# The PMS and ISP descriptive systems for computer structures*

*by* C. GORDON BELL and ALLEN NEWELL

*Carnegie-Mellon University*
Pittsburgh, Pennsylvania

## INTRODUCTION

In this paper we propose two notations for describing aspects of computer systems that currently are handled by a melange of informal notations. These two notations emerged as a by-product of our efforts to produce a book on computer structures (Bell and Newell, 1970). Since we feel it is slightly unusual to present notations per se, outside of the context of particular design or analysis efforts that use them, it is appropriate to relate some background history.

The higher *levels* of computer structure—roughly, all aspects above the level of logical design—are becoming increasingly complex and, as a result, developing into serious domains of engineering design. By serious we mean the growth of techniques of analysis and synthesis, with a body of codified technique and design experience which professional designers must assimilate. In the present state, most knowledge about the technologies for computer architecture is embedded in particular studies for particular computer systems. Nothing exists comparable to the array of textbooks and systematic treatments of logical design or circuit design.

We started off simply to produce a set of readings in computer systems, motivated by this lack of systematic treatment and the inaccessibility of good examples. As we gathered material we became impressed (depressed is actually a better term) with the diversity of ways of describing these higher levels. The amount of clumsy description—downright verbosity—even in purely technical manuals acted as a further depressant. The thought of putting such a congeries of descriptions between hard covers for one person to peruse and study was almost too much to contemplate. We began to rewrite and condense many of the descriptions. As we did so, a set of common notations developed. Becoming aware of what was happening, we devoted a substantial amount of attention and effort to creating notational systems that have some consistency and, hopefully, some chance of doing the job required. These are the PMS descriptive system for what we will call the *PMS level* of computer structure (essentially the information flow level), and the ISP descriptive system for defining the *programming level* in terms of the *logic level* (actually, the *register-transfer* level).

Thus, these two notations were developed to do a descriptive task—to be able to write down the information now given in the basic machine manual in a systematic and uniform way for all current computers. They were to provide a complete defining description for complete systems, such as the IBM 7090 or the SDS 930. Hence, the essential constraints for the notations to satisfy were ones of completeness, flexibility, and brevity (i.e., high informational density).

We think the two notations meet these requirements. They have not yet been used in a way that meets additional requirements that we would all put on notational systems; namely, that there be analysis and synthesis techniques developed in terms of them.*

---

* This paper is taken from Chapters 1 and 2, substantially compressed and rewritten, of a book, *Computer Structures, Readings and Examples* (Bell and Newell, McGraw-Hill, 1970), which is about to be published. All figures in the paper have been reproduced with the permission of McGraw-Hill. The research in this paper was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F 44620-67-C-0058) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

* There is currently a thesis in progress establishing a substantial amount of standard analysis at the PMS level. In addition, there exists at least one simulation system at the register-transfer level (Darringer, 1969) that bears a close kinship to ISP. Finally, one new computer, the DEC PDP-11, reported in this conference (Bell, et al., 1970), was designed using PMS and ISP as the working notations.

use by many people. Thus, they are undoubtedly imperfect in a number of ways (even beyond the usual questions of taste in notation, which always prevents uniform agreement and satisfaction).

By way of justification let us simply note the many places where pure descriptions (without analysis or synthesis techniques) are critical to the operation of the computer field. The programming manual serves as the ultimate educational and reference document for all programmers. Professional papers reporting on new computing systems give descriptions of the overall configuration; currently these are done by informal block diagrams. Each manufacturer adopts descriptive names of its own choosing, often for marketing purposes, to describe the components of its systems in sales brochures—e.g., selector, channel, peripheral processor, adapter, bus. During negotiations for the purchase or sale of computer system, overall descriptions (at the PMS level, as it turns out) are passed between manufacturer and potential customer. Large amounts of rough performance analyses are based on such abbreviated system descriptions. In the classroom (and elsewhere) systems are presented briefly to make particular points about design features. A user, even though he knows the order code of a machine, needs to learn the configuration available at a given installation (which, again, is a description at the PMS level). The list could be extended somewhat further, but perhaps the point is made. There is a substantial need for a uniform way of describing the upper levels of computer structures, not just for computer design, but for innumerable other purposes of marketing, use, comparison, education, etc.

With this preamble, let us describe the two notations. Notations are not theoretically neutral. That is, they are based on a particular view of the systems to be described. Thus, to understand PMS and ISP we give briefly this view of computer systems. This material is elementary and known, at least implicitly, to all computer designers. But it serves to provide the rationale for the notations and to locate them with respect to other descriptions of computer systems. After we have given some of this background, we will describe, first, PMS and then ISP. The two descriptive However, the gains to the computer field simply from the use of good descriptive notations are immense. Thus, we think that these two notations should be put forward to the computer community, both for criticism and as one concrete proposal for the adoption of a uniform notation.** The present notations are quite new and have hardly been thoroughly tested in systems have a common base of conventions, but it is simpler to treat them separately, especially when being informal. We will use the PDP-8 as an example for both PMS and ISP, since it is small enough to be completely described within the confines of this paper. At the end, in order to give some indication of generality, we will treat briefly the CDC 6600.

Our treatment here of these notations is essentially informal and heuristic. A complete treatment, as well as many examples, can be found in the forthcoming book (Bell and Newell, 1970).

## HIERARCHICAL SYSTEM LEVELS

A computer system is complex in several ways. Figure 1 shows the most important. There are at least four levels that can be used in describing a computer. These are not alternative descriptions. Each level arises from abstraction of the levels below it.

A system (at any level) is characterized by a set of components, of which certain properties are posited, and a set of ways of combining components to produce systems. When formalized appropriately, the behavior

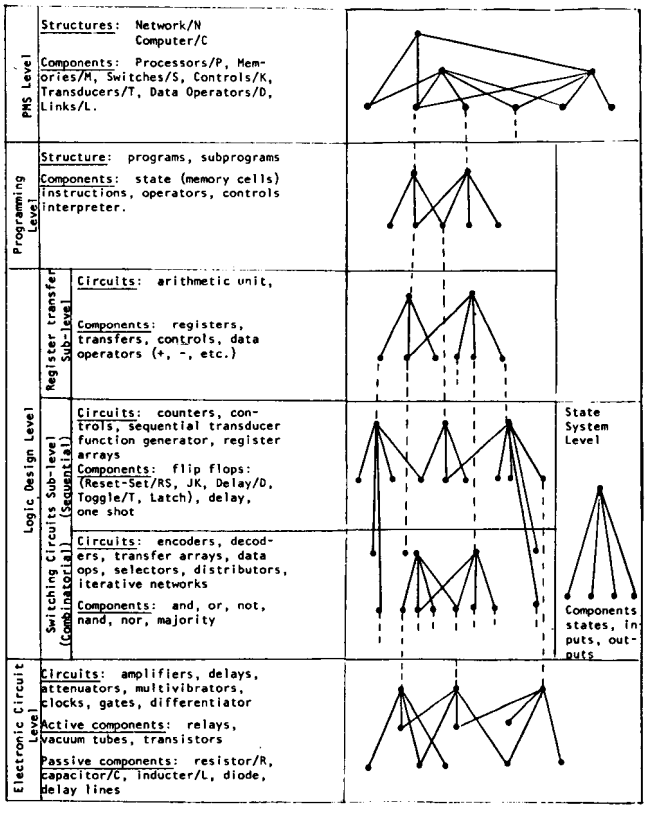



Figure 1—Hierarchy of computer structures

** A standards committee might be set up for dealing with these system levels and their description.

of the systems is determined by the behavior of its components and the specific modes of combination used. Elementary circuit theory is the prototypic example. The components are R's, L's, C's and voltage sources. The mode of combination is to run wires between the terminals of components, which corresponds to an identification of current and voltage at these terminals. The algebraic and differential equations of circuit theory provide the means whereby the behavior of a circuit can be computed from the properties of its components and the way the circuit is constructed.

There is a recursive or nested feature to most system descriptions. A system, composed of components structured in a given way, may be considered a component in the construction of yet other systems. There are primitive components whose properties are not explicable as the resultant of a system of the same type. For example, a resistor is usually not explained by a subcircuit, but is taken as a primitive. Sometimes there are no absolute primitives, it being a matter of convention what basis is taken. For example, one can build logical design systems from many different primitives (AND and NOT; NAND; OR and NOT; etc.).

A *system level*, as we have used the term in Figure 1, is characterized by a distinct language for representing and analyzing the system (that is, the components, modes of combination, and laws of behavior). These distinct languages reflect special properties of the types of components and of the way they combine. Within each level there exists a whole hierarchy of systems and subsystems. However, as long as these are all described in the same language—e.g., a subroutine hierarchy, all given in machine assembly language—they do not constitute separate system levels.

The *circuit level*, and the combinatorial switching sublevel and sequential switching sublevels of the logic level, are clearly defined in the current art. The register-transfer level is still uncertain because there is neither substantial agreement on the exact language to be used for the level, nor on the techniques of analysis and synthesis that go with it. However, there are many reasons to believe it is emerging as a distinct system level.

In the register-transfer level the system undergoes discrete operations, whereby the values of various registers are combined according to some rule, and then stored in another register (thus "transferred"). The law of combination may be almost anything, from the simple unmodified transfer ($A \leftarrow B$) to logical combination ($A \leftarrow B \wedge C$) to arithmetic ($A \leftarrow B + C$). Thus, a specification of the behavior, equivalent to the boolean equations of sequential circuits or the differential equations of the circuit level, is a set of expressions (often called productions) which give the conditions under which such transfers will be made.

There have been a number of efforts to construct formalized register transfers systems. Most of them are built around the construction of a programming system or language that permits computer simulation of systems on the RT level (e.g., Chu, 1962; Darringer, 1969). Although there is agreement on the basic components and types of operations, there is much less agreement on the representation of the laws of the system.

The *state system* representation is also at the logic level, but it has been put off to one side in Figure 1. The state system is the most general representation of discrete system available. A system is represented as capable of being in one of a set of abstract states at any instant of time. (For digital systems the set is finite or enumerable.) Its behavior is specified by a transition function that takes as arguments the current state and the current input and determines the next state (and the concomitant output). A digital computer is, in principle, representable as a state system, but the number of states is far too large to make it useful to do so. Instead, the state system becomes a useful representation in dealing with various subparts of the total machine, such as the sequential circuit that controls a magnetic tape. Here the number of states is small enough to be tractable. Thus, we have placed state systems off to one side as an auxiliary to the logic level.

The *program level* is not only a unique level of description for digital technology (as was the logic level), but it is uniquely associated with computers, namely, with those digital devices that have a central component that interprets a programming language. There are many uses of digital technology, especially in instrumentation and digital controls which do not require such an interpretation device and hence have a logic level but no program level.

The components of the program level are a set of memories and a set of operations. The memories hold data structures which represent things both inside and outside of the memory, e.g., numbers, payrolls, molecules, other data structures, etc. The operations take various data structures as inputs and produce new data structures, which again reside in memories. Thus the behavior of the system is the time pattern of data structures held in its memories. The unique feature of the program level is the representation it provides for combining components—that is, for specifying what operations are to be executed on what data structures. This is the program, which consists of

a sequence of instructions. Each instruction specifies that a given operation (or operations) be executed on specified data structures. Superimposed on this is a control structure that specifies which instruction is to be interpreted next. Normally this is done in the order in which the instructions are given, with jumps out of sequence specified by branch instructions.

In Figure 1 the top level is called the Processor-Memory-Switch level, or PMS level for short. The name is not recognized, nor is any other, since the level exists only informally. Nevertheless, its existence is hardly in doubt. It is the view one takes of a computer system when one considers only its most aggregate behavior. It then consists of central processors, core memories, tapes, discs, input/output processors, communication lines, printers, tape controllers, busses, Teletypes, scopes, etc. The system is viewed as processing a medium, information, which can be measured in bits (or digits, characters, words, etc.). Thus the components have capacities and flow rates as their operating characteristics. All details of the program are suppressed, although many gross distinctions of encoding and information type remain, depending on the analysis. Thus, one may distinguish program from data, or file space from resident monitor. One may remain concerned with the fact that input data is in alphameric and must be converted into binary, or is in bit serial and must be converted to bit parallel.

We might characterize this level as the "chemical engineering view of a digital computer," which likens it more to a continuous process petroleum distilling plant than to a place where complex FORTRAN programs are applied to matrices of data. Indeed, this system level is more nearly an abstraction from the logic level than from the program level, since it returns to a simultaneously operating flow system.

One might question whether there was a distinct systems level here. In the early days of computers almost all computer systems could be represented as in the diagram in MIT's Whirlwind Computer programming manual in Figure 2: the four classic boxes of memory (storage), control, arithmetic, and input/output (separated, in the figure). But current time-sharing and multiprocessing systems are orders of magnitude more complex than this, and it is known that the structure at this level has a determining influence on system performance. (See the PMS diagram for the 6600 in Figure 6, by no means the most complex of current systems.)

With this total view of the various systems levels we can locate both PMS and ISP. PMS is, of course, a systems level of its own, namely, the top one. ISP is a notation for describing the components and modes of combination of the programming level in terms of the

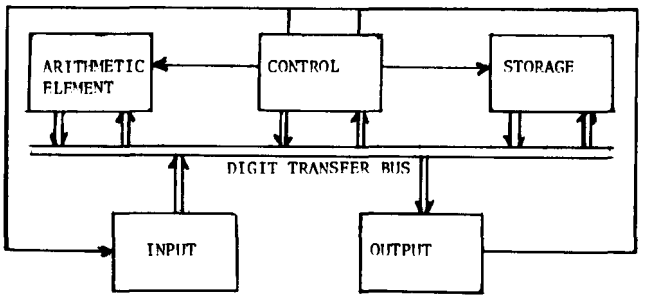



Figure 2—Simplified computer block diagram Whirlwind I (courtesy of M.I.T.)

next level down, i.e., in terms of the register transfer level. That is, the instructions, operations and interpretation cycle are the defining components of the programming level and must be given in terms of a more basic systems level. The programming level itself consists of programs written in the machine code of the system. In essence, a register-transfer description of a processor is an interpreter program for interpreting the instruction set. The interpreter describes the actual hardware of the processor. By carefully structuring a register-transfer description of a processor, instructions are precisely defined.

Thus, ISP is an interface language. Similarly, interface definitions exist at all levels of a system hierarchy, e.g., between the circuit level and the logic level. Normally, it is not necessary to have a special language for the interface; e.g., one simply writes a circuit description of an AND-gate. But with the programming level, it is most useful not to use simply a register transfer language, but to introduce a special notation (i.e., ISP). This will become clear when we describe ISP.

PMS and ISP are also strongly related in that ISP statements express the behavior of PMS components. Thus, for every PMS component there are constructs in ISP that express its behavior; and each ISP statement implies particular PMS structures.

A word should be said about antecedents. The PMS descriptive system is close to the way we all talk informally about the top level of computer systems; no one effort in the environment stands out as a predecessor. Some notations, such as CPU (for central processing units), have become widespread. We clearly have assimilated these. Our modifications, such as Pc instead of CPU, are dictated entirely by the attempt to build a consistent notation over the whole range of computer systems. With respect to ISP, we have been heavily influenced by the work on register transfer languages.* The one that we used most as a kernel

from which to grow ISP was the work of Darringer and Parnas (Darringer, 1968). In particular, their decision to work within the framework of ALGOL suited our own sensibilities, even though the final version of ISP departs from a sequential algorithmic language in a number of respects.

## PMS LEVEL OF DESCRIPTION

Digital systems are normally characterized as systems that at any time exist in one of a discrete set of states, and which undergo discrete changes of state with time. Nothing is said about what physical state corresponds to a system state; or the behavior of components that transform the system from one state to another. The states are given abstract labels: $S_1$, $S_2$, ... . The transitions are provided by a state-transition table (or state diagram) of the form: if the system is in state $S_i$ and the input is $I_j$, then the system is transformed to $S_k$ and evokes output $O_l$. The "state-system" view captures what is meant by a discrete (or digital) system. Its disadvantage is its comprehensiveness, which makes it impossible to deal with large systems because of their immense number of states (of the order $10^{10^7}$ states for a big computer).

Existing digital computers can be viewed as discrete state systems that are specialized in three ways. First, the state is realized by a medium, called information, which is stored in memories. Thus, a processor has all its states made explicit in a set of registers: an accumulator, an address register, an instruction register, status register, etc. No permanent information is kept in digital devices except as encoded in bits or some other information unit base in a memory. Sequential logic circuits that carry out operations in the system may have intermediate non-staticized states (e.g., during a multiply instruction), but these are only temporary. Second, the current digital computer systems consist of a small number of discrete subsystems linked together by flows of information. The natural representation of a digital computer system is as a graph which has component systems at the nodes and information flows as branches. This representation as an information flow network with functionally specialized nodes is a real specialization. Finally, each component in a digital system has associated with it a small number of discrete operations for changing its own state or the state of neighboring components. The total behavior of the system is built up from the repeated execution of the operations as the conditions for their execution become realized by the results of prior operations.

To summarize, we want a way of describing a system of an interconnected set of *components*, which are individual devices that have associated with them a set of *operations* that work on a medium of *information*, measured in bits (or some other base). For the PMS level we ignore all the fine structure of information processing and consider a system consisting of components that work on a homogeneous medium called information. Information comes in packets, called *i-units* (for information units) and is measured in bits (or equivalent units, such as characters). I-units have the sort of hierarchical structure indicated by the phrase: a record consists of 300 words; a word consists of 4 bytes; a byte consists of 8 bits. A record, then, contains $300 \times 4 \times 8 = 9600$ bits. Each of these numbers—300, 4, 8—is called a *length*.

Other than being decomposable into a hierarchy of factors, i-units have no other structure at the PMS level. They do have a *referent*—that is, a *meaning*. At the PMS level we are not concerned with what is referred to, but only with the fact the certain components transform i-units, but do not modify their meaning. These meaning-preserving operations are the most basic information processing operations of all—and provide the basic classification of computer components.

### *PMS primitives*

There are seven basic component types, each distinguished by the kinds of operations it performs:

*Memory, M.* A component that holds or stores information (i.e., i-units) over time. Its operations are reading i-units out of the memory, and writing i-units into the memory. Each memory that holds more than a single i-unit has associated with it an *addressing system* by means of which particular i-units can be designated or selected. A memory can also be considered as a switch to a number of submemories. The i-units are not changed in any way by being stored in a memory.

*Link, L.* A component that transfers information (i.e., i-units) from one place to another in a computer system. It has fixed terminals. The operation is that of transmitting an i-unit (or a sequence of them) from the component at one terminal to the component at the other. Again, except for the change in spatial position, there is no change of any sort in the i-units.

* We have not been influenced in a direct way by the work of Iverson (Falkoff, Iverson and Sussenguth, 1964) in the sense of patterning our notation after his. Nevertheless, his creation of a full description of the IBM System/360 system in APL stands as an important milestone in moving toward formal descriptions of machines.

*Control, K.* A component that evokes the operations of other components in the system. All other components are taken to consist of a set of discrete operations, each of which—when evoked—accomplishes some discrete transformation of state. With the exception of a processor, P, all other components are essentially passive and require some other active agent (a K) to set them into small episodes of activity.

*Switch, S.* A component that constructs a link between other components. Each switch has associated with it a set of possible links, and its operations consist of setting some of these links and breaking others.

*Transducer, T.* A component that changes the i-unit used to encode a given meaning (i.e., a given referent). The change may involve the medium used to encode the basic bits (e.g., voltage levels to magnetic flux, or voltage levels to holes in a paper card) or it may involve the structure of the i-unit (e.g., bit-serial to bit-parallel). Note that T's are meaning preserving, but not necessarily information preserving (in number of bits), since the encodings of the (invariant) meaning need not be equally optimal.

*Data-operation, D.* A component that produces i-units with new meanings. It is this component that accomplishes all the data operations, e.g., arithmetic, logic, shifting, etc.

*Processor, P.* A component that is capable of interpreting a program in order to execute a sequence of operations. It consists of a set of operations of the types already mentioned—M, L, K, S, T and D—plus the control necessary to obtain instructions from a memory and interpret them as operations to be carried out.

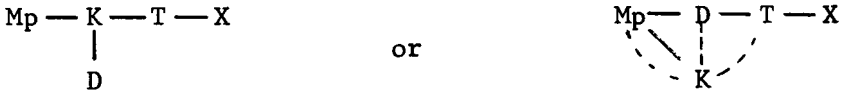
*Computer model (in PMS)*

Components of the seven types can be connected to make *stored program digital computers*, abbreviated by C. For instance, the classical configuration for a computer is:

$$C := Mp - Pc - T - X$$

Here Pc indicates a *central processor* and Mp a *primary memory*, namely, one which is directly accessible from a P and holds the program for it. T (input/output device) is a transducer connected to the external environment, represented by X. (The colon-equals (:=) indicates that C is the name of what follows to the right.)

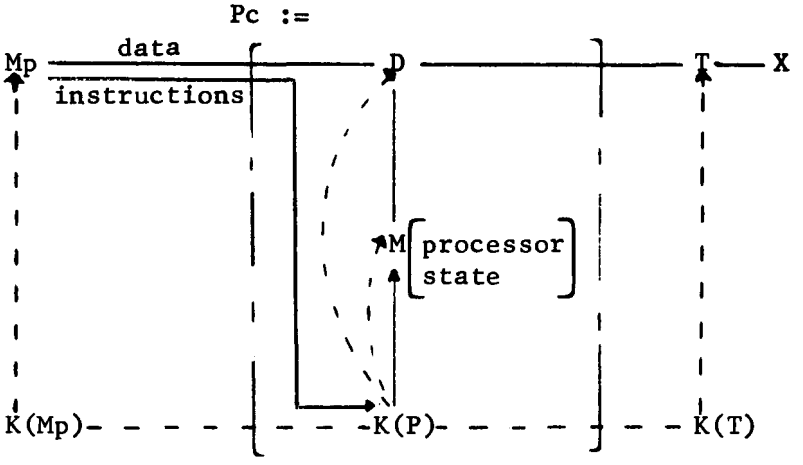
The classic diagrams had four components, since it decomposed the Pc into a control and an arithmetic unit:




where the heavy information carrying lines are for instructions and their data, and the dotted lines signify control.

Often logic operations were lumped with control, instead of with data operations—but this no longer seems to be the appropriate way to functionally decompose the system.

Now we associate local control of each component with the appropriate component to get:



where the heavy lines carry the information in which we are interested, and the dotted lines carry information about when to evoke operations on the respective components. The heavy information carrying lines between K and Mp are instructions. Now, suppressing the K's, then lumping the processor state memory, the data operators, and the control of the data, operators and processor state memory to form a central processor, we again get:

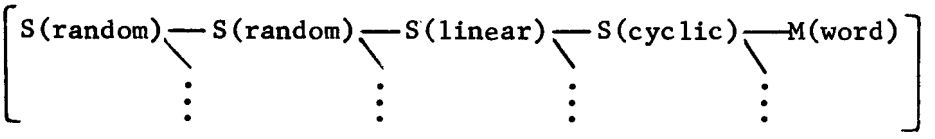
Mp—Pc—T—X

Computer systems can be described in PMS at varying levels of detail. For instance, we did not write in the links (L's) as separate components. These would be of interest only if the delays in transmission were significant to the discussion at hand, or if the i-units transmitted by the L were different from those available at its terminals. Similarly, often the encoding of information into i-units is unimportant; then there is no reason to show the T's. The same statement holds for K's—sometimes one wants to show the locus of control, say when there is one control for many components, as in a tape controller; but often this is not of interest. Then, there is no reason to show K's in a PMS diagram.

As a somewhat different case, it turns out that D's never occur in PMS diagrams of computers, since in the present design technology D's occur only as subcomponents of P's. If we were to make PMS-type diagrams of analog computers, D's would show extensively as multipliers, summers, integrators, etc. There would be few memories and variable switches. The rather large patchboard would be represented as a very elaborate manually fixed switch.

Components are themselves decomposable into other components. Thus, most memories are composed of a switch—the addressing switch—and a number of submemories. Thus a memory is recursively defined as either a memory or a switch to other memories. The decomposition stops with the unit-memory, which is one that stores only a single i-unit, hence requires no addressing. Likewise, a switch is often composed of a cascade of 1-way to n-way switches. For example, the switch that addresses a word on a multiple-headed disk might look like:

M.disk :=




The first S(random) selects a specific Ms.disk␣drive␣ unit; the second S(random) is a switch with random addressing that selects the head (the platter and side); S(linear) is a switch with linear accessing that selects the track; and S(cyclic) is a switch with cyclic address-that finally selects the M(word) along the circular recurring track. Note that the switches are realized by differing technologies. The first two S(random)'s are generally electronic (AND-OR gates) with selection times of 10 $\sim$ 100 microseconds, or perhaps electromechanical (relay). The S(linear) is the electromechanical action of a stepping motor or a pneumatic driven arm which holds the read-write heads—the selection time for a new track is 50 $\sim$ 500 milliseconds. Finally, the S(cyclic) is determined by the rotation time of the disk and requires from 16 $\sim$ 60 milliseconds, depending on the speed (3600 $\sim$ 1000 revolutions/minute). This decomposition capability allows us to be able to describe components with varying precision and accuracy.

The control element of a computer is often shown as being associated with the processor—not to the control of a disk or magnetic tape, such a K is often more complex. When we suppress detail, controls often disappear from PMS diagrams. Alternatively, when we agglomerate primitive components (as we did above when combining Mp and K(Mp) to be just Mp) into the physically distinct sub-parts of a computer system, a separate control, K, often appears. The functionally and physically separate control* has evolved in the last decade. These controls, often larger than a Pc, are sometimes computers with stored control programs. When we decompose such a control there are: data operations (D) for calculating addresses or for error detection and error correction data; transducers (T) for changing logic signal levels and information flow widths; memory (M) as it is used in D, T, K, and for buffering; and finally a large control (K) which coordinates the activities of all the other primitives.

* A variety of names for *K*'s are used, e.g., controller, adapter, selector, interface, buffer multiplexor, etc. Often these names reflect other functions performed by the device.

The components are named according to the function they perform and they can be composed of many different types of components. Thus, a control (K) must have memory (M) as a subcomponent, and a memory, M, may have a transducer (T) as well as a switch (S) as subcomponents. All of these subcomponents, of course, exist to accomplish the total function of the component, and do not make the component also some other type. For instance, the M that does a transduction (T) from voltages on its input wires to magnetism in its cores and a second transduction from magnetism to voltages on its output wires does not thereby become a transducer as far as the total system functioning is concerned. To the rest of the system all the M can do is to remember i-units, accepting and delivering them in the same form (voltages). We define for each component type both a simple component and a compound component, reflecting in part the fact that complex subsystems can be put together to perform a single function from the viewpoint of the total system. For example, a typewriter may have 4 $\sim$ 6 simple information transduction channels using video, tactile, auditory, and paper information carriers.

*PMS notation*

Various notational conventions designate specifications for a component, e.g., Mp for a functional classification, and S(cyclic) for a type of switch access function in the case of rotating memory devices like drums. There are many other additional specifications one wants to give. A single general way of providing additional specifications is used so that if X is a component, we can write:

$$X(a_1: v_1; a_2: v_2; \ldots.)$$

to indicate that X is further specified by attribute $a_1$ having value $v_1$, attribute $a_2$ having value $v_2$, etc. Each *parameter* (as we call the pair $a_i$ $v_i$ is well defined independently of what other parameters are given; hence, there is no significance to the order in which they are written, or to the number which have to be written.

According to this notation we should have written M(function:primary) or S(access-function:random) rather than Mp or S(random). There are conventions for abbreviating and abstracting parameters to avoid such a lengthy description. Alternative ways of writing Mp are:

| | |
|---|---|
| M(function:primary) | complete specification |
| M(primary) | drop the attribute, function, since it can be inferred from the value |
| M.primary | use the value outside the parenthesis, concatenated with a dot |
| M.p | use an explicitly given abbreviation, namely, primary/p (only if it is not ambiguous) |
| Mp | drop the concatenation marker (the dot), if it is not needed to recover the two parts (all components are given by a single capital letter—here M) |

Each of these rules corresponds to a natural tendency to abbreviate when redundant information is given; each has as its condition that recovery must be possible.

In the full description (Bell and Newell, 1970) each component is defined and given a large number of parameters, i.e., attributes with their domain of values. Throughout, the slash (/) is used to introduce abbreviations and aliases as we go.* Any list of parameters does not exhaust those aspects of a component that one might ever conceivably want to talk about. For instance, there are many quite distinct dimensions for any component in addition to the information dimension: packaging, physical size, physical location, energy use, cost, weight, style and color, reliability, maintainability, etc. Furthermore, each of these dimensions includes an entire set of parameters, just as the information dimension breaks out into the set of parameters illustrated in the figures. Thus the descriptive system is an open one and new parameters are definable at any occasion.

The very large number of parameters provides one of the major challenges to creating a viable scheme to describe computer systems. We have responded to this in part by providing automatic ways in which one can compress the descriptions by appropriate abbreviation —while still avoiding a highly cryptic encoding of each separate aspect. Abstraction is another major area in which some conventions can help to handle the large numbers of parameters. For instance, one attribute of a processor is the time taken by its operations. This attribute can be defined with a complex value:

Pc(operation-times: add:4 $\mu$s, store:4 $\mu$s, load:4 $\mu$s, multiply:16 $\mu$s, ...)

That is, the value is a list of times for each separate operation. One might also give only the range of these numbers; this is done by indicating that the value is a range:

Pc(operation-time: 4 $\sim$ 16 $\mu$s).

* There is no difficulty distinguishing this use from the use of slash as division sign—the latter takes priority, since it is the more specific use of the slash.

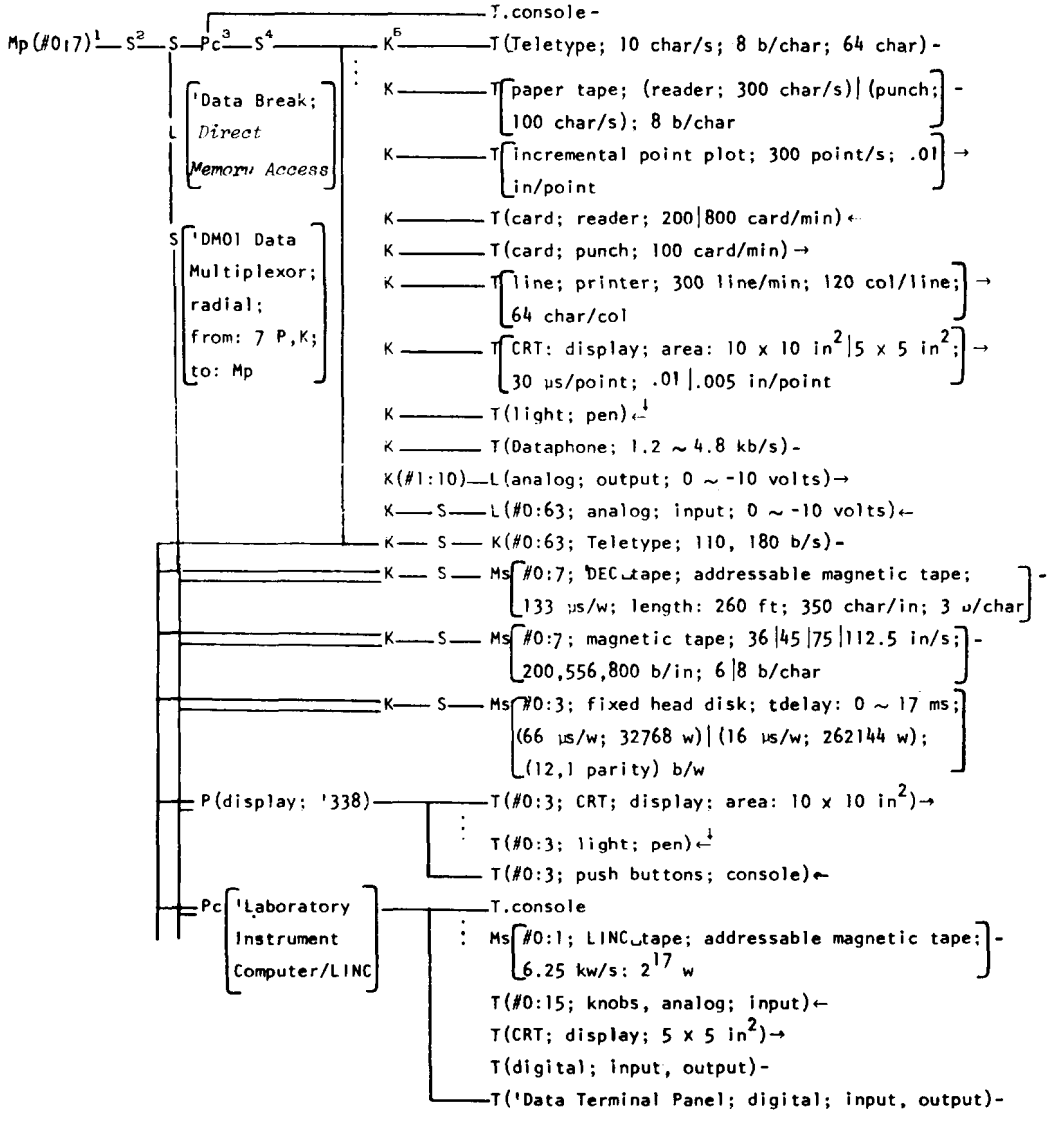
Similarly, one could have given typical and average times (under some assumed frequency mix of instructions):

Pc(operation-times: 4 $\mu$s)
Pc(operation-times: average: 8.1 $\mu$s).

The advantage of this convention, which permits descriptions of values to be used in place of actual values whenever desired, is that it keeps the number of attributes that have to be defined small.

*A PMS example using the DEC PDP-8*

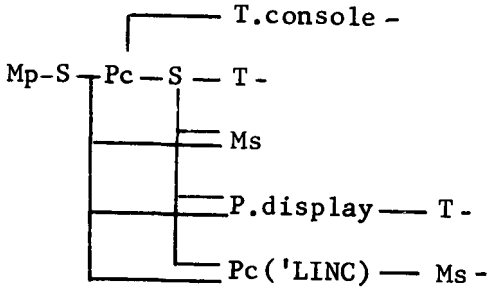
Figure 3 gives the detailed PMS diagram of an actual, small, general purpose computer, the DEC




[1]Mp(core; 1.5 μs/w; 4096 w; (12 + 1)b)
[2]S('Memory Bus)
[3]Pc(1 ~ 2 w/instruction: data: w, i,bv; 12 b/w: M.processor state($2\frac{1}{6}$ ~ $3\frac{1}{2}$) w; technology: transistors; antecedents: PDP-5; descendants; PDP-8S, PDP-8I, PDP-L)
[4]S('I/O Bus; from; Pc; to; 64 K)
[5]K(1 ~ 4 instructions; M.buffer(1 char ~ 2 w))

Figure 3—DEC LINC-8-338 PMS diagram

LINC-8-338, which is a PDP-8 with a LINC processor and a type 338 display processor. We will concentrate on the notation, rather than discussing substantive features of the system. A simplified PMS diagram of the system shows its essential structure:




This shows the basic Mp-Pc-T structure of a C with the addition of secondary memory (Ms) and two processors, one of which, Pc('LINC), has its own Ms. Two switches are used: the I/O-bus which permits access to all the devices, and a direct access path to Mp via Pc for high data rate devices. There are many other switches in the actual system as one can see from Figure 3; for example, Mp is really 1 to 8 separate modules connected by a switch S to Pc. Also there are many T's connected to the input-output switch, S, which we collapsed as a single compound T; and similarly for S(direct memory access).

Consider the Mp module. The specifications assert that it is made with core technology, that its word size is 13 bits (12 data bits plus one other with a different function); that its size is 4096 words; and that its operation time is 1.5 $\mu$s. We could have written the same information as:

M(function:primary; technology:core; operation-time: 1.5 $\mu$s; size: 4096 w; word: (12 + 1) b)

In Figure 3 we wrote only the values, suppressing the attributes, since moderate familiarity with memories permits an immediate inference about what attributes are involved. As another example, we did not specify the function of the additional bit in the word when we wrote (12 + 1) b. Informed readers will assume this to be a parity bit, since this is the common reason for having an extra bit in a word. If the extra bit had some unusual function, then we would have needed to define it. That is, in the absence of additional information, the most common interpretation is to be assumed.

In fact, we could have been even more cryptic and still communicated with most readers:

M.core (1.5 $\mu$s/w; 4 kw; 12 b),

corresponding to the phrase, "A 12 bit, 1.5 $\mu$s, 4k core store". 4 kw stands for 4 $\times$ 1024 = 4096; however, if someone who was less familiar took it to be 4 $\times$ 1000 = 4000 no real harm would be done.

Consider the magnetic tapes for Pc. Since there are eight possible tapes that make use of the same controller, K, through a switch, S, we label them #0 through #7. Actually, # is an abbreviation for the index attribute whose values are integers. Since the attribute is a unique character, we do not have to write #:3 (although we could). The additional parameters give information about the physical attributes of the encoding. These are alternative values and any tape has only one of them. A vertical bar (|) indicates this (as in BNF notation for grammars). Thus, 75|112 in/s says that one can have a tape with a speed of 75 inches per second or one with 112 inches per second, but not a tape which can be switched dynamically to run at either speed.

For many of the components no further information is given. Thus, knowing that M.magnetic␣tape is connected to a control and from there to the Pc tells generally what that K does. It is a "tape controller" which evokes all the actions of the tape, such as read, write, rewind; and therefore these actions do not have to be done by Pc. The fact that there is only one K for many Ms's implies that only one tape can be accessed at a time. Other information could be given, although that just provided is all that is usual in specifying a controller in an overall description of a system.

We have used several different ways of saying the same thing in Figure 3 in order to show the range of descriptive notations. Thus, the 64 Teletypes are shown by describing a single connection through a switch and putting the number of links in the switch above the connecting line.

Consider, finally, the Pc in Figure 3. We have given a few parameters: the number of data types, the number of instructions, and the number of interrupts. These few parameters hardly define a processor. Several other important parameters are easily inferred from the Mp. The basic operation time in a processor is a small multiple of the read time of its Mp. Thus it is predictable that Pc stores and reads information in 2 $\times$ 1.5 $\mu$s (one for instruction fetch, one for data fetch). Again, where this is not the case (as in the CDC 6600) it is necessary to say so. Similarly, the word size in the Pc is the same as the word size of the Mp—12 data bits. More generally, the Pc must have instructions that take care of evoking all the components of the PMS structure. These instructions of course do not use the switches and controls as distinct entities; rather, they speak directly to the operation of the M's and T's connected via these switches and controls.

Other summary parameters could have been given for the Pc. None would come close to specifying its behavior uniquely, although to those knowledgeable in computers still more can be inferred from the parameters given. For instance, knowing both the data types available in a Pc and the number of instructions, one can come very close to predicting exactly what the instructions are. Nevertheless, the way to describe a Pc in full detail is not to add larger and larger numbers of summary parameters. It is more direct and more revealing to develop a description at the level of instructions, which is the ISP description.

In summary, a descriptive scheme for systems as complex and detailed as digital computers must have the ability to range from extremely complete to highly simplified descriptions. It must permit highly compressed descriptions as well as extensive ones and must permit the selective suppression or amplification of whatever aspects of the computer system are of interest to the user. PMS attempts to fulfill these criteria by providing simple conventions for detailed description with additional conventions that permit abbreviation and abstractions, almost without limit. The result is a notation that may seem somewhat fluid, especially on first contact in such a brief introduction as this. But once assimilated, PMS seems to allow some of the flexibility of natural language within enough notational controls to enhance communication considerably.

## ISP LEVEL OF DESCRIPTION

The behavior of a processor is determined by the nature and sequence of its operations. This sequence is determined by a set of bits in Mp, called the program, and a set of interpretation rules, realized in the processor, that specify how particular bit configurations evoke the operations. Thus, if we specify the nature of the operations and the rules of interpretation, the actual behavior of the processor depends solely on the particular program in Mp (and also on the initial state of data). This is the level at which the programmer wants the processor described—and which the programming manual provides—since he himself wishes to determine the program. Thus the ISP (Instruction Set Processor) description must provide a scheme for specifying any set of operations and any rules of interpretation.

Actually, the ISP descriptive scheme need only be general enough to cover some broad range of possibilities adequate for past and current generations of machines along with their likely descendants. As with the PMS level, there are certain restrictions that can be placed on the nature of a computer system, specializing it from the more general concept of a discrete state system. For the PMS level, it processes a medium, called information; it is a system of discrete components linked together by information transfers; and each component is characterized by a small set of operations. Similarly, for the ISP level we can add two more such restrictions, which will in turn provide the shape of its descriptive scheme.

The first specialization is that a program can be conceived as a distinct set of *instructions*. Operationally, this means that some set of bits is read from the program in Mp to a memory within P, called the instruction register, M.instruction/M.i. This set of bits then determines the immediately following sequence of operations. Only a single operation may be determined, as in setting a bit in the internal state of the P; or a substantial number of operations may be determined, as in a "repeat" instruction that evokes a search through Mp. In a typical one or two address machine the number of operations per instruction ranges from 2 to 5. In any event, after this sequence of operations has occurred, the next instruction to be fetched from Mp is determined and obtained. Then, the entire cycle repeats itself.

The above cycle of activity is just the *interpretation cycle*, and the part of the P that performs it is the *interpreter*. The effect of each instruction can be expressed entirely in terms of the information held in memories at the end of the cycle (plus any changes made to the outside world). During execution, operations may have internal states of their own as sequential circuits which are not represented as bits in memories. But by the end of the interpretation cycle, whatever effect is to be carried on to a later time has been staticized in bits in some memory.*

The second additional specialization is on the data operations. A processor's total set of operations can be divided into two parts. One part contains those necessary to operate other components given in the PMS diagram—links, switches, memories, transducers, etc. The operations associated with these components and the extent to which they can be indirectly controlled from P are highly constrained by the basic nature of the

* This description holds true for a *P* with a single active control (the interpreter). Some *P*'s (e.g., the CDC 6600) have several active controls and get involved in "overlapping" several instructions and in reordering operations according to the data and devices available. With these, a more complex statement is required to express the same general restriction we have been stating for simple *P*'s: that the program can be decomposed into a sequence of bit sets (the instructions), each of which has local control over the behavior of the *P* for a limited period of time, with all inter-instruction effects being staticized as bits in *M*'s.

components and their controls. The second part contains those operators associated with a processor's D component. So far we have said nothing at all about them, except to exclude them completely from all PMS components except P. These are the operations that produce bit patterns with new meaning—that do all the "real" processing—or changing of information.* If it weren't for data operators, the system would only just transmit information. As we noted in our original definitions, a P (including a D) is the only component capable of directly changing information. A P can create, modify, and destroy information in a single operation. As we noted earlier, D's are like the primitive components in an analog computer. Later, when we express instruction sets as simple arithmetic expressions, the D's are the primitive operators, e.g., $+$, $-$, $\times$, $/$, $\times 2^n$, $\wedge$, $\vee$, $\oplus$, and concatenation ($\square$), which are evoked by the instruction set interpreter part of a processor.

The specialization is that all the data operations can be characterized as working on various *data-types*. For example, there is a data-type called the signed-integer, and there are data operations that add two signed-integers, subtract them, multiply them, take their absolute value, test for which of two is the greater, etc. A data-type is a compound of two things: the referent of the bit pattern (e.g., that this set of bits refers to an integer in a certain range); and the representation in the bit pattern (e.g., that bit 31 is the sign, and bits 30 to 0 are the coefficients of successive powers of 2 in the binary representation of the integer). Thus, a processor may have several data-types for representing numbers: unsigned-integers, signed-integers, single-precision-floating-point, double-precision-floating-point, etc. Each of these requires distinct operations to process it. On occasion, operations for several data-types may all be encoded into a single instruction from the programmer's viewpoint, as when there is an add instruction with a data-type sub-field that selects whether the data is fixed or floating point. The operations are still separate, no matter how packaged, and so their data-types remain distinct.

With these two additional specializations—instructions and data-types—we can define an ISP description of a processor. A processor is completely described at the ISP level by giving its *instruction-set* and its *interpreter* in terms of its *operations*, *data-types* and *memories*.

Let us first give the instruction-set. The effect of each instruction is described by an *instruction-expression*, which has the form:

$$\text{condition} \rightarrow \text{action-sequence}.$$

The *condition* describes when the instruction will be evoked, and the *action-sequence* describes what transformations of data take place between what memories. The right arrow ($\rightarrow$) is the control action (of a K) of evoking an *operation*.

Since all operations in a computer system result in modifications of bits in memories, each action in a sequence has the form:

$$\text{memory-expression} \leftarrow \text{data-expression}$$

The left arrow ($\leftarrow$) is the transmit operation of a link, and corresponds to the ALGOL assign operation. The left side must describe the memory location that is affected; the right side must describe the information pattern that is to be placed in that memory location. The details of data-expressions and memory expressions are patterned on standard mathematical notation, and are communicated most easily by examples. The same is true of the condition, which is a standard expression involving boolean values and relations among memory contents.

There are two important features of the action-sequence. The first is that each action in the sequence may itself be conditional; i.e., of the form, "condition $\rightarrow$ action-sequence." The second is that some actions are sequentially dependent on each other, because the result of one is used as an input to the other; on other occasions a set of actions are independent, and can occur in parallel. The normal situation is the parallel one. For example, if A and B are two registers, then

$$(\text{A} \leftarrow \text{B};\ \text{B} \leftarrow \text{A});$$

exchanges the contents of A and B. When sequence is required, the term 'next' is used; thus,

$$(\text{A} \leftarrow \text{B};\ \text{next B} \leftarrow \text{A});$$

transfers the contents of B to A and then transfers it back to B, leaving both A and B holding the original contents of B (equivalent to A $\leftarrow$ B).

### *An ISP example using the DEC PDP-8 Pc*

The memories, operations, instructions, and data-types all need to be declared for a processor. Again

* In principle, this view that only *D* components do "real" processing is false. It can be shown that a universal Turing Machine can be built from *M*, *S*, *L*, and *K* components. The key operation is the write operation into *M*, which suffices to construct arbitrary bit patterns under suitably controlled switches. Hence, arbitrary data operations can be built up. The stated view is correct in practice in that the data operations provided in a *P* are highly efficient for their bit transformations. Only the foolish add integers in a modern computer by table look up.

these are most easily explained by example, although full definitions exist (Bell and Newell, 1970). Consequently, let us examine the ISP description of the Pc of the PDP-8, given in Figure 4.

### Processor state

We first need to specify the memories of the Pc in detail, providing names for the various bits. Thus,

AC⟨0:11⟩ *the accumulator*

is a memory called AC, with 12 bits, labeled from 0 to 11 from the left. Comments are given in italics*—in this case that AC is called the accumulator (by the designers of the PDP-8). Alternatively, we could have used the alias or abbreviation convention:

AC⟨0:11⟩/Accumulator⟨0:11⟩.

* There are a few features of the notation, such as the use of italics, which are not easily carried over into current computer character sets. Thus, the ISP of Figure 4 is a publication language.

DEC PDP-8 ISP Description

```
Pc State
   AC<0:11>                                          Accumulator
   L                                                 Link bit/AC extension for overflow and carry
   PC<0:11>                                          Program Counter
   Run                                               1 when Pc is interpreting instructions or "running"
   Interrupt_state                                   1 when Pc can be interrupted; under programmed control
   IO_pulse_1; IO_pulse_2; IO_pulse_4                IO pulses to IO devices

Mp State
Extended memory is not included.
   M[0:7777₈]<0:11>
      Page_0[0:177₈]<0:11> := M[0:177₈]<0:11>        special array of directly addressed memory registers
      Auto_index[0:7]<0:11> := Page_0[10₈:17₈]<0:11> special array when addressed indirectly, is incremented by 1

Pc Console State
Keys for start, stop, continue, examine (load from memory), and deposit (store in memory) are not included.
   Data switches<0:11>                               data entered via console

Instruction Format
   Instruction/I<0:11>
      op<0:2>              := I<0:2>                 op code
      indirect_bit/ib      := I<3>                   0, direct; 1 indirect memory reference
      page_0_bit/p         := I<4>                   0 selects page 0; 1 selects this page
      page_address<0:6>    := I<5:11>
      this_page<0:4>       := PC'<0:4>
      PC'<0:11>            := (PC<0:11> -1)
      IO_select<0:5>       := I<3:8>                 selects a T or Ms device
      io_p1_bit            := I<11>                  these 3 bits control the selective generation of -3 volts,
      io_p2_bit            := I<10>                   0.4 μs pulses to I/O devices
      io_p4_bit            := I<9>
      sma                  := I<5>                   μ bit for skip on minus AC, operate 2 group
      sza                  := I<6>                   μ bit for skip on zero AC
      snl                  := I<7>                   μ bit for skip on non zero Link

Effective Address Calculation Process
   z<0:11> := (                                      effective
      ¬ib → z";
      ib ∧ (10₈ ≤ z" ≤ 17₈) → (M[z"] ← M[z"] + 1; next);   auto indexing
      ib → M[z"])
   z'<0:11> := (¬ ib → z"; ib → M[z"])
   z"<0:11> := (page_0_bit → this_page□page_address;      direct address
                ¬page_0_bit → 0□page_address)

μ microcoded instruction or instruction bit(s) within an instruction
```

```
Instruction Interpretation Process
   Run ∧ ¬ (Interrupt_request ∧ Interrupt_state) → (          no interrupt interpreter
      Instruction ← M[PC]; PC ← PC + 1; next                  fetch
      Instruction_execution);                                 execute
   Run ∧ Interrupt_request ∧ Interrupt_state → (              interrupt interpreter
      M[0] ← PC; Interrupt_state ← 0; PC ← 1)

Instruction Set and Instruction Execution Process
   Instruction_execution := (
      and (:= op = 0) → (AC ← AC ∧ M[z]);                     logical and
      tad (:= op = 1) → (L□AC ← L□AC + M[z]);                 two's complement add
      isz (:= op = 2) → (M[z'] ← M[z] + 1; next               index and skip if zero
                         (M[z'] = 0)L → (PC ← PC + 1))),
      dca (:= op = 3) → (M[z] ← AC; AC ← 0);                  deposit and clear AC
      jms (:= op = 4) → (M[z] ← PC; next PC ← z + 1);         jump to subroutine
      jmp (:= op = 5) → (PC ← z);                             jump
      iot (:= op = 6) → (                                     μ in out transfer, microprogrammed to generate up to 3 pulses
         io_p1_bit → IO_pulse_1 ← 1; next                       to an io device addressed by IO_select
         io_p2_bit → IO_pulse_2 ← 1; next
         io_p4_bit → IO_pulse_4 ← 1);
      opr (:= op = 7) → Operate_execution                     the operate instruction is defined below
                  )                                           end Instruction execution

Operate Instruction Set
The microprogrammed operate instructions: operate group 1, operate group 2, and extended arithmetic are defined as a separate
 instruction set.
   Operate_execution := (
      cla (:= i<4> = 1) → (AC ← 0);                           clear AC. Common to all operate instructions.
      opr_1 (:= i<3> = 0) → (                                 operate group 1
         cll (:= i<5> = 1) → (L ← 0); next                    μ clear link
         cma (:= i<6> = 1) → (AC ← ¬ AC);                     μ complement AC
         cml (:= i<7> = 1) → (L ← ¬ L); next                  μ complement L
         iac (:= i<11> = 1) → (L□AC ← L□AC + 1); next         μ increment AC
         ral (:= i<8:10> = 2) → (L□AC ← L□AC x 2 {rotate});   μ rotate left
         rtl (:= i<8:10> = 3) → (L□AC ← L□AC x 2^2 {rotate}); μ rotate twice left
         rar (:= i<8:10> = 4) → (L□AC ← L□AC / 2 {rotate});   μ rotate right
         rtr (:= i<8:10> = 5) → (L□AC ← L□AC / 2^2 {rotate})); μ rotate twice right
      opr_2 (:= i<3,11> = 10) → (                             operate group 2
         skip condition ⊕ (i<8> = 1) → (PC ← PC + 1); next    μ AC,L skip test
            skip condition := ((sma ∧ (AC < 0)) ∨ (sza ∧ (AC = 0)) ∨ (snl ∧ L))
         osr (:= i<9> = 1) → (AC ← AC ∨ Data switches);       μ "or" switches
         hlt (:= i<10> = 1) → (Run ← 0));                     μ halt or stop
      EAE (:= i<3,11> = 11) → EAE_Instruction_execution)      optional EAE description
```

Figure 4—DEC PDP-8 ISP Description

AC corresponds to an actual register in the Pc. However, the ISP does not imply any particular implementation, and names may be assigned to various sets of bits purely for descriptive convenience. The colon is used to denote a range or list of values. Alternatively, we could have listed each bit, separating the bit names by commas, as:

$$AC\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\rangle.$$

Having defined a second memory, L (which has only a single bit), one could define a combined register, LAC, in terms of L, and AC, as:

$$LAC\langle L, 0{:}11\rangle := L\square AC.$$

The colon-equal (:=) is used for definition, and the middle square box (□) denotes concatenation. Note that the bit named L of register LAC corresponds to the 1 bit L register.

### Memory state

In dealing with addressed memory, either Mp or various forms of working memory within the processor, we need to indicate multidimensional arrays. Thus,

$$Mp[0:7777_8]\langle 0:11\rangle$$

gives the primary memory as consisting of $7777_8$ (i.e., base 8) words of 12 bits each, being addressed as indicated. Such an address does not necessarily reflect the switching structure through which the address occurs, though it often will. (Needless to say, it reflects only addressing space, and not how much actual M is available in a PMS structure.) In general, only memory within the processor will occur as operands of the processor's operators. The one exception is primary memory (Mp), which is defined as a memory external to a P, but directly accessible from it.

In writing memories it is natural to use base 10 for all numbers and to consider the basic i-unit of the memory to be a bit. This is always assumed unless otherwise indicated. Since we used base 8 numbers above for specifying the addressing range, we indicated the change of number base by a subscript, in standard fashion. If a unit of information other than the bit were to be used, we would subscript the angle brackets. Thus,

$$Mp[0:7777_8]\langle 0:1\rangle_{64}$$

reflects the same memory. The choice carries with it, of course, some presumption of organization in terms of base 64 characters—but this would show up in the specification of the operators (and is not true, in fact of the PDP-8). We can also have multi-dimensional memories (i.e., arrays), though no examples are used in Figure 4. These just add the extra dimensions with an extra pair of brackets. For example, a more precise description would have used:

$$Mp[0:7][0:31][0:127]\langle 0:11\rangle$$

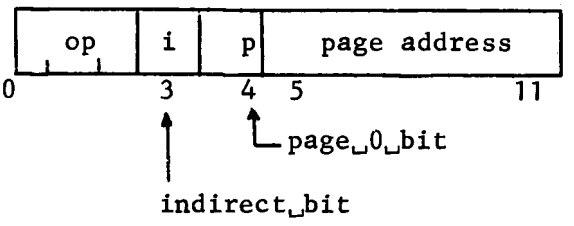
to mean 8 memory fields, each field with 32 pages, each page with 128 words and each word with 12 bits.

### Instruction format

It is possible to have several names for the same set of bits; e.g., having defined instruction ⟨0:11⟩ we define the format of the instruction as follows:

op⟨0:2⟩ := instruction ⟨0:2⟩
indirect␣bit := instruction⟨3⟩
page␣0␣bit := instruction⟨4⟩
page␣address⟨0:6⟩ := instruction⟨5:11⟩

The colon-equal (:=) is used to assign names to various parts of the instruction. In effect, this is a definition equivalent to the conventional diagram for the instruction:




Notice that in page␣address the names of all the bits have been shifted, e.g., page␣address⟨4⟩ := instruction⟨9⟩.

In general, a *name* can be any combination of upper and lower case letters and numerals; not including names which would be considered numbers (integers, mixed numbers, fractions, etc.). A compound name can be sequences of names separated by spaces ( ) or a hyphen. In order to make certain compound names more recognizable, a space symbol (␣) may optionally be used to signify the non-printing character.

### The instruction set

With all the registers defined, the instructions can be given. These are shown on the second page of Figure 4. The second page is a single expression, named Instruction␣execution, which consists of a list of instructions. These are listed vertically down the page for ease of reading. Each instruction consists of a condition and an action sequence, separated by the condition-arrow (→). In this case the condition is an expression of the form (op = octal-digit). Since op is instruction⟨0:2⟩, this expresses the condition that the operation code of the instruction has a particular value. Each condition has been given a name in passing; e.g., 'and' is the name of (op = 0). This provides the correspondence between the operation code and the mnemonic name of the operation code. If this correspondence had been established elsewhere, or if we didn't care what numerical operation code the "and" instruction is, we could have written:

$$\text{and} \rightarrow (AC \leftarrow AC \wedge M[z])$$

We would not have known what condition the name 'and' stood for, but could have surmised (with little difficulty) that it was simply an equality test on the operation code. Or we could define it elsewhere as:

$$\text{and} := (op = 0)$$

Most generally the form of an instruction is written as:

two's␣complement␣add/tad(:= op = 1) →

(L□AC ← L□AC + M[z])

Here, we simultaneously define the action of the tad instruction, its name, an abbreviation for the name, and the conditions for tad's execution. The first parentheses are, in effect, a remark to allow an in-line definition.

The instructions in the list constitute the total instruction repertoire of the Pc. Since all the conditions are disjoint, one and only one condition will be satisfied when a given instruction is interpreted, hence one and only one action sequence will occur. Actually, all operation codes might not be present, so there would be some illegal op codes that would evoke no action sequence. The act of selection is usually called operation decoding. Here, ISP implies no particular mechanism by which this is carried out.

It might be wondered why the conventions are not more stylized—e.g., some sort of table with mnemonic names in one column, bits of the operation code in another, etc. Though standard processors would fit such a stylized scheme, many others would not—e.g., microprogram processors. By making the ISP description a general expression for evoking action-sequences we obtain the generality needed to cover all variations. (Indeed, you will notice that the PDP-8 ISP is a single expression, and that it incorporates two microprogrammed instructions with no difficulty.)

For the action-sequence standard mathematical infix notation is used. Thus we write

$$AC \leftarrow AC \wedge M[z]$$

This indicates that the word in Mp at address z (determined by the expression on page 1 of Figure 4) is anded with the accumulator and the result left in the accumulator. Each processor will have a basic set of operations that work on data-types of the machine. Here the data-type is simply the 12 bit word viewed as an array of bits.

Operators need not necessarily involve memories actually within the Pc (the processor state). Thus,

$$Mp[z] \leftarrow Mp[z] + 1$$

expresses a change in a word in Mp directly. That this must be mechanized in the PDP-8 by means of some temporary register in Pc is irrelevant to the ISP description.

We also use functional notation, e.g.,

$$AC \leftarrow abs(AC)$$

replaces the contents of the AC with its absolute value.

**Effective address calculation**

In the examples just given we used z as the address in Mp. This is the effective address (simplified) and is defined as a conditional expression (in the manner of ALGOL or LISP):

$$z\langle 0{:}11\rangle := (\neg\, \text{indirect-bit} \rightarrow z';\ \text{indirect-bit} \rightarrow Mp[z'])$$

The right arrow (→) is the same conditional sign used in the main instruction, similar to the "if . . . then . . ." of ALGOL. The parentheses are used to indicate grouping in the usual fashion. However, we arrange expressions on the page to make reading easier.

As the expression for z shows, we permit conditional within conditionals, and also the nesting of definitions (z is defined in terms of a variable z'). Again, we should emphasize that the structure of such definitions may reflect directly the underlying hardware organization, but it need not. When describing existing processors the ISP description often does or can be forced to reflect the hardware. But if one were designing a processor, then ISP expressions would be put down as design objectives to be implemented in a register transfer structure, which might differ considerably.

Special note should be taken of the opr instruction (op = 6) in Figure 4, since it provides a microprogramming feature. There are two separate options depending on instruction⟨3⟩ being 0 or 1. But common to both of these is the operation of clearing the AC (or not), associated with instruction ⟨4⟩. Then, within one option (instruction⟨3⟩ = 0) there are a series of independently executable actions (following the clearing of L); within the other (instruction⟨3⟩ = 1), there are three independently settable control actions. The nested conditionals and the use of 'next' to force sequential behavior make it easy to see exactly what is going on (in fact a good deal easier than describing it in natural language, as we have been doing).

**The instruction interpreter**

From the hardware point of view, an interpreter consists of the mechanisms for fetching a new instruction, for decoding that instruction and executing the operations so designated, and for determining the next instruction. A substantial amount of this total job has already been taken care of in the part of the ISP that we have just explained. Each instruction carries with it a condition that amounts to one fragment of the decoding operation. Likewise, any further decoding of the instruction that might be done in common by the interpreter (rather than by the indi-

vidual operation circuits) is implied in the expressions for each instruction, and by the expression for the effective address. The interpreter then fetches the next instruction and executes it.

In a standard machine, there is a basic principle that defines operationally what is meant by the "next instruction." Normally the current instruction address is incremented by one, but other principles are used (e.g., on a processor with a cyclic Mp). In addition, several specific operations exist in the repertoire that can affect what program is in control. The basic principle acts like a default condition—if nothing specific happens to determine program control, the normal "next" instruction is taken. Thus, in the PDP-8 we get an interpretation process that is the classic fetch-execute cycle:

$$\text{Run} \rightarrow (\text{instruction} \leftarrow \text{Mp[PC]};\ \text{PC} \leftarrow \text{PC} + 1;\ \text{next Instruction}\_\text{execution})$$

The sequence is evoked so long as Run is true (i.e., its bit value is 1). The processor will simply cycle through the sequence, fetching, then executing the instruction. In the PDP-8 there exists a halt operation that sets Run to be 0, and the console keys can, of course, stop the computer. It should be noted that this ISP description does not include console behavior, although it could.

The ISP description does not determine the way the processor is to be organized to achieve this sequencing, or to take advantage of the fact that many instructions lead to similar sequences. All it does is specify what operations must be carried out for a program in Mp. The ISP description does specify the actual format of the instruction and how it enters into the total operation, although sometimes indirectly. For example, in the case of the *and* operation (op = 0), the definition of AC shows that the AC does not depend on the instruction and the definition of z shows that z does depend on other fields of the instruction (indirect_bit, page_0_bit, page_address). Likewise, the form of the ISP expression shows that AC and PC both enter into the instruction implicitly. That is, in the ISP description all dependence on memory is explicit.*

*Data-types and data operations*

Each data-type has a set of operations that are proper to it. Add, subtract, multiply and divide are all proper to any numerical data-type, as well as absolute value and negation. Not all of these need exist in a computer just because it has the data-type, since there are several alternative bases, as well as some levels of completeness. For instance, notice that the PDP-8 first of all does not have multiply and divide (unless one has its special option), thus having a relatively minimal level of arithmetic operations; and second, it does not have a subtract operation, using a two's complement add, which permits negation ($-$AC) to be accomplished by complementation ($\wedge$AC) followed by add 1.

The PDP-8, unlike larger C's, does not have several data representations for what is, externally considered, the same entity. An operator that does a floating add and one that does an integer add are not the same. However, we denote both by the same symbol (in this case, +), indicating the difference parenthetically after the expression. Alternatively, the specification of the data-type can be attached to the data. Thus, in the IBM 7094 we would see the following add instructions:

$$\text{Add/ADD} \leftarrow (\text{AC} \leftarrow \text{AC} + \text{M[e]});$$

$$\text{Add and Carry Logical/ACL} \rightarrow (\text{AC} \leftarrow \text{AC} + \text{M[e]\{sl\}}).$$

$$\text{Floating add/FAD} \rightarrow (\text{AC} \leftarrow \text{AC} + \text{M[e]\{sf\}});$$

$$\text{Un-normalized floating add/UFA} \rightarrow (\text{AC} \leftarrow \text{AC} + \text{M[e]\{suf\}});$$

$$\text{Double precision floating add/DFAD} \rightarrow (\text{ACMQ} \leftarrow \text{ACMQ} + \text{M[e]}\square\text{M[e} + 1]\{\text{df}\});$$

$$\text{Double precision un-normalized floating add/DUFA} \rightarrow (\text{ACMQ} \leftarrow \text{ACMQ} + \text{M[e]}\square\text{M[e} + 1]\{\text{duf}\})$$

The braces { } differentiate which operation is being performed. Thus above, the data-type* is enclosed in the braces and refers to all the memory elements (operands) of the expression. Alternatively, we also use braces as a modifier to signify the encoding of the i-unit. For example, a fixed point to floating point data conversion operation would be given:

$$\text{AC\{floating\}} \leftarrow \text{AC\{fixed\}}.$$

We also use the braces as a modifier for the operation type. For example, shifting (left or right) can be a

* This is not correct, actually. In physically realizing an ISP description, additional memories may be utilized (they may even be necessary). It can be said that the ISP description has these memories implicitly. However, it is the case that a consistent and complete description of an ISP can be made without use of these additional memories; whereas with, say, a single address machine, it does not seem possible to describe each instruction without some reference to the implicit memories—as we see in the effective address calculation procedures where definitions look much like registers.

* The conventions for naming data-types is a concatenation of precision, a name and a structure. Examples include i/integer; di/double integer; div/double integer vector; single floating/sf; suf/single unnormalized floating; bv/boolean vector; ch.string/character string.

multiplication or division by a base, but it is not always an arithmetic operation. In the PDP-8, for instance, we had

$$L\square AC \leftarrow L\square AC \times 2\{\text{rotate}\};$$

where the end bits L and AC⟨11⟩ are connected when a shift occurs (the operator is also referred to as a circular shift), or equivalently

$$(L\square AC \leftarrow L\square AC \times 2;\ AC\langle 11\rangle \leftarrow L).$$

In general, the nature of the operations used in processors are sufficiently familiar that no definitions are required, and they can all be taken as primitive. It is only necessary to have agreed upon conventions for the different data representations used. In essence, a data-type is made up recursively of a concatenation of subparts, which themselves are data types. This concatenation may be an iteration of a data-type to form an array.

If required, an operation can be defined in terms of other (presumably more primitive) operations. It is necessary, of course, first to define the data format explicitly (including perhaps some additional memory). Variables for the operands are permitted in the natural way. For example, binary single precision floating point multiplication on a 36 bit machine could be defined in terms of the data fields as follows:

```
sf mantissa/mantissa      := ⟨0:27⟩
sf sign/sign              := ⟨0⟩
sf exponent/exponent      := ⟨28:35⟩
sf exponent␣sign          := ⟨28⟩
x1 ← x2 × x3{sf}          := (x1 mantissa := x2 mantissa × x3 mantissa;
                              x1 exponent := x2 exponent + x3 exponent; next
                              x1 := normalize(x1){sf})
```

where normalize is:

```
x1 ← normalize(x2) {sf}   := (
      (x1 mantissa = 0)   → (x1 exponent := 0)
      (x2 mantissa ≠ 0) ∧ (x2⟨0⟩ = x2⟨1⟩) → (
                  x1 mantissa := x2 mantissa × 2;
                  x1 exponent := x2 exponent-1; next
                  x1 := normalize (x2){sf}))
```

Three additional aspects need to be noted with respect to data-types; two substantive, one notational. First, not everything one does with an item of data makes use of all the properties of its data-type. For example, numbers have to be moved from place to place. This operation is not a numerical operation, and does not depend on the item being a number. Second, one can often embed one kind of operation in another, so as to coalesce data-types. An example is encoding the Mp addresses into the same integer data-type as are used for regular arithmetic. Then there need be no separate data-type for addresses.*

The notational aspect is our use in ISP of an mnemonic abbreviation scheme for data-types. We have already used sf for single-precision-floating-point. More generally, an abbreviation is made up of a letter showing the length, a letter showing the type, and a letter showing the structure. The simple naming convention does not take into account all we know about a data-type. The information carrier for the data is only partially included in the length characteristic. Thus the carrier should also include the data base and the sign convention for representing negative numbers, (e.g., sign-magnitude).

### *PMS structure of the CDC 6600 series*

A simplified PMS structure of the C('6400|'6600) is given in Figure 5. Here we see the C(io; #1:10) each of which can access the primary memory (Mp) of the central computer (Cc). Figure 5 shows why one considers the 6600 to be a network. Each Cio (actually a general purpose, 12 bit C) can easily serve the specialized Pio function for Cc. The Mp of Cc is an Ms for a Cio because the Cio cannot execute programs from this memory. By having a powerful Cio more complex input-output tasks can be handled without Cc intervention. These tasks can include data-type conversion, error recovery, etc. The K's which are connected to a Cio can also be less complex.

A detailed PMS diagram for the C('6400, '6416, '6500, *and* '6600) is given in Figure 6. The interesting structural aspects can be seen from this diagram. The four configurations, 6400 ~ 6600, are included just by considering the pertinent parts of the structure. That

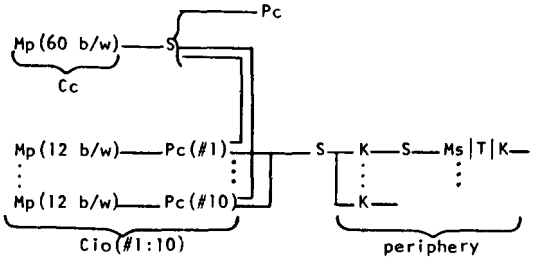
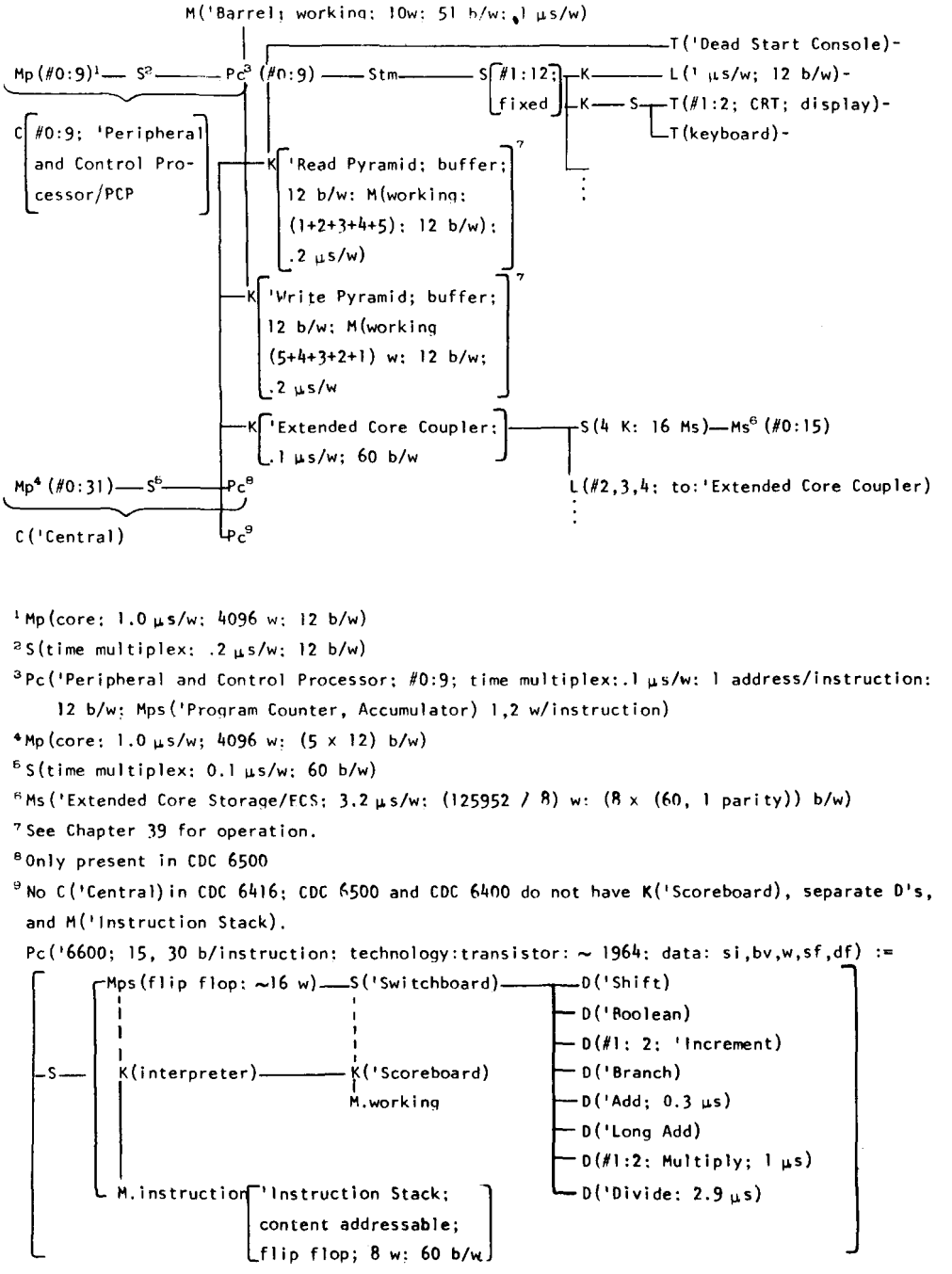



Figure 5—CDC 6600 PMS diagram (simplified)

* However logical such a course may seem, it is not always done this way. For example, the IBM 7090 (and other members of that family) have a 15 bit address data type and a 36 bit integer data type, with separate operations for each.




[1] Mp(core: 1.0 μs/w: 4096 w: 12 b/w)

[2] S(time multiplex: .2 μs/w: 12 b/w)

[3] Pc('Peripheral and Control Processor; #0:9; time multiplex:.1 μs/w: 1 address/instruction: 12 b/w; Mps('Program Counter, Accumulator) 1,2 w/instruction)

[4] Mp(core: 1.0 μs/w; 4096 w: (5 x 12) b/w)

[5] S(time multiplex: 0.1 μs/w: 60 b/w)

[6] Ms('Extended Core Storage/ECS; 3.2 μs/w: (125952 / 8) w: (8 x (60, 1 parity)) b/w)

[7] See Chapter 39 for operation.

[8] Only present in CDC 6500

[9] No C('Central) in CDC 6416; CDC 6500 and CDC 6400 do not have K('Scoreboard), separate D's, and M('Instruction Stack).

Pc('6600; 15, 30 b/instruction: technology:transistor: ~ 1964; data: si,bv,w,sf,df) :=




Figure 6—CDC 6600 PMS Diagram

is, a 6416 has no large Pc; a 6400 has a single straightforward Pc; a 6500 has two Pc's; and the 6600 has a single powerful Pc. The 6600 Pc has 10 D's, so that several parts of a single instruction stream can be interpreted in parallel. A 6600 Pc also has considerable M.buffer to hold instructions so that Pc need not wait for Mp fetches.

The implementation of the 10 Cio's can be seen from the PMS diagram (Figure 6). Here, only 1 physical processor is used on a time shared basis. Each 0.1 μs a new logical P is processed by the physical P. The 10 Mp's are phased so that a new access occurs each 0.1 μs. The 10 Mp's are always busy. Thus, the information rate is (10 × 12) b/μs or 120 megabit/s.

This structure for shifting a new Pc state into position each 0.1 $\mu$s has been likened by CDC to a barrel.

The T's, K's and M's are not given in the figures, although it should be mentioned that the following units are rather unique: a K for the management of 64 telegraph lines to be connected to a Cio; and Ms(disk) with four simultaneous access ports, each at 1.68 megachar/s data transfer rate; and a capacity of 168 megachar; a Ms(magnetic tape) with a K(#1:4) and S to allow simultaneous transfers to 4 Ms; the

```
CDC 6400, 6500, 6600 Central Processor ISP Description

Pc State
   P<17:0>                                  Program counter
   X[0:7]<59:0>                             Main arithmetic registers. X[1:5], are implicitly loaded from
   A[0:7]<17:0>                              Mp when A[1:5] are loaded.  X[6:7] are implicitly stored in
                                             Mp when A[6:7] are loaded.
   B[0]<17:0>  := 0                         B registers are general arithmetic registers, and can be used
   B[1:7]<17:0>                              as index registers.
   Run                                      1 if interpreting instructions, not under program control.
   EM<17:0>                                 Exit mode bits
      Address␣out␣of␣range␣mode  := EM<12>
      Operand␣out␣of␣range␣mode  := EM<13>
      Indefinite␣operand␣mode    := EM<14>
 The above description is incomplete in that the above 3 mode's alarm allow conditions to trap Pc at Mp[RA]. Trapping occurs if
 an alarm condition occurs "and" the mode is a one.

Mp State
   Mp[0:777777₈]<59:0>                      main core memory of 2^18 w, (256 kw)
   Ms[0:2015232]<59:0>                      ECS/Extended Core Storage Program can only transfer data between
                                             Mp and Ms.  Program cannot be executed in Ms.
   RA<17:0>                                 reference (or relocation) address register to map a logical Mp'
                                             into physical Mp
   FL<17:0>                                 field length - the bounds register which limits a program's
                                             access to a range of Mp'
   RAECS<59:36>                             reference or relocation register for Ms(Extended Core Storage)
   FLECS<59:36>                             field length for ECS
   Address␣out␣of␣range                     a bit denoting a state when memory mapping is invalid

Memory Mapping Process
 This process maps or relocates a logical program, at location Mp', and Ms', into physical Mp and Ms.
   Mp'[X]  := ((X < FL) →Mp[X + RA]);       logical Mp'
              (X ≥ FL) →(Run ←0; Address␣out␣of␣range ←1))
   Ms'[X]  := ((X < FLECS) →Ms[X]+ RAECS]); logical Ms'
              (X ≥ FLECS) →(Run ←0; Address␣out␣of␣range ←1))

Exchange jump storage allocation map at location, n within Mp:
 The following Mp" array is reserved when Pc state is stored, and switched to another job.  The exchange jump instruction in
 a Peripheral and Control Processor enacts the operation:  (Mp" ← Mp; Mp ← Mp").
   Mp"[n]<53:0>     := P□A[0]□000000₈
   Mp"[n+1]<53:0>   := RA□A[1]□B[1]
   Mp"[n+2]<53:0>   := FL□A[2]□B[2]
   Mp"[n+3]<53:0>   := EM□A[3]□B[3]
   Mp"[n+4]         := RAECS□A[4]□B[4]
   Mp"[n+5]         := FLECS□A[5]□B[5]
   Mp"[n+6]<35:0>   := A[6]□B[6]
   Mp"[n+7]<35:0>   := A[7]□B[7]
   Mp"[n+10₈:n+17₈]:= X[0:7]
```

Figure 7—CDC 6400, 6500, 6600 Central Processor ISP Description

T(direct; display) for monitoring the system's operation; K's to other C's and Ms's; and conventional T(card reader, punch, line-printer, etc.).

## ISP OF THE CDC 6600

The ISP description of the Pc is given in Figure 7. The Pc has a straightforward scientific calculation

```
Instruction Format
   instruction<29:0>                                  although 30 bits, most instructions are 15 bits; see
                                                       Instruction Interpretation Process
      fm<5:0>          := instruction<29:24>          operation code or function
      fmi<8:0>         := fm□i                        extended op code
      i<2:0>           := instruction<23:21>          specifies a register or an extension to op code
      j<2:0>           := instruction<20:18>          specifies a register
      k<2:0>           := instruction<17:15>          specifies a register
      jk<5:0>          := j□k                         a shift constant (6 bits)
      K<17:0>          := instruction<17:0>           an 18 bit address size constant
      long_instruction := ((fm < 10_8) ∨              30 bit instruction
                           (50 < fm < 53) ∨
                           (60 < fm < 63) ∨
                           (70 < fm < 73))
   short_instruction   := ¬ long instruction          15 bit instruction

Instruction Interpretation Process
 A 15 bit (short) or 30 bit (long) instruction is fetched from Mp'[P]<p x 15 + 15 - 1:p x 15> where p = 3, 2, 1, or 0.  A 30
 bit instruction cannot be stored across word boundaries (or in 2, Mp' locations).
   p<1>_4                                             a pointer to 15 bit quarter word which has instruction
   Run →(instruction<29:15> ←Mp'[P]<(p x 15 + 14):(p x 15)>; next      fetch
      p ←p - 1; next
      (p = 0) ∧ long_instruction →Run ←0;
      (p ≠ 0) ∧ long_instruction → (
         instruction<14:0> ←Mp'[P]<(p x 15 + 14):(p x 15)>;
           p ←p - 1); next
      Instruction_execution; next                                  execute
      (p = 0) →(p ← 3; P ← P + 1))

Instruction Set and Instruction Execution Process
 Operand fetches or stores between Mp' and X[i] occur by loading or storing registers A[i].  If (0 < i < 6) a fetch from
 Mp'[A[i]] occurs.  If (i ≥ 6) a store is made to Mp'[A[i]].  The description does not describe Address_out_of_range case,
 which is treated like a null operation.
   Instruction_execution := (
   Set A[i]/XA
      "SAi Aj + K"  (fm = 50) → (A[i] ←A[j] + K; next Fetch_Store);
      "SAi Bj + K"  (fm = 51) →(A[i] ←B[j] + K; next Fetch_Store);
      "SAi Xj + K"  (fm = 52) →(A[i] ←X[j]<17:0> + K; next Fetch_Store);
      "SAi Xj + Bk" (fm = 53) →(A[i] ←X[j]<17:0> + B[k]; next Fetch_Store);
      "SAi Aj + Bk" (fm = 54) →(A[i] ←A[j] + B[k]; next Fetch_Store);
      "SAi Aj - Bk" (fm = 55) →(A[i] ←A[j] - B[k]; next Fetch_Store);
      "SAi Bj + Bk" (fm = 56) →(A[i] ←B[j] + B[k]; next Fetch_Store);
      "SAi Bj - Bk" (fm = 57) →(A[i] ←B[j] - B[k]; next Fetch_Store);
      Fetch_Store := (
         (0 < i < 6) →(X[i] ←Mp'[A[i]]);                  process to get operand in X or store operand from X when A
         (i ≥ 6) →(Mp'[A[i] ← X[i]))                       is written
   Operations on B and X
   Set B[i]/SBi
      "SBi Aj + K" (fm = 60) →(B[i] ←A[j] + K);
```

Figure 7 (Continued)

oriented ISP with 48 bit mantissa single precision floating point (also double precision floating point operations is provided). The Pc state has three sets of general registers. This structure assumes that a program consists of several read accesses to a large array(s), a large number of operations on these accessed ele-

```
    "SBi Bj + K" (fm = 61) → (B[i] ← B[j] + K);
    "SBi Xj + K" (fm = 62) → (B[i] ← X[j]<17:0> + K);
    "SBi Xj + Bk" (fm = 63) → (B[i] ← X[j]<17:0> + B[k]);
    "SBi Aj + Bk" (fm = 64) → (B[i] ← A[j] + B[k]);
    "SBi Aj - Bk" (fm = 65) → (B[i] ← A[j] - B[k]);
    "SBi Bj + Bk" (fm = 66) → (B[i] ← B[j] + B[k]);
    "SBi Bj - Bk" (fm = 67) → (B[i] ← B[j] - B[k]);
Set X[i]/SXi
    "SXi Aj + K" (fm = 70) → (X[i] ← sign_extend(A[j] + K));
    "SXi Bj + K" (fm = 71) → (X[i] ← sign_extend(B[j] + K));
    "SXi Xj + K" (fm = 72) → (X[i] ← sign_extend(X[j] + K));
    "SXi Xj + Bk" (fm = 73) → (X[i] ← sign_extend(X[j] + B[k]));
    "SXi Aj + Bk" (fm = 74) → (X[i] ← sign_extend(A[j] + B[k]));
    "SXi Aj - Bk" (fm = 75) → (X[i] ← sign_extend(A[j] - B[k]));
    "SXi Bj + Bk" (fm = 76) → (X[i] ← sign_extend(B[j] + B[k]));
    "SXi Bj - Bk" (fm = 77) → (X[i] ← sign_extend(B[j] - B[k]));
Miscellaneous program control
    "PS" (:= fm = 0) → (Run ← 0);                                  program stop
    "NO" (:= fm = 46) → ;                                          no operation; pass
Jump unconditional
    "JP Bi + K" (:= fm = 02) → (P ← B[i] + K; p ← 3);              jump
Jump on X[j] conditions
    "ZR Xj K" (:= fmi = 030) → ((X[j] = 0) → (P ← K; p ← 3));      zero
    "NZ Xj K" (:= fmi = 031) → ((X[j] ≠ 0) → (P ← K; p ← 3));      non zero
    "PL Xj K" (:= fmi = 032) → ((X[j] ≥ 0) → (P ← K; p ← 3));      plus or position
    "NG Xj K" (:= fmi = 033) → ((X[j] < 0) → (P ← K; p ← 3));      negative
    "IR Xj K" (:= fmi = 034) → (                                   out of range constant tests
    ¬((X[j]<59:48>= 3777)∨ (X[j]<59:48>= 4000)) →P ← K; p ← 3);
    "OR Xj K" (:= fmi = 035) → (
       (X[j]<59:48>=3777) ∨ (X[j]<59:48>=4000)→ (P ← K; p ← 3));
    "DF Xj K" (:= fmi = 036) → (                                   indefinite form constant tests
       (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000)  →(P ← K; p ← 3));
    "ID Xj K" (:= fmi = 037) → (
       (X[j]<59:48>=1777) ∨ (X[j]<59:48>=6000)   (P ← K; p ← 3));
Jump on B[i], B[j] comparison
    "EQ Bi Bj K" (:= fm = 04) →((B[i] = B[j]) →(P ← K; p ← 3));    equal
    "NE Bi Bj K" (:= fm = 05) →((B[i] ≠ B[j]) →(P ← K; p ← 3));    not equal
    "GE Bi Bj K" (:= fm = 06) →((B[i] ≥ B[j]) →(P ← K; p ← 3));    greather than or equal
    "LT Bi Bj K" (:= fm = 07) →((B[i] < B[j]) →(P ← K; p ← 3));    less than
Subroutine call
    "RJ K" (:= fmi = 010) →(                                       return jump
        M[K]<59:30> ←04₈□00₈□(P + 1)□000000₈; next
        (P ←K + 1; p ←3));
Peading (REC) and writing (WEC) Mp with Extended Core Storage, subjected to bounds checks, and Ms', Mp' mapping
    "REC Bj + K" (:= fmi = 011) →(                                 read extended core
```

Figure 7 (Continued)

ments, followed by occasional write accesses to store results.

Cc has provisions for multiprogramming in the form of a protection and relocation address. The mapping is given in the ISP description for both Mp, but an Ms('Extended Core Storage/ECS) is not described.

```
    Mp'[A[0]:A[0] + B[j] + K-1] ← Ms'[X[0]:X[0] + B[J] + K-1];
  "WFC Bj + K" (:= fmi = 012) → (                                  write extended core
    Ms'[X[0]:X[0] + B[j] + K-1]  Mp'[A[0]:A[0] + B[J] + K-1]);
Fixed Point Arithmetic and Logical operations using X
  "IXi Xj + Xk" (:= fm = 36)  ·(X[i] ← X[j] + X[k]);               integer sum
  "IXi Xj - Xk" (:= fm = 37)  ·(X[i] ← X[j] - X[k]);               integer difference
  "CXi Xk" (:= fm = 47)  ·(X[i] ← sum_modulo_2(X[k]);              count the number of bits in X[k]
  "BXi Xj" (:= fm = 10₈)  ·(X[i] ← X[j]);                          transmit
  "BXi Xj * Xk" (:= fm = 11₈)  ·(X[i] ← X[j] ∧ X[k]);              logical product
  "BXi Xj + Xk" (:= fm = 12)  ·(X[i] ← X[j] ∨ X[k]);               logical sum
  "BXi Xj - Xk" (:= fm = 13)  ·(X[i] ← X[j] ⊕ X[k]);               logical difference
  "BXi - Xk" (:= fm = 14)  ·(X[i] ← ¬ X[k]);                       transmit complement
  "BXi - Xk * Xj" (:= fm = 15)  ·(X[i] ← X[j] ∧ ¬ X[k]);           logical product and complement
  "BXi - Xk + Xj" (:= fm = 16)  ·(X[i] ← X[j] ∨ ¬ X[k]);           logical sum and complement
  "BXi - Xk - Xj" (:= fm = 17)  ·(X[i] ← X[j] ⊕ ¬ X[k]);           logical difference and complement
  "LXi jk" (:= fm = 20)  ·(X[i] ← X[i] × 2^jk {rotate});
  "AXi jk" (:= fm = 21)  →(X[i] ← X[i] / 2^jk);                    arithmetic right shift
  "LXi Bj Xk" (:= fm = 22)  →(                                     left shift nominally
    ¬B[j]<17> → X[i] ← X[k] × 2^B[j]<5:0> {rotate};
     B[j]<17> → X[i] ← X[k] / 2^¬ B[j]<10:0>);
  "AXi Bj Xk" (:= fm = 23)  →(                                     arithmetic right shift nominally
    ¬B[j]<17> → X[i]←X[k] / 2^B[j]<10:0>;
     B[j]<17> → X[i] ← X[k] × 2^¬ B[j]<5:0> {rotate});
  "MXi jk" (:= fm = 43)  →(                                        form mask
    X[i]<59:59-jk+1> ← 2^jk - 1;
    (jk = 0) → X[i] ← 0);
Floating Point Arithmetic using X
 Only the least significant (lo) part of arithmetic is stored in Floating LP operations.
  "FXi Xj + Xk" (:= fm = 30) → (X[i] ← X[j] + X[k] {sf});          floating sum
  "FXi Xj - Xk" (:= fm = 31) → (X[i] ← X[j] - X[k] {sf});          floating difference
  "DXi Xj + Xk" (:= fm = 32) → (X[i] ← X[j] + X[k] {ls.df});       floating dp sum
  "DXi Xj - Xk" (:= fm = 33) → (X[i] ← X[j] - X[k] {ls.df});       floating dp difference
  "RXi Xj + Xk" (:= fm = 34) → (
    X[i] ← round(X[j]) + round(X[k]) {sf});
  "RXi Xj - Xk" (:= fm = 35) → (                                   round floating difference
    X[i] ← round(X[j]) - round(X[k]) {sf});
  "FXi Xj * Xk" (:= fm = 40) → (X[i] ← X[j] × X[k] {sf});          floating product
  "RXi Xj * Xk" (:= fm = 41) → (                                   round floating product
    X[i] ← X[j] × X[k] {sf}; next X[i] ← round(X[i]) {sf});
  "DXi Xj * Xk" (:= fm = 42) → (X[i] ← X[j] × X[k] {ls.df});       floating dp product
  "FXi Xj / Xk" (:= fm = 44) → (X[i] ← X[j] / X[k] {sf});          floating divide
  "RXi Xj / Xk" (:= fm = 45) → (X[i] ← round(X[j] / X[k]) {sf});   round floating divide
  "NXi Bj Xk" (:= fm = 24) → (                                     normalize
    X[i] ← normalize(X[k]) {sf};
    B[j] ← normalize_exponent(X[k]) {sf});
```

Figure 7 (Continued)

```
"ZXi BJ Xk" (:= fm = 25) → (                                    round and normalize
   X[i] ← round(X[k]) {sf}; next
   X[i] ← normalize(X[i]) {sf};
   B[J] ← normalize_exponent(X[i]) {sf});
"UXi BJ Xk' (:= fm = 26) → (B[J] ← X[k]<58:48> {si};            unpack
                            X[i] ← X[k]<59,47:0> {si});
"PXi BJ Xk" (:= fm = 27) → (X[k]<58:48> ← B[J] {si};            pack
                            X[k]<59,47:0> ← X[i] {si});
                         )                                      end Instruction_execution
```

Figure 7 (Continued)

## SUMMARY

We have introduced two notations for two aspects of the upper levels of computer systems: the topmost information-flow level, here called the PMS level (there being no other common name); and the interface between the programming level and the register transfer level, called ISP.

We were induced to create these notations as an aid in writing a book describing the architecture of many different computers—which served to make us painfully aware of the (dysfunctional) diversity that now exists in our way of describing systems. It would have been preferable to have notational systems constructed around techniques of analysis or synthesis (i.e., simulation languages). But our immediate need was for adequate descriptive power to present computer systems for a text. Considering the amount of effort it has taken to make these notational systems reasonably polished, it seems to us they should be presented to the computer profession, for criticism and reaction.

The main sources of experience with the notation so far is in the aforementioned book, where we have developed PMS diagrams for 22 systems* and ISP descriptions for 14 systems.** The levels of details in all of these is as adequate as the programming manual, i.e., as complete as the description of the PDP-8 example given here. In addition at least one new machine, the DEC PDP-11 (these proceedings), has made use of the notation at the formulation and design stage.

## REFERENCES


1 C G BELL A NEWELL
*Computer structures: Readings and examples*
In Press McGraw-Hill Company 1970

2 Y CHU
*Digital computer design fundamentals*
McGraw-Hill Book Company 1962

3 J A DARRINGER
*The description, simulation, and automatic implementation of digital computer processors*
Thesis for Doctor of Philosophy degree College of Engineering and Science Department of Electrical Engineering Carnegie-Mellon University Pittsburgh Pennsylvania May 1969

4 A D FALKOFF K E IVERSON E H SUSSENGUTH
*A formal description of system/360*
IBM Systems Journal Vol 3 No 3 pp 198–261 1956

5 T B STEEL JR
*A first version of UNCOL*
Proceedings WJCC pp 371–377 1961


* ARPA network; Burroughs B5500, B6500; CDC 6600; LGP 30; ComLogNet; DEC LINC-8-338, PDP-11; English Electric Deuce, KDF-9; IBM 1800, 7401, 7094, System/360 (Models 30 ~ 91), ASP network; LRL network; MIT's Whirlwind I; NBS'S Pilot; RW 40, SDS 910 930; UNIVAC 1108.

** The computers and the associated number of description pages (enclosed in parentheses) are CDC 160A (2), 6600 PPU (2), 6600 CPU (4¼); DEC PDP-8 (2, 3 with options), PDP-11 (5), 338 (5), IBM 1800 (3½), 1401 (3½), 7094 CPU (7), 7094 Data Channel (6½); LINC (~3); RW 40 (2½); SDS 92 (~3), 930 (4).