTS PC             ; regresa al programa en FORTRAN
       .END

>FOR MULTF=MULTF
>MAC MULTM=MULTM

>TKB MULTF=MULTF,MULTM

>RUN MULTF
```

PROGRAMA # 5

```
>EDI ORDENAF.FTN

C     Programa en FORTRAN para realizar las operaciones de lectura y escritura
C     del programa ORDENAM.MAC
C     Este programa ordena letras o numeros en orden ascendente.
C     ( solo es permitido un maximo de 10 elementos )
C
      INTEGER A(10),OPT,RES
1     TYPE *,'TECLEA OPCION DE ORDENAMIENTO'
      TYPE *,'LETRAS --> 1       NUMEROS --> 2'
      ACCEPT *,OPT
      IF (OPT.LT.1.OR.OPT.GT.2) GOTO 55
      GOTO(5,20),OPT
5     TYPE *,'DAME EL NUMERO DE LETRAS A ORDENAR, MAXIMO 10'
      ACCEPT *,N
      TYPE *,'DAME LAS LETRAS, SIN BLANCOS INTERMEDIOS'
      ACCEPT 10,(A(I),I=1,N)
10     FORMAT(10A1)
      TYPE *,'ESTE ES EL VECTOR DESORDENADO'
      TYPE 15,(A(I),I=1,N)
15    FORMAT(1X,10(A1,2X))
      GOTO 30
20    TYPE *,'DAME EL NUMERO DE DATOS A ORDENAR, MAXIMO 10'
      ACCEPT *,N
      TYPE *,'DAME LOS DATOS, UNO POR RENGLON CON FORMATO ENTERO'
      TYPE *,'Y UN MAXIMO DE 4 DIGITOS'
      ACCEPT *,(A(I),I=1,N)
      TYPE *,'ESTE ES EL VECTOR DESORDENADO'
      TYPE 25,(A(I),I=1,N)
25    FORMAT(1X,10(I5,2X))
30    CALL ORDENA(N,A(1))     ! subrutina en MACRO-11
      TYPE *
      TYPE *,'ESTE ES EL VECTOR ORDENADO'
      GOTO(35,40),OPT
35    TYPE 15,(A(I),I=1,N)
      GOTO 45
40    TYPE 25,(A(I),I=1,N)
45    TYPE *
      TYPE *,'DESEAS HACER OTRO ORDENAMIENTO [Y/N] ?'
      ACCEPT 50,RES
50    FORMAT(A1)
      IF (RES.EQ.'Y') GOTO 1
      GOTO 60
55    TYPE *,'ERROR EN LA OPCION'
      GOTO 1
60    CALL EXIT
      END
```

```
>EDI ORDENAM.MAC
; Programa en MACRO-11 que realiza el ordenamiento de letras o numeros
; en orden ascendente por el metodo de la burbuja.
; Este programa sera ligado a ORDENAF.FTN
;
J:      .WORD 0             ; J <-- 0
I:      .WORD 0             ; I <-- 0
ORDENA::                    ; nombre de la subrutina en MACRO-11
        MOV @2(R5),R1       ; R1 <-- N , numero de elementos a ordenar
ETI1:   MOV 4(R5),R2        ; R2 contiene la direccion del elemento A(1)
        MOV #0,J            ; J <-- 0
        MOV #1,I            ; I <-- 1
ETI2:   CMP R1,I            ; mientras N<>I sigue
        BEQ ETI4            ; en caso contrario ve a ETI4
        CMP (R2),2(R2)      ; si A(I)<A(I+1)
        BMI ETI3            ; ve a ETI3
        MOV (R2),R3         ; en caso contrario AUX=A(I)
        MOV 2(R2),(R2)      ; A(I)=A(I+1)
        MOV R3,2(R2)        ; A(I+1)=AUX
        INC J               ; J <-- J+1
ETI3:   INC I               ; I <-- I+1
        ADD #2,R2           ; se mueve el apuntador al siguiente elemento
        BR ETI2             ; ve a ETI2
ETI4:   TST J               ; si J<>0
        BNE ETI1            ; ve a ETI1
        RTS PC              ; en caso contrario regresa al programa en FORTRAN
        .END
```

```
>FOR ORDENAF=ORDENAF

>MAC ORDENAM=ORDENAM

>TKB ORDENAF=ORDENAF,ORDENAM

>RUN ORDENAF
```

PROGRAMA # 6

```
>EDI BUSCAF.FTN

C   Programa en FORTRAN para realizar las operaciones de lectura y escritura
C   del programa BUSCAM.MAC
C   Este programa encuentra la posicion de un dato dentro de un vector con
C   un maximo de 10 elementos.
C
      INTEGER A(10),DATO,POSICION,BANDERA
      TYPE *,'DAME EL NUMERO DE DATOS DEL VECTOR, MAXIMO 10'
      ACCEPT *,NELE
      TYPE *,'DAME LOS ELEMENTOS ORDENADOS DE MENOR A MAYOR'
      TYPE *,'UNO POR RENGLON, CON FORMATO ENTERO'
      ACCEPT *,(A(I),I=1,NELE)
      TYPE *,'DAME EL NUMERO A BUSCAR, CON FORMATO ENTERO'
      ACCEPT *,DATO
      CALL BUSCA(NELE,DATO,POSICION,BANDERA,A) !  subrutina en MACRO-11
      IF (BANDERA.EQ.0) GOTO 15
      TYPE 10,DATO,POSICION
10    FORMAT(1X,'EL NUMERO',I6,' OCUPA LA POSICION',I3)
      GOTO 25
15    TYPE 20,DATO
20    FORMAT(1X,'EL NUMERO',I6,' NO SE ENCUENTRA EN EL VECTOR')
25    CALL EXIT
      END
```

```
>EDI BUSCAM.MAC

  Programa en MACRO-11 que busca la posicion de un dato en un vector -
; por el metodo de busqueda binaria.
; Este programa sera ligado a BUSCAF.FTN
;
Y:     .WORD 0         ; Y <-- 0
N:     .WORD 0         ; N <-- 0
D:     .WORD 0         ; D <-- 0
BUSCA::                ; nombre de la subrutina en MACRO-11
      MOV @2(R5),N     ; N <-- NELE, se carga el numero de elementos
      MOV @4(R5),D     ; D <-- DATO, se carga el dato a buscar
ETI1: MOV 12(R5),R2    ; R2 contiene la direccion del elemento A(1)
      MOV #2,R1        ; R1 <-- 1
      MOV Y,R0         ; R0 <-- Y
      ADD N,R0         ; R0 <-- R0+N
      CALL $DIV        ; subrutina para efectuar divisiones enteras
      MOV R0,R3        ; R3 <-- R0 , indice actual del vector
      CLC              ; se limpia bandera de carry
      ROL R0           ; realiza una rotacion a la izquierda(multiplicacion X 2)
      SUB #2,R0        ; R0 <-- R0-2
      ADD R0,R2        ; R2 <-- R2+R0 , direccion del elemento con indice R3
      CMP (R2),D       ; si (R2)<>DATO
      BNE ETI3         ; ve a ETI3
ETI2: CMP (R2),D       ; si (R2)=DATO
      BEQ ETI5         ; ve a ETI5
      CMP N,R3         ; si N=R3
      BEQ ETI5         ; ve a ETI5
      BR ETI1          ; en caso contrario ve a ETI1
ETI3: BMI ETI4         ; si (R2)<DATO ve a ETI3
      MOV R3,R4        ; R4 <-- R3 , salvamos indice actual
      SUB #1,R3        ; R3 <-- R3 -1
      MOV R3,N         ; N <-- R3
      MOV R4,R3        ; R3 <-- R4 , restauramos indice actual
      BR ETI2          ; ve a ETI2
ETI4: MOV R3,R4        ; R4 <-- R3 , salvamos indice actual
      ADD #1,R3        ; R3 <-- R3+1
      MOV R3,Y         ; Y <-- R3
      MOV R4,R3        ; R3 <-- R4 , restauramos indice actual
      BR ETI2          ; ve a ETI2
ETI5: CMP (R2),D       ; si (R2)<>DATO
      BNE ETI6         ; ve a ETI6
      MOV R3,@6(R5)    ; POSICION <-- R3 , posicion del dato
      MOV #1,@10(R5)   ; BANDERA <-- 1
      BR ETI7          ; ve a ETI7
ETI6: MOV #0,@10(R5)   ; BANDERA <-- 0
ETI7: RTS PC           ; regresa al programa en FORTRAN
      .END


>FOR BUSCAF=BUSCAF

 AC BUSCAM=BUSCAM

>TKB BUSCAF=BUSCAF,BUSCAM

>RUN BUSCAF
```

BIBLIOGRAFIA

- RSX-11M
  FORTRAN IV User's Guide

- IAS/RSX-11
  MACRO-11 Reference Manual

- RSX-11M
  Task Buider Reference Manual

- Processor Handbook
  pdp11/05/10/35/40

- OMSI PASCAL-1 User's Guide for RSX-11M

PRACTICA # 3

PROGRAMACION DE ENTRADA/SALIDA

PRACTICA # 3

PROGRAMACION DE ENTRADA/SALIDA

INTRODUCCION:

La capacidad de programar un computador para hacer calculos seria de poco uso si no existiera la forma de meter los datos a la maquina y conseguir los resultados de calculos realizados por ella. Por consiguiente, un prógramador debe estar provisto de los medios para transferir informacion entre el computador y los dispositivos perifericos que suministran la entrada o que sirven como medio de salida.

Con el fin de realizar una funcion de E/S, el programador debe especificar cuales son los datos, a donde deben ir o de donde vienen y como debe ser controlado el dispositivo de E/S. Dependiendo del tipo de computador que se utilice, la funcion de E/S puede requerir que el CPU espere hasta que la operacion de E/S se haya completado o la funcion de E/S puede permitir que el CPU continue procesando otras funciones mientras la operacion esta siendo realizada. Cuando la funcion E/S retiene el CPU decimos que la operacion de E/S esta entrecruzada con el CPU. Cuando ambas pueden ser realizadas simultaneamente, decimos que la E/S es concurrente con el proceso de computacion.

En otra forma, la funcion de E/S opera directamente entre la memoria y la unidad de E/S. Este modo de operacion requiere un camino separado [llamado camino de acceso de memoria directo(DMA)] entre la memoria y la unidad de E/S. El DMA permite que se realice la funcion de E/S con un minimo de dependencia del CPU.

La programacion de E/S depende de la maquina. La complejidad del sistema de E/S determina la complejidad de la programacion de E/S. En el PDP, la programacion de los dispositivos de E/S es extremadamente simple y no se requieren instrucciones nuevas de E/S para manejar las operaciones de E/S.

La clave de la simplicidad para la programacion de E/S es el UNIBUS. El UNIBUS permite una estructura de direccionamiento unificada en la cual el control, el estado y los registros de datos para los dispositivos perifericos son directamente direccionados como posiciones de memoria. Por consiguiente,todas las operaciones en los registros, como la transferencia de informacion hacia o fuera de ellos o la manipulacion de datos con ellos, son realizados por instrucciones normales de referencia a la memoria.

Todos los dispositivos perifericos estan especificados por un grupo de registros que son direccionados como la memoria y la manipulados con lexibilidad de un acumulador.Con cada dispositivo estan asociados dos tipos de registros:

1. Registros de control y estado.

2. Registros de datos.

Cada periferico tiene uno o mas registros de control y de estado (CSR), que contiene toda la informacion necesaria para la comunicacion con este dispositivo. Muchos dispositivos requieren menos de 16 bits de estado.Otros dispositivos requeririan mas de 16 bits y por lo tanto necesitaran registros adicionales de estado y control. Ca da dipositivo tiene al menos un registro buffer, ademas de los registros CSR, para el almacenamiento temporal de datos que han de ser transferidos desde o hacia el computador.

OBJETIVO:

El alumno aprendera a manejar dispositivos perifericos tales como lectora de tarjetas CR-11 y UDC11 (convertidores D/A), atraves de programas escritos en lenguaje FORTRAN y ensamblador MACRO-11.

DESARROLLO:

Realizar los siguientes programas:

Programa # 7 - Programacion de la lectora de tarjetas CR-11.
Programa # 8 - Programacion en ensamblador MACRO-11 de los convertidores Digital/Analogico.
Programa # 9 - Programacion en FORTRAN de los convertidores D/A para la utilizacion de un graficador mecanico.

PROGRAMA # 7

```
>EDI LECTF.FTN

C  Programa en FORTRAN para realizar la operacion de escritura del
C  programa LECTM.MAC, el cual realizara una lectura de tarjetas
C  perforadas sobre la lectora CR-11. Posteriormente escribira el
C  contenido de estas en la terminal.
C
      INTEGER R(64),C(64),H(80),B(80)
      INTEGER CONTA
C
C  conjunto de caracteres
C
      DATA C/' ','1','2','3','4','5','6','7','8','9','0',
     1        '#','@',':','>','?','A','B','C','D','E','F','G','H','I',
     2        '&','.',']','(','<','\','[','^','J','K','L','M','N','O',
     3        'P','Q','R','-','$','*',')',';','_','+','/','S','T','U',
     4        'V','W','X','Y','Z','`',',','%','=','"','!'/
C
C  codigo hollerith de los caracteres
C
      DATA R/0,256,128,64,32,16,8,4,2,1,512,66,34,130,522,518,2304,2176,
     1        2112,2080,2064,2056,2052,2050,2049,2048,2114,1154,2066,
     2        2082,2054,2178,1536,1280,1152,1088,1056,1040,1032,1028,
     3        1026,1025,1024,1090,1058,1042,1034,1030,2058,768,640,
     4        576,544,528,520,516,514,513,642,578,546,10,6,1154/
C
C
5     CALL LEE(H)  ! llamada a la subrutina en MACRO-11
C
C  checa si la tarjeta leida contiene 5 unos
C
      DO 15 I=1,5
        IF (H(I).NE.1) GOTO 15
        CONTA=CONTA+1
15    CONTINUE
C
C  se almacena la informacion leida en el vector B
C
      DO 30 I=1,80
        DO 20 J=1,64
          IF (H(I).NE.R(J)) GOTO 20
          B(I)=C(J)
          J=64
20      CONTINUE
30    CONTINUE
C
C  se escribe la informacion de la tarjeta leida
C
      TYPE 35,(B(I),I=1,80)
35    FORMAT(80A1)
C
C  si la tarjeta leida contiene 5 unos, se toma como senal de que ya
C  no hay mas tarjetas para leer y termina el programa, en caso
C  contrario prosigue la lectura.
C
      IF (CONTA.NE.5) GOTO 5
      CALL EXIT
      END
```

```
>EDI LECTM.MAC

; Este programa en MACRO-11 muestra un metodo de programar a la lectora
  de tarjetas CR-11, para efectuar la lectura de tarjetas perforadas
 /Este programa sera ligado a LECTF.FTN
CUENTA: .WORD 0           ; CUENTA=0
LEE::
      CRS=177160          ; registro de status de la lectora de tarjetas
      CRB=177162          ; buffer de datos (12bits)
      MOV #0,CUENTA       ; CUENTA <-- 0
      MOV 2(R5),R4        ; mueve el apuntador a la direccion del elemento H(1)
      MOV #CRS,R1         ; R1 contiene la direccion de CRS
      MOV #CRB,R2         ; R2 contiene la direccion de CRB
      MOV #80.,R3         ; R3 <-- 80. , numero de columnas a leer
ETI1: BIT @R1,#1400       ; checa si la lectora esta en linea
      BNE ETI1            ; en caso de no estar se va a ETI1 y repite este ciclo
                          ; hasta que se encuentre en linea
ETI2: MOV #1,@R1          ; lee una tarjeta
ETI3: BIT @R1,#140000     ; checa si hay condicion especial o si esta encendido
                          ; el bit "card done"
      BGT ETI2            ; condicion especial apagada, pero "card done" esta
                          ; encendido
      BEQ ETI4            ; ambas condiciones estan apagadas
      BR ETI5             ; en caso contrario ve a fin
ETI4: CMP CUENTA,R3       ; si cuenta=80.
      BEQ ETI5            ; ve a ETI5
      TSTB @R1            ; checa la siguiente columna
      BPL ETI3            ; si no esta lista se va a ETI3
      MOV @R2,(R4)+       ; se guarda la informacion leida
      INC CUENTA          ; se incrementa cuenta, contador de columnas leidas
      BR ETI4             ; ve a ETI4
ETI5: RTS PC              ; regresa al programa en FORTRAN
      .END
```

```
>FOR LECTF=LECTF

>MAC LECTM=LECTM

>TKB LECTF/AC=LECTF,LECTM
```

El switch /AC hace a la tarea privilegiada

```
>RUN LECTF
```

PROGRAMA # 8

```
>EDI CONVDA.MAC

; Programa en ensamblador MACRO-11 para la utilizacion de los
; convertidores-D/A del susbsistema UDC11.
; Este programa mueve una palabra de control a la direccion
; efectiva del modulo A633(D/A).
; Como entrada se tiene un dato digital, que es cargado atraves
; de la palabra de control, y como salida se tendra un voltaje
; analogico equivalente al dato.
;
;
;                     FORMATO DE LA PALABRA DE CONTROL
;
;
; -------------------------------------------------------------------------
; | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
; -------------------------------------------------------------------------
;  \___  ___/ \____________________  _____________________/ \_____  _____/
;      V                           V                              V
; selector de canal           datos 10 bits                   no usados
;
;
      .MCALL EXIT$S            ; directiva que nos permite usar la funcion de
                               ; de biblioteca EXIT.
                               ; $S indica que se crea dinamicamente en el stack
                               ; al momento de la ejecucion
 .I:  MOV #37777,@#171004      ; 37777 es el contenido de la palabra de control
                               ; y 171004 es la direccion del primer modulo A633
                               ; (contiene los canales 0,1,2 y 3).
                               ; la instruccion pone 10 volts al canal 0.
      EXIT$S                   ; directiva para que el sistema operativo
                               ; termine la ejecucion de la tarea
      .END ETI


>MAC CONVDA=CONVDA

>TKB CONVDA/AC=CONVDA

>RUN CONVDA
```

```
>EDI CIRCULO.FTN

        PROGRAMA QUE EJERCITA LOS CONVERTIDORES D/A
C
C
C
C       Este elabora una serie de circunferencias de radio variable
C       mediante la utilizacion de los convertidores D/A y un grafi-
C       cador mecanico.
C
        TYPE 10
10      FORMAT('   CUAL ES EL NUMERO DE CIRCULOS A GRAFICAR  ')
        ACCEPT 20,NUMVE
20      FORMAT(I1)
C
        DO 30 I=1,NUMVE
                CALL EJER
30      CONTINUE
C
        CALL EXIT
        END
C
C
C       SUBRUTINA EJERCITADORA DEL UDC (CONVERTIDORES D/A)
C
C
        SUBROUTINE EJER
C
        DIMENSION IVOLT(1022),ISB(2),IDATA(1022),INFER(1022)
        TYPE 40
40      FORMAT(5X,'***PRUEBA DE LOS CONVERTIDORES A VOLTAJE***',////)
50        TYPE *,'ESPECIFICA EL RADIO EN UN RANGO DE'
         TYPE *,'        0.05  <=  RADIO  <=  5.0     '
         ACCEPT 55,RADIO
55       FORMAT(F5.2)
         IF(RADIO.LT.0.05)GO TO 50
         IF(RADIO.GT.5.0) GO TO 50
C
C        Calculo de todos los puntos que seran graficados
C                  con los convertidores
C
C
C
C       Niveles de referencia.
C
C
1       INM=1
         IGENO=1023.*5./10.
         IREST=1023.*RADIO/10.
         IDESP=IGENO-IREST
         IREST=IREST+IDESP
         CREMEN=2.*RADIO/1022.
         ICONT=1
C
C       Desplaza el canal 1 a un nivel 5V de referencia
```

```
C
      CALL AO(INM,ICONT,IGENO,ISB)
      VOLTS=0.
      ICONT=0
C
C     Se desplaza el canal  0 a el radio especificado
C
      CALL AO(INM,ICONT,IDESP,ISB)
C
C     Calcula los puntos de la circuferencia
C
      DO 60 I=1,511
      Y=SQRT(RADIO**2-VOLTS**2)
C
C     Puntos superiores
C
      IDATA(512-I)=Y*1023./10.+IREST
      IDATA(511+I)=IDATA(512-I)
C
C     Puntos inferiores
C
      INFER(512-I)=IREST-Y*1023./10.
      INFER(511+I)=INFER(512-I)
C
C     Desplazamiento horizontal
C
      IVOLT(I)=VOLTS*1023./10.+IDESP
      IVOLT(511+I)=IVOLT(I)+IREST-IDESP
60    VOLTS=VOLTS+CREMEN
C
C
C     Los puntos calculados son puestos en los convertidores D/A
C     mediante la subrutina CALL AO(INM,ICONT,IDATA,ISB) en donde:
C              INM - # de canales que tendran salida simultanea de
C                       acuerdo a el dato de entrada.
C            ICONT - Son los canales a los que se hace referencia.
C             IDATA - Es el dato de entrada .
C               ISB - Es un campo que indica si hubo error o no en
C                      la operacion efectuada.
C
C
      DO 70 I=1,1022
      ICONT=1
      CALL AO(INM,ICONT,IDATA(I),ISB)
      ICONT=0
      CALL AO(INM,ICONT,IVOLT(I),ISB)
C
C     Retraso de tiempo para adecuar el tiempo de respuesta
C     de el graficador con el de la maquina.
C
      DO 70 L=1,10
      DO 70 M=1,10
70    CONTINUE
```

```
      DO 80 I=1,1022
      ICONT=1
      CALL AO(INM,ICONT,INFER(I),ISB)
      ICONT=0
      CALL AO(INM,ICONT,IVOLT(1023-I),ISB)
C
C    Retraso de tiempo para adecuar el tiempo de respuesta
C    de el graficador con el de la maquina.
C
      DO 80 L=1,10
      DO 80 M=1,10
80    CONTINUE
      ICONT=1
      CALL AO(INM,ICONT,IGENO,ISB)
      RETURN
```

```
>FOR CIRCULO=CIRCULO

>TKB <cr>

TKB>CIRCULO=CIRCULO

TKB>/

ENTER OPTIONS:

TKB>COMMON=UDCOM:RW

TKB>//

>RUN CIRCULO
```

BIBLIOGRAFIA

- RSX-11M
  FORTRAN IV User's Guide

- IAS/RSX-11
  MACRO-11 Reference Manual

- RSX-11M
  Task Buider Reference Manual

- Processor Handbook
  pdp11/05/10/35/40

- RSX-11M
  I/O Drivers Reference Manual
  Cap. Universal Digital Controller Driver

DIVISION DE EDUCACION CONTINUA
FACULTAD DE INGENIERIA U.N.A.M.

INTRODUCCION A LAS MINICOMPUTADORAS PDP-11

APLICACIONES

ING. JORGE IVAN EUAN AVILA

NOVIEMBRE, 1984.

Escrito para el curso de Introduccion a la PDP-11, Educacion Continua de la F. de I. UNAM. Jorge I Euan Avila.

APLICACIONES

I.- Arquitectura de las Bases de Datos.

Los sistema de bases de datos, evolucionaron en una primera etapa de simples manejadores de archivos a manejadores que permitieran ligar archivos y poder tener acceso a la información desde otras aplicaciones. En esta primera etapa el diseño de una base de datos fue considerado como la especificacion de los registros y su organizacion en los dispositivos de almacenamiento secundario. Con el objeto de mejorar la independencia de los datos y estructurar el proceso de diseño de una base da datos en 1971 CODASYL/DBTG propuso una arquitectura de dos niveles (Esquema-Subesquema), la cual fue seguida en 1975 por una arquitectura de tres niveles (Externo-Conceptual-Interno) propuesto por ANSI/X3/SPARC.

I.1- CODASYL/DBTG.- La propuesta de este grupo fue un primer intento por estandarizar el diseño de las bases de datos y de agrupar los aspectos relacionados con el usuario en varios "subesquemas", mientras que la vista total y los aspectos relacionados con el almacenamiento fueron especificados en el "esquema". La separacion sin embargo no fue completa, como puede ser evidente en algunas instrucciones del lenguaje de manipulacion de datos y del lenguaje de definición de datos. Esta estructura aunque es la base de sistemas comerciales presenta ciertos problemas:

a) El diseño de la base no es transparente al usuario ya que tiene que estar enterado de los mecanismos de acceso.

b) Cambios en las estrategias de almacenamiento afectan al usuario.

c) Los sistemas estan restringidos al modelo de red.

d) Su operación resulta muy eficiente a expensas de ser flexible en el diseño lógico de la base.

I.2- ANSI/X3/SPARC.- Este grupo, formuló una arquitectura que permitiera una independencia completa de los datos. Esto sería logrado usando tres esquemas : un esquema externo que presentaría al usuario una vista parcial de los datos similar al subesquema de codasyl, un esquema conceptual que es global el cual proporciona una vista imparcial de todo el sistema y que es independiente de las vistas de los usuarios, asi como del alamcenamiento físico; y un esquema interno el cual es un plan detallado de la estructura de almacenamiento físico. Esta arquitectura, trato de identificar los componentes que garantizaran una flexibilidad máxima y una independencia total de los datos. La parte innovadora de esto, es el esquema conceptual, el cual aisla el almacenamiento físico y los mecanismos de acceso del las vistas del usuario de modo tal que se logra la independencia total de los datos. Esta arquitectura recibe el nombre de coexistencial.

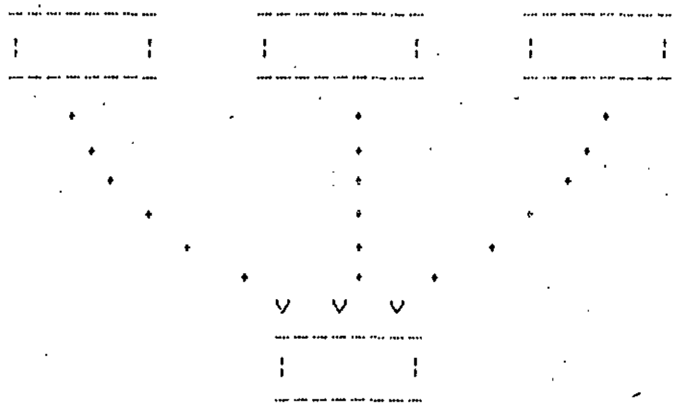
II.- Diseño del esquema conceptual.

El esquema conceptual, es la parte central del modelo de ANSI/X3/SPARC y la tarea de definir las entidades y sus relaciones constituyen el proceso de diseño del esquema conceptual. Debido a los modelos que se utilizan, el esquema conceptual algunos autores lo han dividido en dos partes: la vista global representada con un modelo independiente y el esquema lógico definido con las instrucciones del manejador (DBMS).

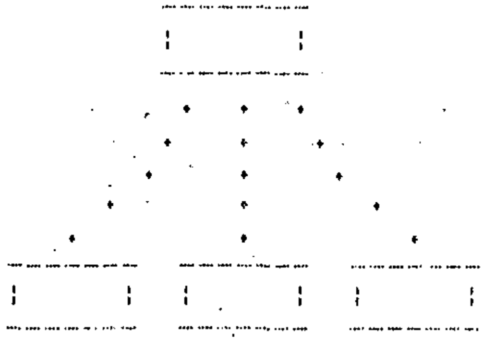
III.- Modelos.

Para la representación de las vistas globales los modelos que más se han Utilizado son los siguientes:

Redes (networks).- El modelo de red es una gráfica dirigida en la que los nodod representan entidades y los arcos representan asociaciones.



Jerárquico.- Este modelo es un árbol en el que los nodos representan entidades y los arcos asociaciones.



Relacional.- En este modelo los datos estan organizados en relàciones. Una relación esta definida como un subconjunto del producto cartesiano.

$$R \subset D1 \times D2 \times \ldots \times Dn$$

Binario asociativo.- En este modelo no se distinguen entidades, atributos ni relaciones. Los únicos elementos de este modelo son "objetos" y "asociaciones" entre pares de objetos.

IV.- Manejadores Comerciales.

Los menejadores de Bases de Datos comerciales se apegan a las arquitecturas y modelos mencionados con ligeras particularidades. Uno de estos manejadores es TOTAL desarrollado por CINCOM bajo la filosofia de modelo de red. Su arquitectura, podemos decir que no se apega a las dos expuestas anteriormente por ser uno de los primeros manejadores comerciales.

V.- Total.

Las bases de datos que se manejan consisten de un grupo de archivos llamados data-set dentro de los cuales identificamos dos tipos : los maestros t los variables. Los archivos maestros son independientes y se accesan los registros por la llave de control; los variables son dependientes y estan ligados a uno o mas archivos maestros. Los registros del archivo variable estan encadenados por grupos y cada grupo esta ligado a un registro de un archivo maestro. Esta liga es la que proporcina el mecanismo de acceso a los registros variables. Veasè la siguiente figura.

```
mmmmmmmmmmmmmm .........vvvvvvvvvvvvv
                                    .
                                    .
                        vvvvvvvvvvvvv
                                    :
                                    .
                        vvvvvvvvvvvvv
                                    .
                                  etc.
```

Un archivo maestro puede tener asociado varios variables y un variable puede tener asociado varios maestros.Vease la siguiente figura.

```
                         ............var1
                         .
                         ............var2
      maestro............
                         .............var3
                         .
                         .
                        etc.
```

```
   mst1................
                       .
   mst2................
                       .......variable
   mst3................
                       .
                       .
                      etc.
```

Asociaciones entre registros.

El registro maestro contiene informacion para el manejo de dos trayectorias, una que apunta al primer registro variable y otra que apunta al ultimo registro variable; cada registro variable contiene informacion para el manejo de dos trayectorias, una para el antecesor y otra para el sucesor directo. Vease la siguiente figura.

```
M..........> V1 ...........> V2 .........> V3 ......          ....> Vn

M..........> Vn ...........> Vn-1 .......> Vn-2 ....         .....> V1
```

Formato de los registros Maestros y Variables.

Maestro(root,key,link1,link2,.....,data1,data2,..........).

root- la asignación de los registros al area de alamacenamiento la hace total utilizando una función de Hash y para el manejo de las colisiones se apoya en el campo root para hacer una lista ligada de registros sinonimos.

key- es el campo que ha sido definido como llave del registro.

link1,link2,... son los apuntadores a los diferentes archivos variables con los que puede estar relacionado el maestro.

data1,data2,... son los datos del registro.

```
Variable1(key1,link1,key2,link2,......,data1,data2,.........).
```

key1,key2,... son las llaves de control de cada maestro que esta asociado con este variable.

link1,link2,... son las trayectorias asociadas a cada maestro que esta relacionado con este variable.

data1,data2,... son los datos del variable.

```
Variable2(|code,key1,link1,data1,|key2,link2,key3,link3,data2,data3|)
    datos-base                        datos-redefinidos
```

Los registros codificados permiten tener un registro con varios formatos. Para implementarlos se define un area de datos-base y otra de datos-redefinidos. Los datos base son definidos en todos los formatos diferentes y los datos-redefinidos son los que cambian en cada caso.

V.1.- Lenguaje de definición de datos.

El lenguaje de definicion de datos es un conjunto de instrucciones con las cuales se declara y describe la base de datos. Estas instrucciones permiten que el usuario declare: los nombres de sus archivos, los campos de sus registros y las caracteristicas del medio ambiente. Vea a continuacion la explicacion de cada una de las instrucciones del lenguaje. El orden en el que aparecen es importante y debe ser respetado.

B.

```
@@ BEGINGEN
      EXPRESIONES PARA INICIAR LA GENERACION DE BASES DE DATOS

     BEGIN-DATA-BASE-GENERATION

     DATA-BASE-NAME=XXXXXX (SE PONE UN NOMBRE DE 6 CARACTERES PARA LA
                           BASE DE DATOS).

     OPTIONS:LOG=X,OUTPUT=X,QUEUE=N
             (CON LOG= SE ESPECIFICA SI SE DESEA QUE HAYA LOGGING )
             (PONER LOG=Y PARA SI O LOG=N PARA NO; VEASE SOS TOT LOGGING )
             (CON OUTPUT= SE ESPECIFICA SI SE DESEA LA SALIDA DEL ARCHI-
              VO EN MACRO-11, PONER =D PARA SI O =N PARA NO)
                  (VEASE SOS TOT DBGEN)
             (CON QUEUE=N SE ESPECIFICA EL NUMERO MAXIMO DE REGISTROS
              QUE PODRAN SER ENCOLADOS, O SEA RESERVADOS; EL DEFAULT
              SON 10 REGISTROS)

     OPTIONS:TASKS=N,TIMEOUT=N,PASSWORD=XXXXXX,EXT=XXX
             (CON TASKS=N SE ESPECIFICA EL NUMERO MAXIMO DE TAREAS, O SEA
              DE PROGRAMAS DE APLICACION QUE PODRAN ESTAR USANDO EL
              DBMOD DE ESTA BASE DE DATOS AL MISMO TIEMPO)
                 \(VEASE >SOS TOT DBMOD)
             (CON TIMEOUT=N SE ESPECIFICA EL TIEMPO MAXIMO EN SEGUNDOS
              QUE PODRA RETENERSE UN REGISTRO; UN O INDICARA QUE NO HAY
              TIEMPO MAXIMO DE RETENCION ).
             (CON PASSWORD=XXXXXX SE PROPORCIONA UN PASSWORD O CLAVE

              CON LA CUAL SOLAMENTE PODRA SER DESACTIVADO EL DBMOD)
             (CON EXT=XXX SE ESPECIFICAN 3 CARACTERES QUE SE DESEAN
              ESTEN PUESTOS COMO EXTENSION DE LOS ARCHIVOS DE LA BASE
              DE DATOS Y DE LOS ARCHIVOS DE LOGGING).

      SHARE-IO     (ESTA EXPRESION INDICA QUE EMPEZARAN A SER DEFINIDAS
                    LAS AREAS DE I/O QUE ESPECIFICAMENTE SERAN USADAS EN
                    LA DEFINICION DE REGISTROS DENTRO DE ESTE DBMOD).

     IOAREA=XXXX=N
                  ( XXXX SERA UN NOMBRE DE 4 CARACTERES PARA LA
                    IOAREA Y QUE SERA USADA EN LA DEFINICION DE
                    DATA SETS EN ESTE DBMOD)
                   ( N PODRA PONERSE OPCIONALMENTE PARA ESPECIFI-
                     CAR EL NUMERO DE COPIAS QUE SE DESEAN DE ESTA
                      I/O AREA).

     END-IO      MARCA LA TERMINACION DE LA DEFINICION DE I/O AREAS.

@@ MDS
    A CONTINUACION VIENEN LAS EXPRESIONES PARA LA ESPECIFICACION
O DECLARACION DE DATA SETS MAESTROS (SINGLE ENTRY).
```

BEGIN-SINGLE-ENTRY-DATA-SET (MARCA EL COMIENZO DE LA DESCRIPCION DE DATA SETS MAESTROS).

DATA-SET-NAME=MMMM (DONDE MMMM SERA UN NOMBRE DE 4 CARACTERES CON EL CUAL SERA RECONOCIDO EL DATA SET).

IOAREA=XXXX (DONDE XXXX SERA EL NOMBRE DE LA IOAREA QUE SERA ASIGNADA AL DATA SET; UNA AEREA PUEDE SER ASIGNADA A DATA SETS MAS DE UNA VEZ SOLO CUANDO SEAN DEL MISMO TIPO YA SEA MAESTROS O VARIABLES).

MASTER-DATA (MARCA EL COMIENZO DE LA DEFINICION DE LOS ELEMENTOS DEL DATA SET)

MMMMROOT=8 (DONDE MMMM DEBE SER SUSTITUIDO POR EL NOMBRE DEL DATA SET MAESTRO QUE SE ESTA DEFINIENDO Y EL =8 ESTA INDICANDO QUE ESTE ELEMENTO TENDRA UNA LONGITUD DE 8 YA QUE ES UN ELEMENTO DE CONTROL EL CUAL TIENE UN USO INTERNO PARA EL MANEJO DE SINONIMOS, DEBIENDO SER EL PRIMER ELEMENTO DEL REGISTRO.

MMMMCTRL=N (DONDE MMMM ES EL NOMBRE DEL DATA SET Y N SERA LA LONGITUD DESEADA PARA LA LLAVE DE CONTROL, VEASE >SOS TOT CTRL).

MMMMLKXX=8 (DONDE MMMM ES EL NOMBRE DEL DATA SET; XX ES UN CODIGO DE 2 CARACTERES PARA IDENTIFICAR EL CAMINO DE LIGA Y =8 ES UNA LONGITUD OBLIGATORIA DE 8 CARACTERES PARA EL ELEMENTO DE LIGA).

DESCRIPCION- ESTE ELEMENTO DEFINIRA UN CAMINO DE LIGA DEL DATA SET (VEASE >SOS TOT LIGA):

A) UN DATA SET MAESTRO PUEDE TENER CUALQUIER NUMERO DE LIGAS
B) UN DATA SET DE ENTRADA VARIABLE PUEDE ESTAR LIGADO DESDE VARIAS LIGAS DEL MISMO DATA SET MAESTRO
C) A LOS CAMINOS DE LIGA NUNCA SE LES DA UN NIVEL
D) LOS CAMINOS DE LIGA SERAN ALINEADOS A FRONTERA DE BYTE.

(.P.)MMMMXXXX=N .P.= ES UN VALOR NUMERICO OPCIONAL QUE ESPECIFICA UN NIVEL PARA EL ELEMENTO.
MMMM= NOMBRE DEL DATA SET MAESTRO.
XXXX= IDENTIFICADOR DEL ELEMENTO.
N= LONGITUD DESEADA DEL ELEMENTO.

NOTA: CON ESTA EXPRESION SE DEFINE UN ELEMENTO O CAMPO QUE ADEMAS PUEDE SER SUBDIVIDIDO EN SUB-ELEMENTOS DANDOLES A ESTOS UN SUB-NIVEL Y ESTOS, A SU VEZ, TAMBIEN PUEDEN SER SUDIVIDIDOS.

A)LOS NIVELES PURDEN SER DEL 1 AL 5, EL NIVEL 0 ESTA RESERVADO Y CUALQUIER ELEMENTO QUE NO TENGA NUMERO DE NIVEL LE ES ASIGNADO EL NIVEL 0.

B) LA LONGITUD DE UN ELEMENTO QUE ESTA SUBDIVIDIDO DEBE SER LA LONGITUD TOTAL DE TODOS LOS ELEMENTOS QUE LO SUBDIVIDEN.

C) EL NOMBRE O IDENTIFICADOR DEL ELEMENTO DEBE SER UNICO.

END-DATA MARCA LA TERMINACION DE LA ESPECIFICACION DE LOS ELEMENTOS DEL DATA SET MAESTRO.

UIC=[N,N] UIC SIGNIFICA USER IDENTIFICACION CODE Y SE TRATA DEL NUMERO DE CUENTA (CLAVE) QUE TIENE UN USUARIO Y BAJO LA CUAL SON GUARDADOS TODOS SUS ARCHIVOS EN UN DIRECTORIO. CON ESTA EXPRESION SE INDICA EN QUE CUENTA SE DESEA QUE SEAN CREADOS Y ALMACENADOS LOS ARCHIVOS DEL DATA SET Y POR DEFAULT AL NO ENCONTRARSE ESTA EXPRESION ASUME LA CUENTA BAJO LA CUAL SE ESTA HACIENDO LA COMPILACION.

DEVICE=XXXX. XXXX= ES EL TIPO DE DISPOSITIVO EN DONDE RESIDIRA EL DATA SET, PONER RK07.

RETRIEVAL-POINTERS=N N= NUMERO DE RETRIEVAL POINTERS (APUNTADORES DE RETIRO) PARA EL ARCHIVO DEL DATA SET; POR DEFAULT, EN CASO DE NO ENCONTRARSE, SE ASUME 7.

TOTAL-LOGICAL-RECORDS=N N= NUMERO MAXIMO DE REGISTROS QUE SE ESPERAN TENER EN EL DATA-SET.

NOTA: N ES DEPENDIENTE DEL NUMERO DE REGISTROS POR BLOQUE, DE LA LONGITUD DEL REGISTRO Y DEL TIPO DE DISPOSITIVO (DRIVE) ESPECIFICADOS ANTERIORMENTE.

LOGICAL-RECORD-LENGTH=N N= LONGITUD DESEADA PARA EL REGISTRO, DEBIENDOSE TOMAR EN CUENTA TODOS LOS,CAMPOS DE CONTROL PARA LA LONGITUD FINAL.

CONTROL-INTERVAL=N N= NUMERO DE REGISTROS QUE SERAN PUESTOS EN UN CILINDRO LOGICO, SI LA EXPRESION ES OMITIDA EL DEFUALT ASUMIDO SERA EL NUMERO DE REGISTROS QUE CABEN EN UN CILINDRO FISICO.

DRIVE= N,N,XXN   LA PRIMERA N DEBERA SER EL NUMERO LOGICO DE LA UNIDAD QUE SE DESEA SEA ASOCIADA CON ESTA SECCION (ESTE NUMERO DEBE SER UNICO O SEA NO DEBE ESTAR REPETIDO EN LA BASE DE DATOS).
LA SEGUNDA N DEBERA SER EL NUMERO DE SECTORES FISICOS QUE SE DESEAN SEAN ASIGNADOS A ESTA SECCION (UN VALOR DE 40 ES BUENO).
XXN= DISPOSITIVO Y NUMERO DE UNIDAD PARA ASIGNAR A ESTA SECCION (PARA EL CASO DEL LABORATORIO DE COMPUTACION PONER: DBO.SYO,1,2... ETC).

NOTA: SE PUEDEN USAR MAS DE UNA EXPRESION DE DRIVE SI EL DATA SET LO REQUIERE, CON UN MAXIMO DE 32 PERMITIDOS.

@@ VDS

A CONTINUACION VIENEN LAS EXPRESIONES PARA LA DEFINICION DE DATA SETS VARIABLES :

BEGIN-VARIABLE-ENTRY-DATA-SET   INDICA EL COMIENZO DE LA DEFINICION DE UN DATA SET VARIABLE.

DATA-SET-NAME=VVVV   VVVV= NOMBRE DE 4 CARACTERES QUE SERA

USADO COMO IDENTIFICADOR DEL DATA SET VARIABLE.

IOAREA=XXXX   XXXX= NOMBRE DE 4 CARACTERES ALFANUMERICOS DE LA AREA DE ENTRADA SALIDA (I/O) QUE SERA USADA POR EL DATA SET VARIABLE.

VEASE LA EXPRESION DE IOAREA PARA DATA SET MAESTRO.

BASE-DATA   ESTA EXPRESION INDICA EL COMIENZO DE LA DEFINICION DE LOS ELEMENTOS O CAMPOS QUE FORMAN PARTE DE LOS REGISTROS DEL DATA SET VARIABLE.

VVVVCODE=2   VVVV= NOMBRE DEL DATA SET VARIABLE; ESTA EXPRESION ES REQUERIDA PARA RESERVAR ESPACIO EN EL REGISTRO PARA EL CODIGO DEL REGISTRO.

(.P.)VVVVXXXX=N   .P.= ES OPCIONAL Y ESPECIFICA EL NIVEL QUE SE DESEA PARA EL ELEMENTO, VEASE LA MISMA EXPRESION PARA DATA SET MAESTRO.
VVVV= NOMBRE DEL DATA SET VARIABLE.
XXXX= CUATRO CARACTERES QUE SIRVEN PARA IDENTIFICAR AL ELEMENTO.

NOTA: VEASE LA MISMA EXPRESION PARA DAT SET MAESTRO.

MMMMLKXX=8=VVVVXXXX   MMMM= NOMBRE DE UN DATA SET MAESTRO.
XX= DOS CARACTERES PARA CODIGO DE LIGA
8= LONGITUD OBLIGATORIA DE 8.

ESTA EXPRESION DEFINE UN CAMINO DE LIGA DESDE UN DATA SET MAESTRO EN LA LLAVE DE CONTROL DEL REGISTRO. LA DEFINICION

DEL ELEMENTO QUE CONTIENE ESTA LLAVE DEBE PRECEDER A ESTA EXPRESION.

A) MMMMLKXX ES EL CAMINO DE LIGA COMO FUE DEFINIDO EN EL DATA SET MAESTRO QUE ES LIGADO AL DATA SET VARIABLE. ESTA ENTRADA DEBE SER ESPECIFICADA EXACTAMENTE COMO EN EL DATA SET MAESTRO PARA PODER ESTABLECER LA LIGA REQUERIDA.

B) VVVVXXXX ES EL ELEMENTO DEFINIDO EN ESTE REGISTRO QUE CONTIENE LA LLAVE DE CONTROL PARA ESTE CAMPO LIGA.

C) UN DATA SET VARIABLE PUEDE SER LIGADO DESDE VARIOS DATA SETS MAESTROS Y PUEDE TENER LIGAS MULTIPLES DESDE EL MISMO DATA SET MAESTRO.

D) A LOS CAMINOS DE LIGA NUNCA SE LES DA UN NUMERO .

E) LOS CAMINOS DE LIGA SON ALINEADOS SI ES REQUERIDO EN BYTES PAR.

RECORD-CODE=XX

ESTA EXPRESION ES OPCIONAL E INDICA EL COMIENZO DE LA DEFINICION DE UN GRUPO DE ELEMENTOS QUE REDEFINEN AL ULTIMO ELEMENTO DE LA PARTE DE BASE-DATA. (DESCRITA ANTERIORMENTE EN ESTE TEXTO) EL CODIGO DE DOS CARACTERES IDENTIFICA A LA REDEFINICION.

A) LA PARTE REDEFINIDA DEL REGISTRO PUEDE DEFIRIR DE UN REGISTRO CODIFICADO A OTRO EN EL MISMO DATA SET, A DIFERENCIA DE LA PARTE DE BASE-DATA QUE NO CAMBIA EN TODO EL DATA SET.

B) SE PUEDE REDEFINIR CUANTAS VECES SEA NECESARIO PERO CADA REDEFINICION DEBERA ESTAR IDENTIFICADA POR UN CODIGO DIFERENTE.

C) LOS CODIGOS DE REGISTRO NO DEBEN TENER UN NUMERO DE NIVEL.

.P.VVVVXXXX=N VEASE LA EXPLICACION CORRESPONDIENTE EN LA PARTE DE BASE-DATA.

YA QUE LOS ELEMENTOS EN LA PORCION DE RECORD-CODE ESTAN REDEFINIENDO AL ULTIMO ELEMENTO DE LA PORCION DE BASE-DATA DEL REGISTRO, ESTOS DEBEN TENER UN NUMERO DE NIVEL.

MMMMLKXX=8=VVVVXXXX

VEASE LA EXPLICACION CORRESPONDIENTE EN LA PARTE DE BASE-DATA.

END-DATA INDICA LA TERMINACION PARA LA DEFINICION DE LOS ELEMENTOS DEL DATA SET.

DEVICE=RK07 ESPECIFICA EL TIPO DE MEMORIA DE ACCESO DIRECTO EN DONDE RESIDE EL DATA SET.

TOTAL-LOGICAL-RECORDS=N

N= NUMERO TOTAL DE REGISTRO QUE SE TENDRAN EN EL DATA SET.

LOGICAL-RECORD-LENGTH=N N= LONGITUD DEL REGISTRO.
VEASE LA EXPLICACION CORRESPONDIENTE AL REGISTRO MAESTRO.

LOGICAL-RECORDS-PER-BLOCK=N
VEASE LA DECRIPCION CORRESPONDIENTE PARA REGISTRO MAESTRO.

CONTROL-INTERVAL=N
VEASE LA DESCRIPCION CORRESPONDIENTE PARA REGISTRO MAESTRO.

DRIVE= N,N,XXN
VEASE LA DESCRIPCION CORRESPONDIENTE PARA REGISTRO MAESTRO.

LOAD-LIMIT=N N= PORCENTAJE EXPRESADO COMO VALOR ENTERO.
SE TRATA DE UN PORCENTAJE PARA SER USADO EN EL MANEJO DE ESPACIO Y EL DE DEFAULT ES DE '80'.

UIC=[N,N]
VEASE LA DESCRIPCION CORRESPONDIENTE PARA REGISTRO MAESTRO.

RETRIEVAL-POINTERS=N
VEASE LA DESCRIPCION CORRESPONDIENTE PARA REGISTRO MAESTRO.

END-VARIABLE-ENTRY-DATA-SET
INDICA LA TERMINACION PARA LA ESPECIFICACION DE UN DATA SET DE ENTRADA VARIABLE.

@@ LDS
EXPRESIONES PARA DATA SETS DE LOGGING A DISCO.

BEGIN-DISK-LOG-DATA-SET
ESTA DEBE SER LA PRIMERA EXPRESION PARA EMPEZAR LA DEFINICION DE DATA SETS DE LOGGING A DISCO.

DATA-SET-NAME=XXXX
SE PROPORCIONA UN NOMBRE DE CUATRO CARACTERES ALFANUMERICOS QUE ES USADO PARA IDENTIFICAR EL DATA SET DE LOGGING.

LOG-BLOCKSIZE=N
SE PROPORCIONA UN VALOR NUMERICO ENTRE 1 Y 10 QUE DEFINE EL TAMA&O DEL BUFFER DE LOG EN SEGMENTOS DE 1024 BYTES PARA SER USADOS POR LOGGING; EL DEFAULT ES 1.

LOG-LOAD-LIMIT=N
SE PROPORCIONA UN VALOR NUMERICO ENTRE 1 Y 100 QUE ESPECIFICA QUE PORCENTAJE DEL ARCHIVO DE LOGGING SERA UTILIZADO ANTES DE QUE EL ARCHIVO EMPIECE A SER USADO DE NUEVO (FILE FLIP-FLOP).

UIC=[N,N]
SE PROPORCIONA UNA CUENTA VALIDA QUE ESPECIFICA EN QUE DIRECTORIO SERAN CREADOS Y ALOJADOS LOS DATA SETS DE LOGGING.

DRIVE=N,N,XXN

PARA LOS DATA SETS DE LOGGING SE REQUIEREN DOS EXPRESIONES DE DRIVE.
VEASE LA EXPLICACION DE DRIVE EN LA DEFINICION DE REGISTROS MAESTROS.

END-DISK-LOG-DATA-SET

ESTA DEBE SER LA ULTIMA EXPRESION PAR DAR POR TERMINADA LA DEFINICION DEL DATA SET DE LOGGING A DISCO.

@@ ENDDBGEN

EXPRESION PARA TERMINAR LA GENERACION DE UNA BASE DE DATOS

END-DATA-BASE-GENERATION

V.2.- lenguaje de manipulación de datos.

Con las caracteristicas de este lenguaje el usuario esta en posibilidades de manejar los datos de la base para implementar sus aplicaciones. Las instrucciones se implementan como llamadas a subrutinas desde el lenguaje donde se esta programando la aplicación, llamado lenguaje "huesped o anfitrión". TOTAL puede ser utilizado desde FORTRAN, COBOL, ASEMBLER o PASCAL. En FORTRAN, la subrutina se llama DATBAS y con el siguiente ejemplo ilustramos algunos de sus parametros.

```
call datbas(readm,stat,cust,key,elem-list,user-area,endp).
```

readm-nombre de la función a realizar. En este caso lectura de un registro maestro.

stat-regresa datos sobre la realización de la operación.

cust-nombre del archivo maestro sobre el que se opera.

key-llave del registro que se quiere accesar.

elem-list-lista de los atributos que se desean.

user-area-contenido de los atributos despues de la operación de lectura.

endp-indica el fin de la lista de parametros.

Las operaciones que se pueden realizar con TOTAL se muestran en la siguiente tabla:

E.3 A List of DML Commands

E.3.1 Serial Processing Functions

The processing of records one by one according to their physical sequence in a data set by repeating the same function.

1. RDNXT: Serially read a master or a varaiable entry data set.

E.3.2 Master Data Set Functions

The processing of a record from a master data set.

1. READM: Read a master record
2. WRITM: Write a master record.
3. ADD-M: Add a master record.
4. DEL-M: Delete a master record.

E.3.3 Variable Data Set Functions

The processing of a record from a variable data set.

1. READV: Read a variable record along the forward direction of a variable record chain.
2. READR: Read a variable record along the reverse direction of a variable record chain.
3. READD: Read a variable record directly by specifying its position.
4. WRITV: Write the variable record retrieved by the preceding read.
5. ADDVC: Add a variable record to the end of a chain.
6. ADDVB: Add a variable record before the one retrieved by the preceding read.
7. ADDVA: Add a variable record after the one retrieved by the preceding read.
8. DELVD: Delete the variable record retrieved by the preceding read.

E.3.4 Special Functions

1. SINON: Sign-on a program
2. SINOF: Sign-off a program
3. RQLOC: Request the home location of a master record
4. WRITD: Write a master or variable record directly into a specific location.
5. QUIET: Check point the data base and log device.
6. LOADD: Load a data base descriptor
7. FREEF: Free held records for a file
8. FREEX: Free all held records for this program.

A master record is chained to a group of variable records with respect to control key K.

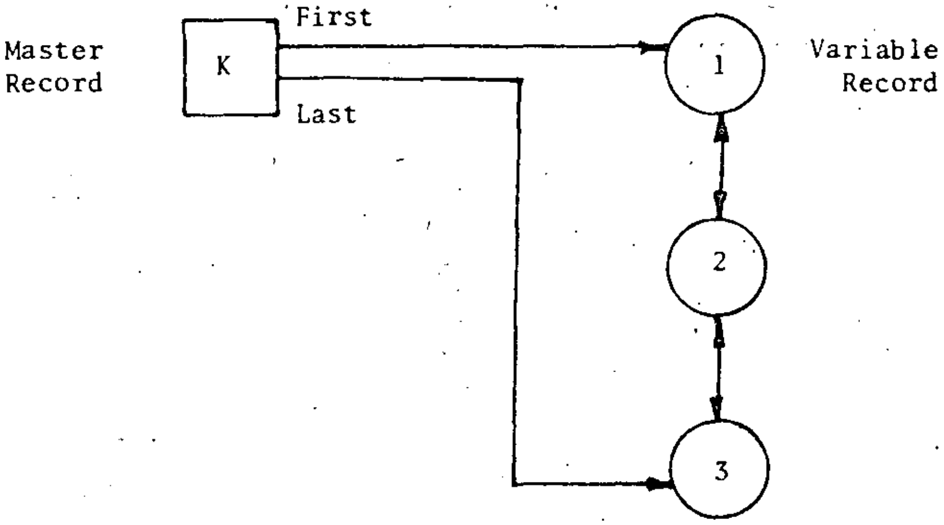


FIGURE 2-12: A SINGULAR RECORD CHAIN

An Add-Continue function will add a variable record to the bottom of the chain.

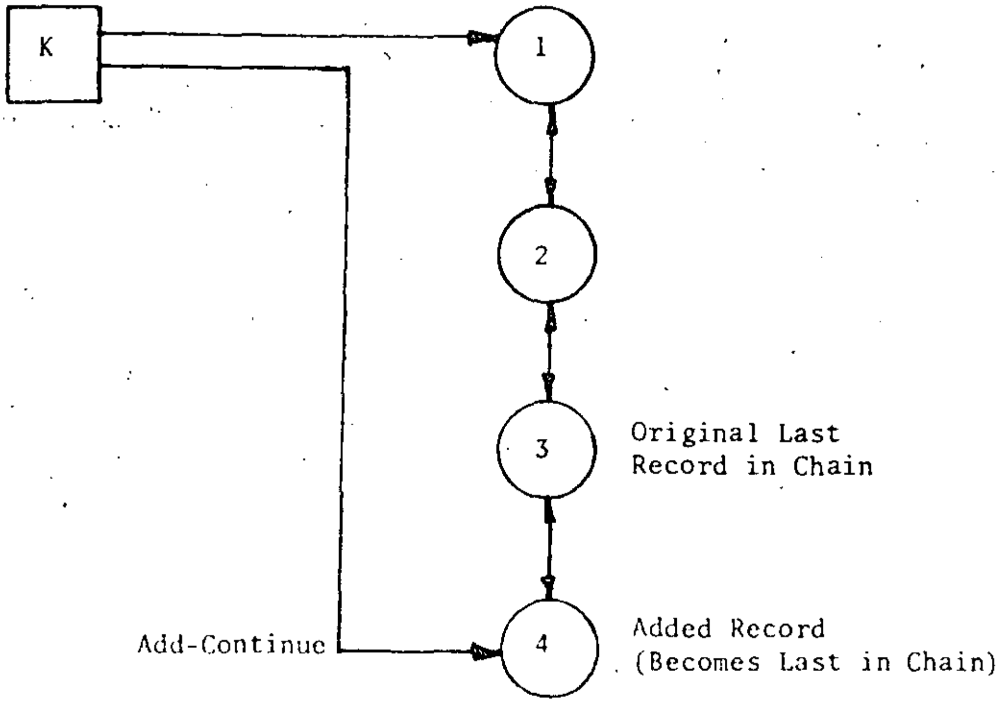


FIGURE 2-13 ADD CONTINUE

[16]

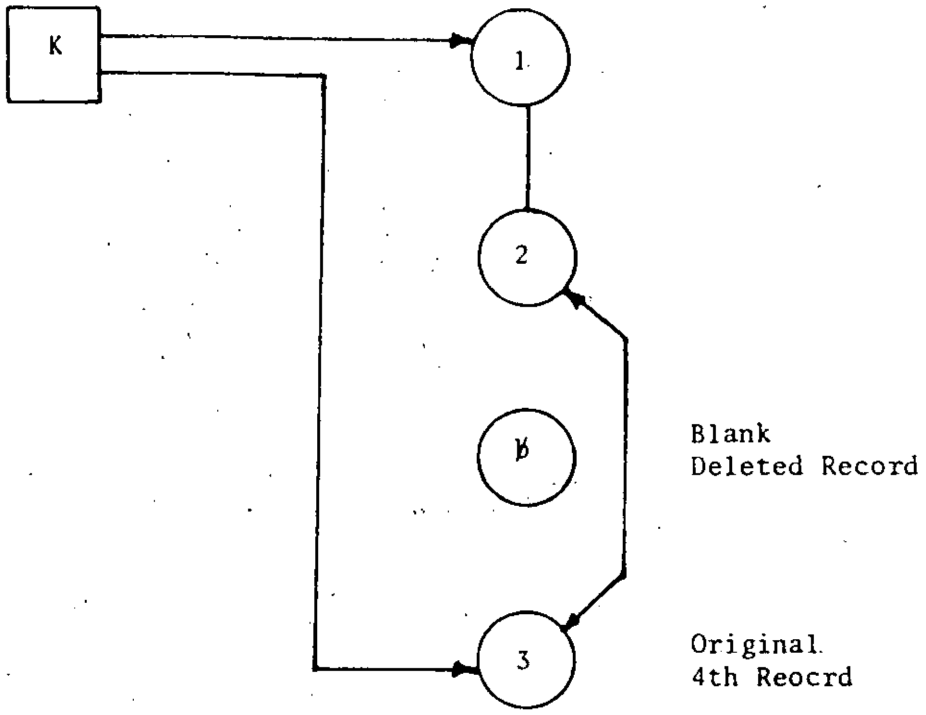


FIGURE 2-14: DELETE A VARIABLE

A delete function is used to delete the third variable record in the chain. The third record must be retrieved by the execution of the Read function before a deletion can occur. After deletion the original fourth record becomes the third. The deleted record will be blanked and is available for immediate reuse.

An Add-Before function is used to add a variable record and place it before (logically in front of) the second record in the chain. The second record must be retrieved by the execution of the Read function before the add can occur. The added record becomes the second record; the original second and third records become the new third and fourth records in the chain, respectively.

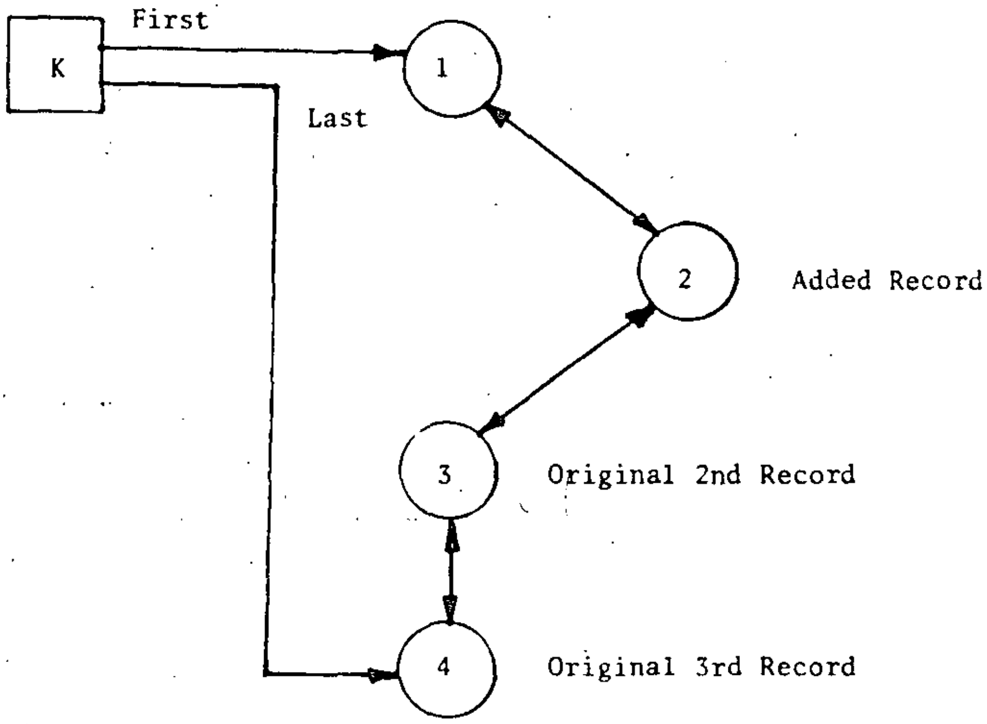


FIGURE 2-15: ADD-BEFORE

An Add-After function is used to add a variable record and place it after (logically behind) the second record in the chain. The second record must be retrieved by the execution of the Read function before the add can occur. The added record becomes the third record and the original third record becomes the fourth record in the chain.

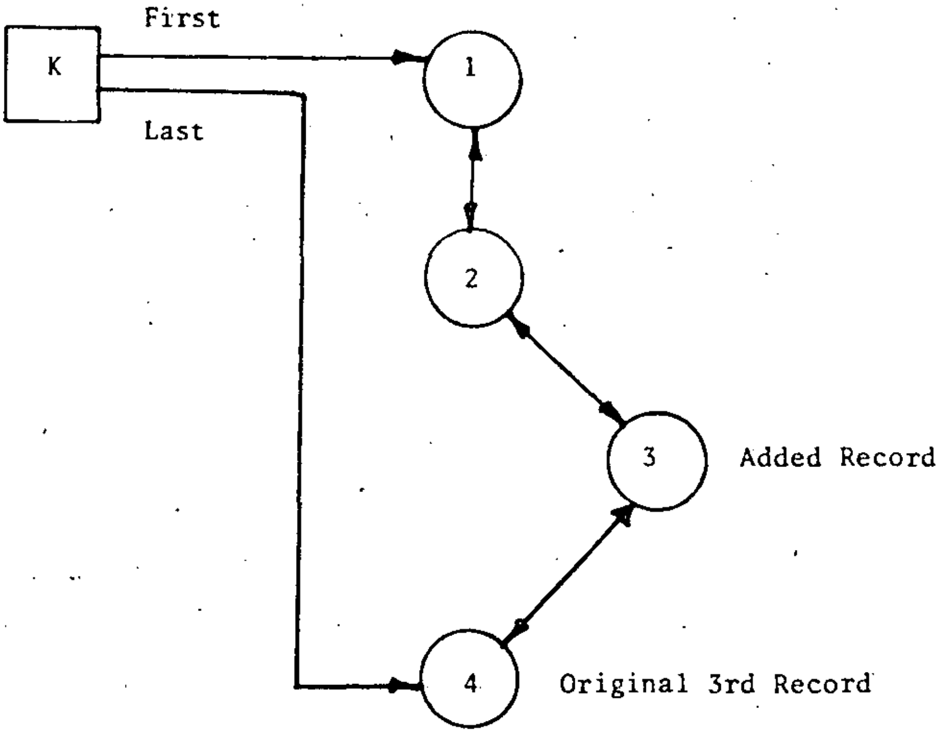


FIGURE 2-16: ADD-AFTER

The manipulation of the second category is shown from Figure 2-17 to Figure 2-19. It is assumed that a single variable data set is associated with two master data sets.

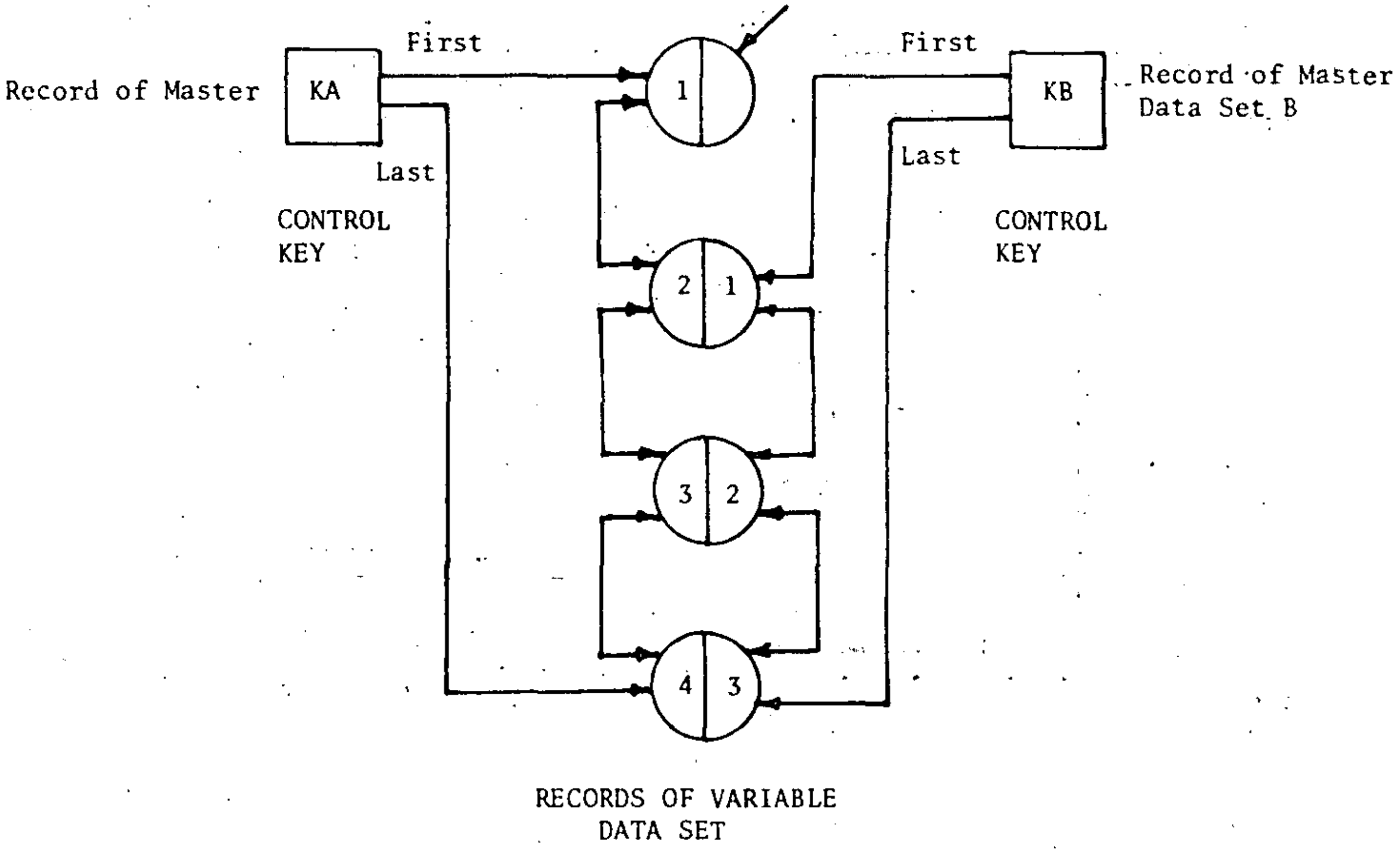
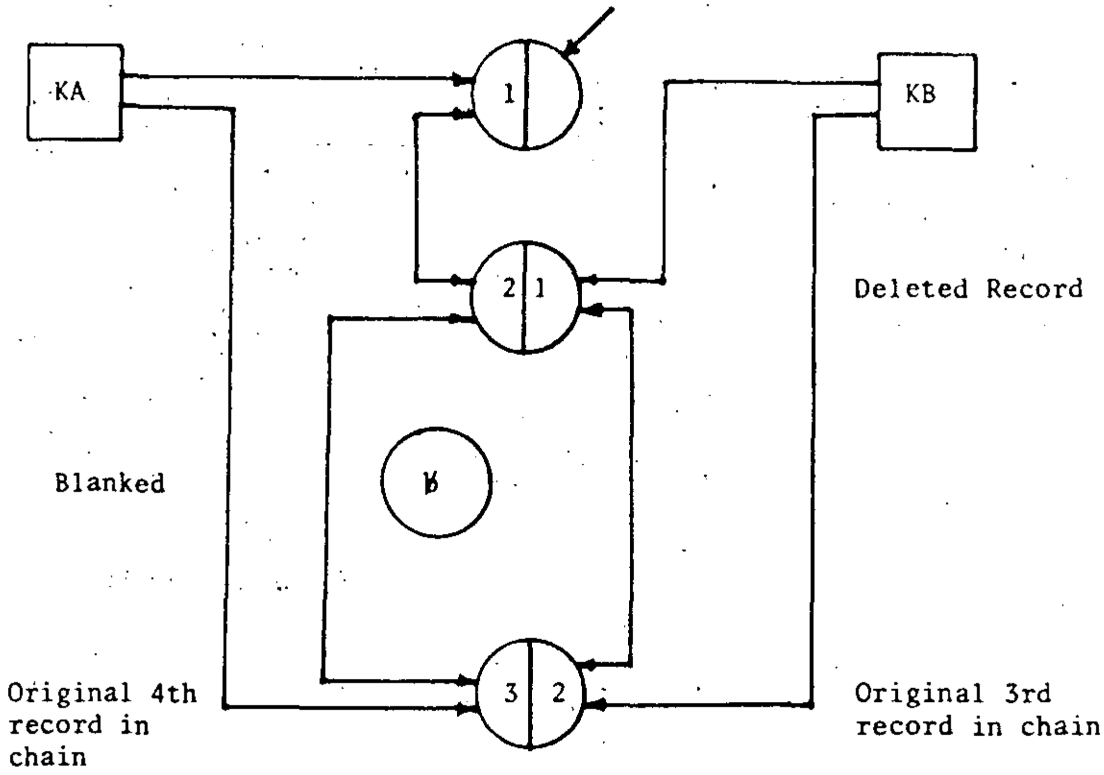


FIGURE 2-17: MULTIPLE RECORD CHAINS (TWO)

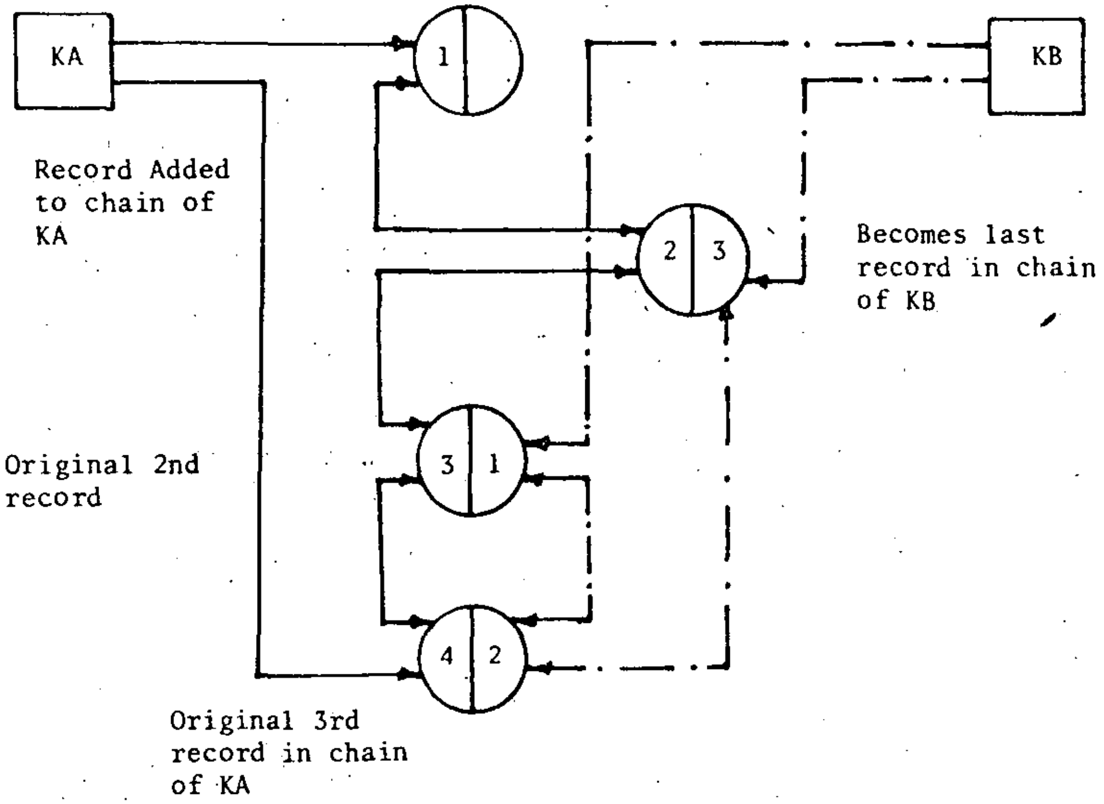
When a Delete function is used to delete the third record from the chain having the control key KA, the original fourth record in the chain becomes the third. Since the deleted record was also the second record of this chain having the control key KB, relinkage is needed in order to drop the record from that chain. Thus the original third record becomes the second record in the chain. The deleted record will be blanked for reuse.

FIGURE 2-18: DELETE RECORD (refer to figure 2-17)



An Add-Before function is used to add a variable record before the second record in the chain having the control key KA. The added record becomes the second. The original second and third record become the third and fourth records in the chain, respectively. The significance is that the added record will automatically be linked to the bottom of the record chain having the control key KB. The same concept applies to the Add-After command. As for the Add-Continue command, the added record will become respectively the last record of both chains.

FIGURE 2-19: ADD-BEFORE (refer to Figure 2-18)


KA
1
KB
Record Added to chain of KA
2
3
Becomes last record in chain of KB
Original 2nd record
3
1
4
2
Original 3rd record in chain of KA

V.3.- Ejecución de las llamadas.

Para ilustrar como se procede cuando se ejecuta una llamada a DATBAS vease la siguiente figura.

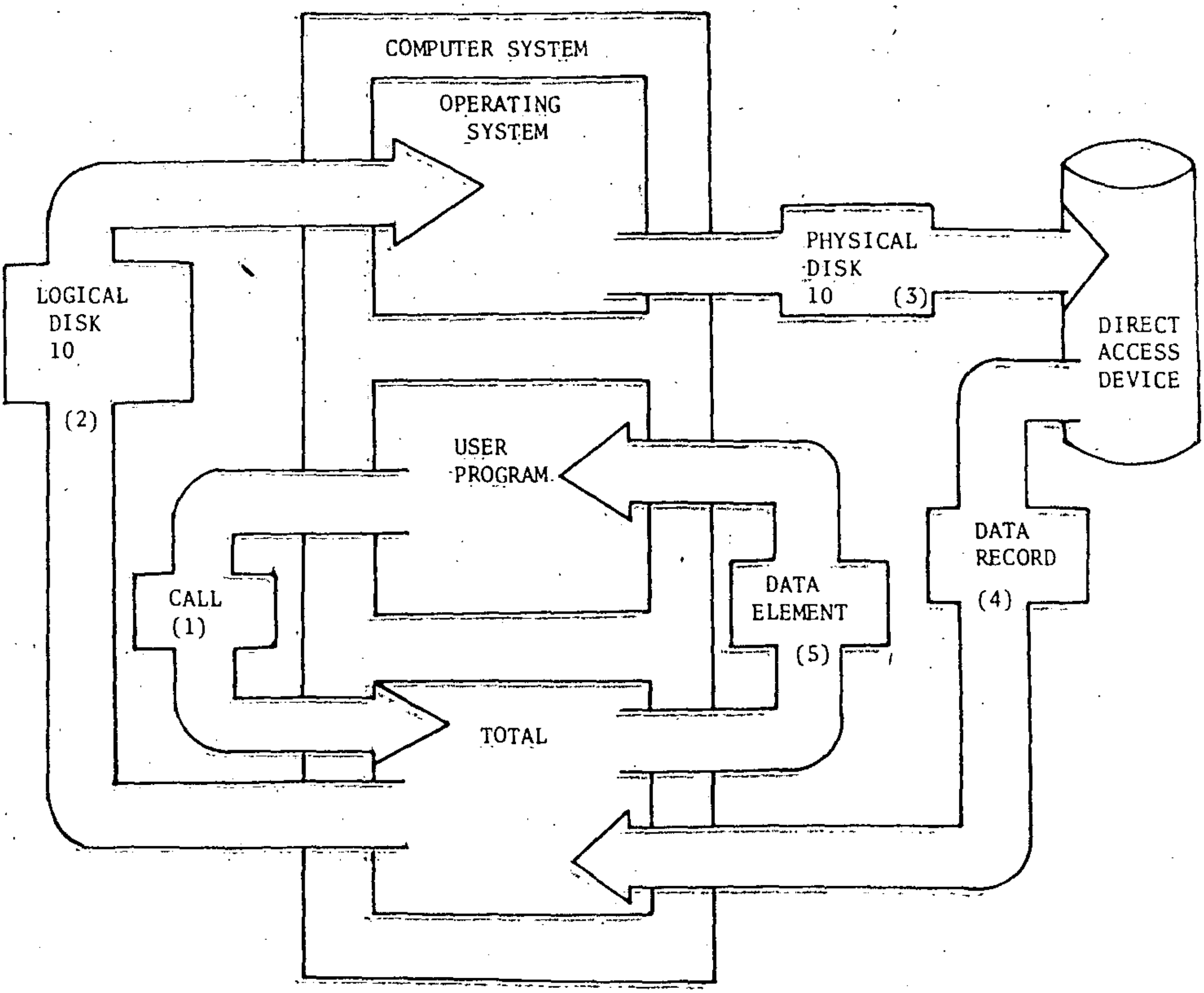


FIGURE 2-11: STRUCTURE AND EXECUTION OF DML COMMAND