ULD AND A DESCRIPTION OF THE PDP-8

by
John A. N. Lee

Computer and Information Science
University of Massachusetts
Amherst, MA 01002

Technical Report 73A-1

September 1, 1973

A paper presented at
the SIGARCH/TCCA Workshop on
Computer Description Languages
at
Rutgers University
September 1973

ULD AND
A DESCRIPTION OF THE PDP-8.

BY

JOHN A. N. LEE

COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF MASSACHUSETTS
AMHERST, MA. 01002

SEPTEMBER 1, 1973

Manuscript Documentation Unit

Key Words

Formal description, formal definition, processor, mini-computer, PL/I, PDP-8, abstract machine, model, abstract syntax, ULD, Vienna Definition Language

CR Numbers

4.1, 4.10, 5.2, 5.23, 6.1

Abstract

A formal description of a Digital Equipment Corporation PDP-8-like computer is presented in terms of the definitional scheme which has previously been used for the description of the programming language PL/I. This description serves not only as an example of the universality of the definitional system and as a formal description of a machine from the user's point of view, but also as a possible basis for the formal design of the logical components of a digital processor. The author draws parallels between the components of the definitional system and the components of the machine being modelled as examples of the potential of this system as a design scheme.

Introduction

This paper gives an example of a means of formal description of a mini-computer based on the techniques developed by Lucas *et al.* [1] for the description of programming languages. For this purpose the Digital Equipment Corporation PDP-8 was chosen as the vehicle for formal definition as described in the "Small Computer Handbook."[2]

This definition has been developed from the point of view of a user of the machine so that accurate descriptions are given of the results of executing PDP-8 commands, particularly with respect to the contents of memory and the various registers.

It should not be concluded that this lack of "model closeness" indicates a shortcoming of the definitional system. During the development of this definition, two omissions from the verbal description in reference (2) were noted. Neither of these would seriously affect the concept of the machine from the point of view of a programmer but would have shown a designer (before the PDP-8 was built) that additional circuitry was required. In one case, the omission was the fault of the manual; and, in the other, the programmer should be aware of a singularly peculiar effect which could occur when an instruction is executed from the last word position in each page. The complete definition of the PDP-8-like computer is contained in reference (4).

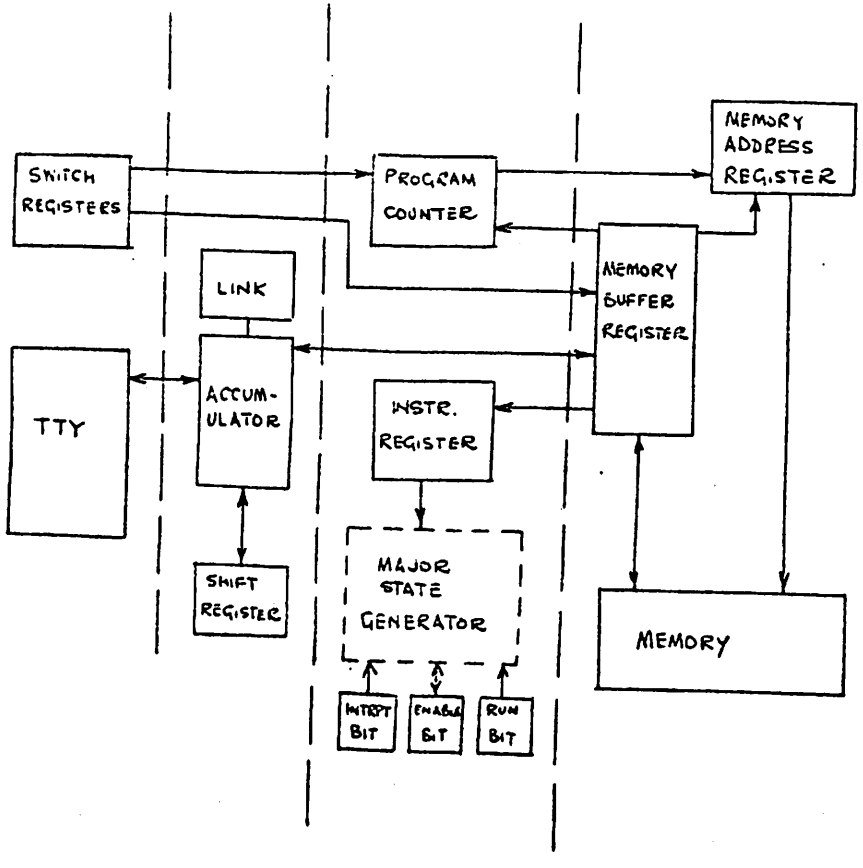
Method of Definition

The definitional system employed herein is based on that developed by IBM Laboratory Vienna and as used in connection with the formal

description of PL/I.[1] In this case, however (as compared to the formal description of an artificial language), there does not exist a source language for which is needed a concrete text (and hence a concrete syntax) and it is not necessary to consider the translation of such a language into an abstract text. It is assumed that the PDP-8 program being interpreted exists in the memory of the machine, that there exists a means for loading the program counter with the address of the first instruction to be executed and a means to initiate interpretation of the program at that address.

The Representation of the Machine

The definition system which is utilized here can be thought of as representing an abstract machine which is composed of two parts: an outer machine which contains the program to be executed together with the appropriate data, and an inner machine which acts as an interpreter. Within the inner machine will be found a *control stack* in which are lodged those instructions of the interpreter which are waiting to be executed, the definitions of those instructions, and a state transition function which controls and directs the execution of the interpreter's instructions. All objects in this abstract machine are represented by tree-like structures which are finite constructs and whose branches are named by *selector* functions. In the case of objects in the outer machine (the representation of the program and data set), the nodes of the tree-like structures do not "contain" any value but merely form a point from which other branches emanate. On the other hand, the leaves of the tree do contain elementary objects.



BLOCK DIAGRAM OF THE PDP-8-LIKE MACHINE

Fig. 1

Since a generalized definition scheme must be applicable to all possible instances of programs and data sets, a system is established by which the acceptable form of programs and data may be either formed or recognized. In the case of the PDP-8-like machine, the outer machine is described as a single object which is the *state* of the machine and which symbolically is represented by ξ Within this object exist the representations of the various registers of the machine being modelled. The names of the branches of the tree ξ are pointers (selectors) to the various components of the system being modelled as shown in Table 1:

Abstract Machine Selector	Actual Machine Element
s-tto	teletype output buffer
s-tti	keyboard input buffer
s-kbf	keyboard flag
s-tpf	teleprinter flag
s-enable	enable bit
s-int	interrupt bit
s-run	run bit
s-switch	switch register
s-link	link register
s-ac	accumulator
s-mb	memory buffer register
s-mar	memory address register
s-pc	program counter
s-ir	instruction register
s-mem	memory
s-shift	shift register

Table 1: Selectors and their corresponding machine elements.

The application of any one of these functions to the state of the machine will yield the *contents* of the selected register. In the description of the state ξ, which is given in the abstract syntax (that is, the syntax of the object which represents the outer machine), the syntax or predicate applicable to each register is specified. For example, the link register is specified by the manufacturer to be composed of a single bit register; thus the object selected from the state by the function *s-link* is specified to conform to the predicate *is-bit*. Examining the definition of *is-*ξ further, it may be seen that each pair of the form

<s-xxx:is-yyy>

is the description of the subtree which emanates from the root of the object ξ; *s-xxx* is the name of the branch, and *is-yyy* is the description of the object at the end of the branch.

These component objects may themselves be composite, and thus are represented by tree structures. For example, the set of twelve bit registers (switch register, accumulator, memory buffer and memory address register) are represented by an object which conforms to the predicate *is-word*. We find that an object which conforms to the predicate *is-word* is a structure composed of branches named *bit(i)* where i takes the values (in binary) from *0* to *1011*. The function *bit* may be regarded as a bit selector within the register and i is an argument which specifies the particular bit. By convention, the set notation

$$(f(i) \mid x \leq i \leq y)$$

is taken to mean the set of all $f(i)$ while i takes all possible integer

values between and including x and y. That is, for example, the set expression $(<bit(j):is\text{-}bit> \mid 0 \leq j \leq 10)$ is a shorthand for

$$(<bit(0):is\text{-}bit>\ ,\ <bit(j):is\text{-}bit>\ ,\ <bit(10):is\text{-}bit>)$$

Thus, a twelve bit register is represented by a tree with twelve branches named *bit(i)* which terminate in objects which conform to the predicate *is-bit*.

In the earlier work[†] on this definitional system which is used here, the authors of reference (1) used the standard set notation to define the contents of lists. Thus, $<(x \mid 10 \leq x \leq 17)>$ is equivalent to the list <10,11,12,13,14,15,16,17>. In reference (1) this notation was altered to

$$\text{LIST}\ f(i)\ \begin{matrix} b \\ i=a \end{matrix}$$

presumably so as not to lose the essential unordered nature of sets. The former notation is used in this paper to maintain conformity with earlier work.

In the definition of this machine, the only elementary objects which can exist are those which conform to the predicate *is-bit*, which itself may be described by the predicate expression

$$is\text{-}bit = 0 \vee 1.$$

That is, the objects which satisfy the predicate *is-bit* are 0 or 1. One other elementary object could exist in the machine, though this particular definition does not make use of its existence. That is, the

---

[†]See reference (5) in (1).

null object. By formal definition, the null object is that object which is generated when a selector function is applied to an object to which it is not applicable. For example, consider the description of any register of twelve bits. The only selector functions which are applicable to those registers are those specified in the predicate *is-word*. Thus if the selector *bit(* $\Pi 00$*)* were to be applied to one of the objects which represent these twelve bit registers, the result would be the null object, since no such branch exists and thus no object exists at the end of the branch.

Apart from the twelve bit registers, the other registers have organizations peculiar to their usage. For example, the input/output buffers are described as conforming to the predicate *is-byte*, which will be found to be the description of an eight bit structure. The instruction register, selected from the state by the selector function *s-ir*, is a three bit register since the operation codes in the machine are three bits in length. The program counter (*s-pc*(ξ)) is described here as being composed of two other structures which conform to the predicates *is-page-address* and *is-word-address*, respectively. A page address in this machine is composed of a five bit field while the word address (within a page) is a seven bit address. This model of the program counter is not exactly as constructed in the prototype, where the program counter is simply a twelve bit register. This division in the model mirrors the logical construction and usage of the program counter in the prototype rather than its actual construction. For example, in memory referencing instructions, the address of the operand is represented by a word

address (seven bits) and a one bit flag which indicates whether page 00000 or the current page is being referenced. The current page is that in which the instruction being executed is contained, that page being recorded in the program counter. Thus it is a common machine action to select the page address from the program counter, and thus the model was designed to reflect this common action.

The shift register (selected from the state by the selector function *s-shift*) does not appear in the list of registers in the "Small Computer Handbook" but was included here to facilitate the definition of shifting. Whether this register actually exists in the machine or whether the description here exactly models a shift register is not considered, since the definition is from a user's point of view rather than that of the designer.

The *s-mem*-component of the state represents the core memory of the PDP-8-like machine, being represented by a sequence of objects which conform to the predicate *is-page*, which in turn are represented by objects which are a sequence of components of the form is-word. Careful examination of the indices of the selector functions *s-page* and *s-word* will reveal that the memory is composed of 40_8 pages each composed of 200_8 words.

The overall structure of the machine as represented by the state is shown in Figure 2.

The original description of the definitional system[1] describe a special form of an object which represents a list and which conforms to the predicate *is-list*. This object is constructed so that the indices

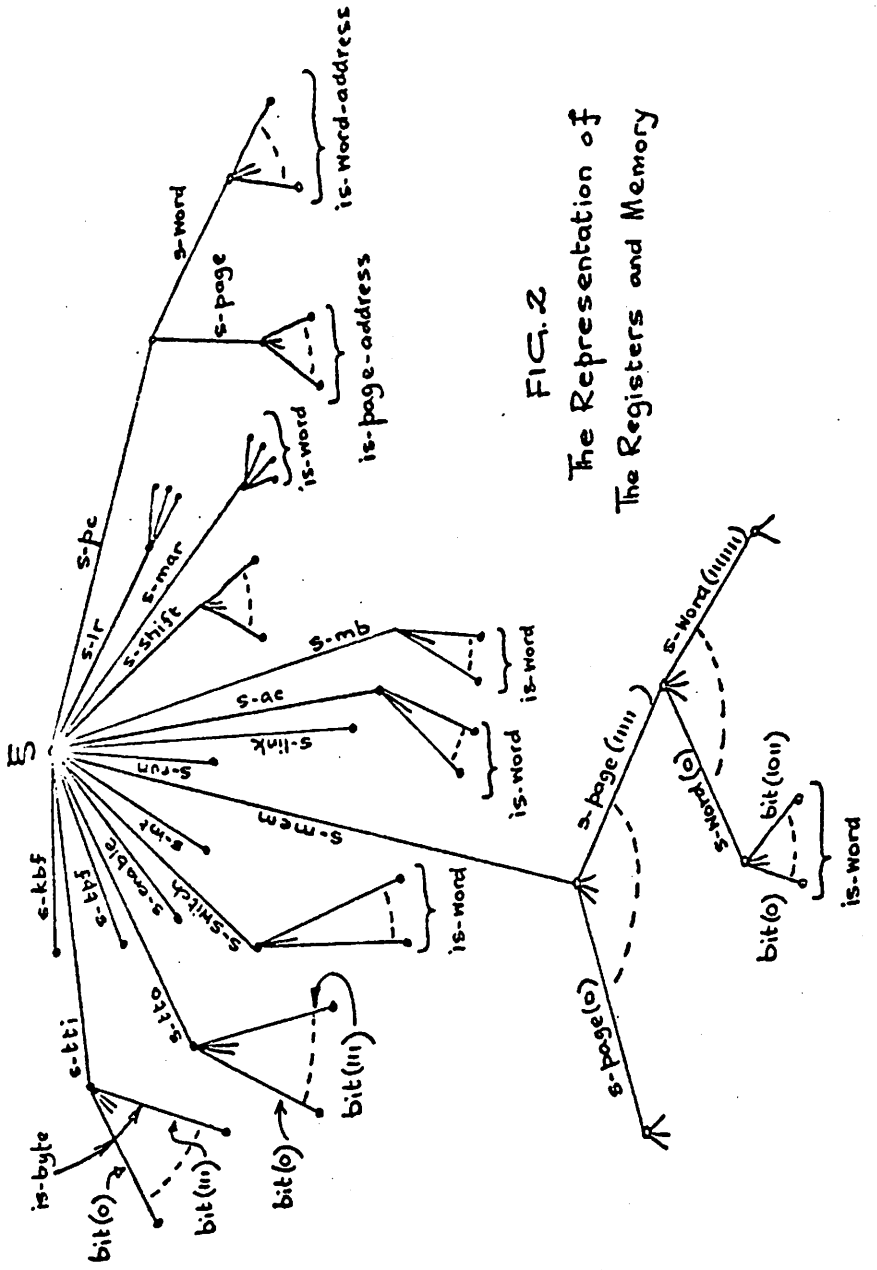


FIG. 2
The Representation of
The Registers and Memory

of the selectors *elem* are continuous (that is, there are no "holes" in a list) and the first (head) element is numbered one (1). Over this type of structure there are defined a number of list operations such as selecting the head element, the tail of the list and concatenation of two lists, so that in each case the appropriate predicate is satisfied. For example, the definition of concatenation (symbolized by the operation symbol ^) is defined so that the result conforms to the predicate is-list. Though not specifically described by Lucas *et al.*,[1] the indices of elements of a list are decimal representations, whereas in this description of the PDP-8-like machine, all references to any component of an object is expected to be in binary notation.

Obviously, if the index *i* represents a value, it is immaterial which numeric notation is used to represent that value. In this definition, however, all references will be in binary notation. To overcome the problem of a zero based reference system as compared with a one based reference system for lists in reference (1), but to take advantage of the operations over lists defined in (1), a functional equivalence relationship is defined $bit(i) \equiv elem(i+1)$.

In the same manner as $elem(j)(A)$ may be written as $elem(j,A)$, so we shall permit the form $bit(i,S)$ as being equal to $bit(i)(S)$.

Using this notation, it may be seen that the objects which represent the teleprinter buffer ($s\text{-}tto(\xi)$), keyboard buffer ($s\text{-}tti(\xi)$), switch register ($s\text{-}switch(\xi)$), accumulator ($s\text{-}ac(\xi)$), memory buffer register ($s\text{-}mb(\xi)$), memory address register ($s\text{-}mar(\xi)$), instruction register ($s\text{-}ir(\xi)$), shift register ($s\text{-}shift(\xi)$), the components of the

program counter (*s-page's-pc*(ξ) and *s-word's-pc*(ξ)),[†] and each word in the memory (*s-word(i)'s-page(j)'s-mem*(ξ)) are in conformance with the predicate *is-list*, and thus are susceptible to list operations such as concatenation (^), *head* and *tail*.

Although an actual object which is a list should be represented by a list of selector/object pairs in which each selector is of the general form *elem(i)*, it is common practice to represent a list by the notation <a,b,c,...,z> where the ordering of the objects is as specified in the written list. Thus, a twelve bit word containing all zeros would be written as

<0,0,0,0,0,0,0,0,0,0,0,0>

In this notation, the selector *bit(0)* and *elem(1)* would select the same component object.

The Interpreter

The *inner machine* of the definition consists of two structures and a state transition function, which together constitute an interpreter over the *outer machine* which represents the prototype.

The first structure is a set of instruction definitions which are analogous to the logical compositions of instructions in the control unit of an actual machine. Instructions are "executed" from the second structure, which is a tree-like object containing instructions at the nodes and leaves of the tree. Only those instructions which are contained

[†]By definition in (1), the successive application of selector functions to an object such as *s-a* ◦ *s-b* (*s-c*(A))) is written as *s-a*'*s-b*'*s-c*(A), where the composition *s-a*'*s-b*'*s-c* is a composite selector and may be thought of as a "path" through the tree.

in the leaves of the tree can be executed. The execution of an instruction may result in either its removal from this control tree (after it has affected some modifications to the state ξ) or its replacement by another structure which contains instructions. Thus instructions may be classified into one of two groups: macro-expansion instructions or value returning instructions. In effect, these two types of instructions are similar to the augmented instructions and memory referencing instructions of the PDP-8 computer. That is, the macro-expansion instructions are defined to be a composition of other (micro) instructions, while the value returning instructions directly affect the state of the machine in a manner similar to that of the memory referencing instructions.

The definition of an instruction takes the general form

$$\underline{inst\text{-}name}(P_1, P_2, \ldots, P_n) \equiv$$
$$t_1 \rightarrow group_1$$
$$t_2 \rightarrow group_2$$
$$\ldots$$
$$\ldots$$
$$t_m \rightarrow group_m$$

where $group_i$ describes the action to be taken when the value of t_i is true, where t_i is any predicate expression over the state of the machine and the evaluated parameters $P_1, \ldots, P_n$. Together, the pairs $t_i \rightarrow group_i$ are the components of a conditional expression[(3)], which is defined such that the value of the expression is that $group_j$ whose corresponding predicate expression t_j is true and all predicate expressions t_i (for i less than j) are false, irrespective of the values

of the predicate expressions t_k for k greater than j. If there is no t_j which is true then the value of the predicate expression is undefined. For example, a conditional expression can be used in other definitional systems to describe (say) the operation of evaluating the *signum* function:

$$signum(x) = (x<0 \rightarrow -1,\ x>0 \rightarrow +1,\ x = 0 \rightarrow 0)$$

where the ordering of predicate expression/value pairs is from left to right, being separated by commas. In this example, no two predicate expressions can be true at the same time, and thus the problem of which value is chosen as the value of the function is simplified. However, cases can arise in which the ordering of the pairs is extremely important. Commonly, the last predicate expression in a conditional expression should correspond to the term "otherwise." That is, no matter what conditions hold, other than those already sieved out by the preceding predicate expressions, the value of the conditional expression shall be that associated with the "otherwise" predicate expression. By using the tautology T (that is, true), this "otherwise" conditional may be developed. Thus the definition of the signum function above may be amended to

$$signum(x) = (x<0 \rightarrow -1,\ x>0 \rightarrow +1,\ T \rightarrow 0)$$

since, after sieving out the conditions of less than or greater than zero, the only possible other case is equality with zero. The only possible disadvantage of using this "catch-all" or default condition is that there might exist the possibility that the parameter x is not numeric. For example, if for some reason x is currently assigned the value of an alphabetic character, then in the latter definition of the *signum* function the value to be assigned to the function would

be 0, whereas in the first definition the function would be undefined. In the conditional expressions used in the definition of the PDP-8-like machine, it is known that there is a highly restrictive set of objects in the *outer machine*, and thus the possibility of an error in definition due to the use of the *otherwise* term is zero, provided that the *outer machine* conforms to the predicates described in the abstract syntax.

Within the definition of an instruction, individual groups are not constrained to be of the same type throughout. Thus under certain conditions an instruction may be a macro-expansion instruction while under other conditions it is value returning.

A value returning definition group takes the general form

PASS: exp

$\underline{s\text{-}sc_1}$: exp_1

$\underline{s\text{-}sc_2}$: exp_2

...

...

$\underline{s\text{-}sc_r}$: exp_r

where the $\underline{s\text{-}sc_i}$ are selectors over the state and exp_i are value expressions. The line preceded by the term *PASS:* is used in the definitional system to define a value which is to be "passed" to the argument list of another instruction in the control part of the inner machine but is not used in the definition of the PDP-8-like machine. The result of executing an instruction as a value returning group is that the components of the state ξ selected by the selector functions $\underline{s\text{-}sc_i}$

are replaced by the values of the expressions exp_i, and the instruction is removed from the control part of the inner machine. Thus, for example, in the definition of the instruction

clear-mb

which is the model of the operation of clearing the memory buffer register

the *s-mb*-component of the state is set to

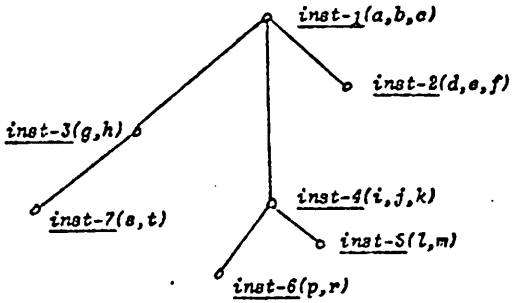
<0,0,0,0,0,0,0,0,0,0,0,0> ,

that is, a list of twelve zeros.

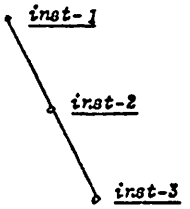
The original description of the definitional method (1) restricted the selectors $s\text{-}sc_i$ to be simple selectors (that is, non-composite selectors) over the state of the machine. This was found to be too restrictive in this case where it was necessary to operate over (say) a word in the memory of the machine being modelled. It would have been possible to have constructed a model of the PDP-8-like machine so that each memory word would have been directly accessible from the state, but this modification would have sacrificed "model-closeness," which was felt to be of some importance. Further, the author could find no logical reason for this restriction, particularly when the original definers took so much trouble to create "simple objects" which represented composite selectors to be used in the value returning definitions of the PL/I machine. Thus in the definition of the PDP-8-like machine, the selectors $s\text{-}sc_i$ of the value returning groups are permitted to be composite selectors. For example, in the definition of a parallel shift of operation, in which the bits of the accumulator are transferred to the shift register, the composite selectors *bit(i)'s-shift*

denote the bit position in the shift register which are to receive the bits from the accumulator (and link).

Macro-expansion definition groups are descriptions of a structure of instructions which are to replace the instruction currently being "executed." While such structures could be represented in a written definition by diagrams, such as:



A system of writing involving indentation and punctuation is used to implicitly describe these structures. For example, a *sequential* system having the diagrammatic structure of a single branch tree, such as:



is written so that each level is indented from the level above and each

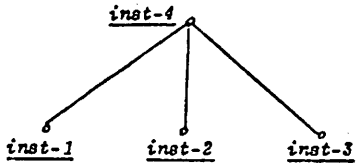
level is terminated by a semicolon - except for the lowest level. Thus, the obove structure would be symbolized by

inst-1;

inst-2;

inst-3

Conversely, a structure having several instructions at the same level such as:



is symbolized by a vertical list, without indentation, separated by commas - except for the last instruction, which then is the end of a level and may be followed by a semicolon. The above structure would be represented by:

inst-4;

inst-1,

inst-2,

inst-3

While complex structures such as that shown on the previous page may be represented in this manner, the definition of the PDP-8-like machine given here does not require such complex representations.

Within the control part of the state of the machine (the inner machine), instructions which are leaves of the tree may be executed in any order. In particular, value returning instructions, after execution, are removed from the tree and thus reveal other instructions which then may be a candidate for execution since they now lie on a leaf of the tree. However, in macro-expansion execution of an instruction, that instruction is replaced by at least one other instruction, and thus its execution does not reveal other instructions in the tree but, rather, replaces itself as a candidate for execution.

The instruction *null* is a system defined instruction which is always a macro-expansion instruction, its execution being simply achieved by its own deletion, with no replacement, from the control tree. This instruction is basically a "do nothing" command, though it is prescribed *in this definition* that the "cycle time" of the *null* instruction is the same as that of any other instruction, thus maintaining the synchronization of the instructions which are executed simultaneously.

In certain instances, there exist a class of instructions which are similar in definition except for a single character or numeric value. These have been defined by a single definition in which the general form is shown. For example, there exists a class of instructions of the form *and-i* which are *ac-and-mb* (representing the bitwise and operation over the accumulator and the memory buffer register). It is not intended that part of the instruction name be a parameter but simply the definition is the general form of these instructions. Thus, the actual definition of *and-101* is

and-101 =

bit(101,s-ac(ξ)) → 1 → bit(101)'s-ac:bit(101,s-mb)

T → null

A similar scheme is used in the definition for 12-bit addition.

Departing from the concept of instruction execution from the control part of the state of the machine whereby "instructions associated with the terminal nodes of the control-tree are candidates for being executed next" and where the order of execution is immaterial, we have chosen in this description to insist that all candidates for execution are interpreted simultaneously. By this means we simulate the parallel actions occurring within the processor. Further, we shall insist that the "time" to execute all instructions which are being executed simultaneously is the same. In certain instances, dummy instructions have been added to instruction definitions to provide a delay and thus to synchronize subsequent executions.

With respect to the definitional scheme, severe restrictions were placed on the types of instructions to be used and, in particular, on the passing of arguments between instructions in the control part of the state of the machine. To maintain model closeness, it was decided to insist that all data necessary for the interpretation of a definition instruction should be selected directly from the state rather than being passed through an argument list and that the components of the state should correspond directly with the registers in the machine. It is argued that where it is necessary to provide an argument to an instruction there must exist a register which is not described in the

manual or which is necessary to the means of implementation described here. Secondly, in so far as possible, instructions in the description were designed to correspond with actions to be taken within the processor and thus may be considered to describe the actions of logical units within the major state generator.

The actual "execution" of instructions in the inner machine is under the control of a state transition function, which selects the instructions to be executed, monitors the execution and creation of a new state, and initiates the next cycle. The new state contains not only the modified "outer machine" but also the reorganized control part of the "inner machine." Because the state transition function continually creates new states, there is no timing problem in the definition of an exchange as might occur in a real computer program. For example, the value returning group

bit(i)'s-ac:s-link(ξ)

s-link:bit(i)'s-ac(ξ)

references the old state on the right hand side of each colon for the source of the values and the new state on the left hand side for the placement of those values. Thus the two replacements described in the definition of the serial addition process are not out of synchronization but merely refer to the old state for their data. That is, the functions *and* and *eor* operate over the i-th bits in the accumulator and memory buffer register in the old state and are the values to be placed into the i-th and $(i + 1)$-th bits of the accumulator and memory buffer register of the new state respectively.

Besides instructions which define the interpretation of the program which is contained in the outer machine, there also exist functions which define either special names given to selectors, such as *bit(i)* or *head*, the attribute of an object, such as *word-length*, or an operation over the components of the system in terms of simpler or fundamental operations. In general, functions are defined as conditional expressions, though in the list of definitions later will be found equivalences of function names also. That is, *bit(i)* is equivalent to *elem(i+1)* and *head* is equivalent to *bit(0)*.

An important function is *convert*. This function transforms a list of bits (conforming to the predicate *is-bit-list*) into a single numeric value. This value is used as an index in addressing words of memory

or as the "value" of the operation code in the instruction register during the interpretation of the instruction in definition of *execute*. It is important to note that since all computation is being performed in binary, the definition of the function convert uses a position value of 10 rather than (decimal) 2.

The definitions relating to the logical operations *or*, *eor* (exclusive or), *and* and *not* are based on the premise that a bit value of *1* indicates truth while *0* indicates false.

The problem of defining the source of an interrupt pulse has been left unsolved, it being assumed that there exists some other (abstract) machine which provides an interrupt pulse to the state of this machine.

The means for transferring this data from the state of the external machine to the state of this machine is not considered.

In order to identify the major states of the PDP-8, we have chosen to define the initiation of the fetch cycle as the state (in the abstract machine) in which the *fetch* instruction is the candidate for execution, and similarly for the defer (*defer*) and execute (*execute*) cycles.

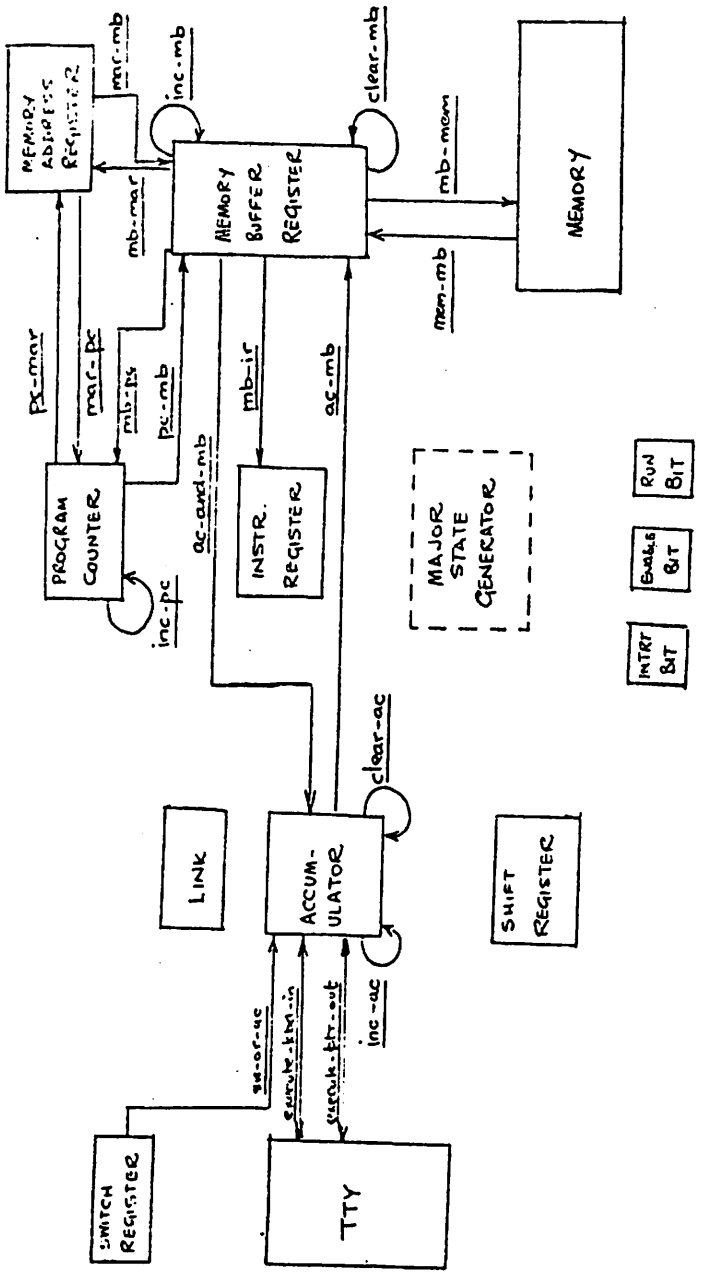


Fig. 3. SOME RELATIONS BETWEEN REGISTERS AND INSTRUCTIONS

Summary

By the definitional system which has been described and used here, a means of describing the logic of a processor design can be evolved. By taking the analogy of an "outer" and "inner machine" one more step, it may be seen that there exists a correspondence between data paths in the machine being modelled and certain instructions (for example, *mb-mar* which symbolizes the data path between the memory buffer register and the memory address register), between registers and the immediate components of the state ξ, (for example, between $s\text{-}ir(\xi)$ and the instruction register) and operations over the registers and certain instructions (for example, the PDP-8 instruction OPR and its embedded microinstructions, and the inner machine instruction *execute-opr*). Further, with the restriction that all instructions be without formal arguments, it can be claimed that the inability to meet this restriction shows that there is a need for the creation of another register within the state to hold the data required for this instruction. For example, in the description of incrementation here, it is necessary to have the argument whose value is the size (word-length) of the register being incremented. This value should be contained in some "count register" which is associated with the incrementation logic of the machine being modelled.

There is a direct relationship between macro-expansion instructions and the augmented instructions OPR and IOT in the PDP-8-like machine, up to the point at which the microinstructions have been identified and are ready for execution. At this point there is a relationship between the operations over registers (including memory) and value returning instructions.

Slight variations on the original definitional method have been used here, the most important being that of parallel execution of candidates for execution. In the original system, the order of execution was unspecified but sequential. There does exist an unanswered question of the possibility of side effects which will result in an ambiguous definition due to the simultaneous action of instructions on the state of the "outer machine." There exists the possibility, which was followed here, merely because the PDP-8-like machine was designed in this manner, that no two instructions being executed simultaneously be allowed to modify the same component of the state. The definition of the PDP-8-like machine,[4] for example, does not permit the simultaneous left and right shifts of the accumulator that would appear to be possible according to the reference (2). This would appear to be both logically, and practically, improper.

The conditions under which simultaneous execution of instructions would be possible are the subject of another paper, but suffice it to mention here that it is the author's contention that these conditions are similar to those imposed on the mutation operator (μ) of the definition system (see reference 1) where several selector/object pairs are to be replaced in the state; that is, the ordering of successive

mutations shall not influence the creation of the new object. Where such an ordering is an influence on the object created, then the operation of mutation is undefined. The conditions under which the operation of mutation is undefined are easily derived, and it is expected that similar conditions can be determined for the case of simultaneous execution of instructions.

This paper has not included any examples of the use of this definition of a PDP-8-like machine since the definitions of the various "outer machine" instructions are mirrored in the instructions of the "inner machine." For example, questions relating to the instruction DCA (operation code 3_8) may be answered by reference to the "inner machine" instruction *execute-dca*. Similarly, an explanation of the actions of the fetch cycle are explained in the instruction *fetch*. By design, there is a one-to-one correspondence between the instructions in the outer machine and those named in the inner machine. Those instructions of the inner machine which relate to the data paths in the machine being modelled together with some of the "housekeeping" instructions are shown in Figure 3. In general, the abbreviation *int* means interpret, *inc* stands for increment, *wd* for word, and *pg* for page. Elsewhere instruction names can be read literally, with the exception of the group of the form *opri-et-j* which imply the operation OPR at event time (et) j in microinstruction group i.

References

(1) Lucas, P., and Walk, K. "On the Formal Description of PL/I," Annual Review on Automatic Programming, Vol. 6, No. 3, 1969. Pergamon Press, New York.

(2) _____. "The Small Computer Handbook." Digital Equipment Corp., Maynard, Mass.

(3) McCarthy, J. "A Basis for a Mathematical Theory of Computation," in Braffort, P., and Hirschberg, D. (eds.), Studies in Logic, "Computer Programming and Formal Systems," North-Holland Publishing Co., Amsterdam, 1963.

(4) Lee, J. A. N., COMPUTER SEMANTICS, Van Nostrand Reinhold Pub. Co., New York, 1972.

COMPUTER AND INFORMATION SCIENCE

TECHNICAL REPORTS

The following TECHNICAL REPORTS are now available at the Computer and Information Science Department, Graduate Research Center.

70C-2 Organizational Principles for Embryological and Neurophysiological Processes by Michael A. Arbib.

70C-3 On the Likely Evolution of Communicating Intelligence on Other Planets by Michael A. Arbib (August 1, 1970, revised December 30, 1970.

70C-4 Contextual Error Detection by Roger W. Ehrich and Edward M. Riseman.

70C-5 Transformations and Somatotopy in Perceiving Systems by Michael A. Arbib.

70C-6 Organizational Principles for Theoretical Neurophysiology by Michael A. Arbib, (August 15, 1971).

71B-1 The Definition and Validation of the Radix Sorting Technique by John A. N. Lee, (January, 1972).

71B-2 Two Papers on Group Machines by Michael A. Arbib.

71B-4 Automata with Ranked State Sets by Dieter Schütt.

71C-6 Machines in a Category by M. A. Arbib and E. G. Manes.

72A-1 A Study of the Constraints upon the Parallel Dispatching and Execution of Machine Code Instructions by Caxton C. Foster and Edward M. Riseman.

72B-1 The Formal Definition of the Basic Language by John A. N. Lee.

72B-2 Decomposable Machines and Simple Recursion by Michael A. Arbib and E. Manes.

72C-1 A Contextual Postprocessing System for Error Detection and Correction in Character Recognition by Edward M. Riseman and A. Hanson (October 1972).

73B-1 Adjoint Machines, State-Behavior Machines, and Duality by Michael A. Arbib and Ernest G. Manes (January 1973).

73B-2 Natural State Transformations by Suad Alagić (February 1973).

73C-3 A Model of Posited Decisionary and Learning Mechanisms in Mammalian CA-3 Hippocampus by William Kilmer, T. McLardy, and M. Olinski (February 1973).

73C-4 Four Faces of Hal: Using Artificial Intelligence Techniques in Computer-Assisted Instruction by Howard A. Peelle and Edward M. Riseman (March, 1973).

73C-5 System Design of an Integrated Pattern Recognition System, or How to Get the Best Mileage out of your Used Pattern Classifier by A.R. Hanson and E.M. Riseman, (June 1973).

73C-6 Neural Models of Spatial Perception and the Control of Movement, by Michael A. Arbib, C. Curt Boylls & Parvati Dev (June 1973).

73B-3 Foundations of System Theory, I by Michael A. Arbib and Ernest G. Manes, (July, 1973.)

73A-1 ULD and a Description of the PDP-8, by John A. N. Lee (September 1973).

COMPUTER AND INFORMATION SCIENCE

TECHNICAL NOTES

The following TECHNICAL NOTES are now available at the Computer and Information Science Department, Graduate Research Center.

TN/CS/00001 (Out of Date) — Current Research Toward the Standardization and Formal Definition of PL/I by John A.N. Lee (April 1,1968).

TN/CS/00002 — Discrete Markov Chains: An Heuristic Approach by Sue N. Stidham (July 1,1968).

TN/CS/00003 — The Domelki Syntactic Analysis Algorithm by Susan L. Gerhart (August 28,1968).

TN/CS/00004 — A Survey of Hashing Techniques by John A.N. Lee (September 15,1968).

TN/CS/00005 — The Recognition and Use of Null Elements in a Syntax Directed Translator by John A.N. Lee (September 19,1968).

TN/CS/00006 (Revision is TN/CS/00020) — SYNFUL, A Proposed General Purpose Translator System by John A.N. Lee (October 1,1968).

TN/CS/00007 — Sorting Almost Ordered Arrays by Caxton C. Foster (November 18,1968).

TN/CS/00008 (Out of Date) — Vienna Definition Language--Semantics by John A.N. Lee (December 13,1968).

TN/CS/00009 — An Examination of Two Hash Transforms by Caxton C. Foster (May 9,1969).

TN/CS/00010 — Multiplexing Without Tears by Caxton C. Foster (May 30,1969).

TN/CS/00011 (Out of Date) — Vienna Definition Language--A Generalization of Instruction Definitions by John A.N. Lee and Delmore Wu (April,1969).

TN/CS/00012 — An Unclever Time-Sharing System by Caxton C. Foster (October,1969).

TN/CS/00013 — The Formal Definition of the Basic Language by John A.N. Lee (April,1970). (Also published in Computer Journal and as Technical Report 72B-1)

TN/CS/00014 — A Debugging Aid by Caxton C. Foster and Hugh C. Schulz (January 16,1970).

TN/CS/00015 — Conditional Interpretation of Operation Codes by Caxton C. Foster and Robert H. Gonter (February,1970).

TN/CS/00016 — Some Simple Algorithms for Content Addressable Memories by Caxton C. Foster (July,1970).

TN/CS/00017 — A Data Distributor--The Sprinkler System by Caxton C. Foster (July,1970).

TN/CS/00018 An Annotated Bibliography on Syntax-Directed Translation by John A.N. Lee, Taveta K. Bogert, and Helen Gigley (July,1970).

TN/CS/00019 Chargogagge,manchaugagogchaubunagungamaug--A Novel Multiply-by-Three Circuit by Caxton C. Foster, Edward Riseman, Fred Stockton, and Conrad Wogrin (September,1970).

TN/CS/00020 SYNFUL--A General Purpose Translator System and Extensive Modification of Technical Note #00006 by John A.N. Lee and Helen Gigley, edited by Taveta K. Bogert (November,1970).

TN/CS/00021 Maintenance Manual for the UMASS Timesharing Version of SYNFUL by Ronald Lautmann (November,1970).

TN/CS/00022 Microprogramming: A Design Alternative by Michael J. Sullivan (December,1970).

TN/CS/00023 A Simulated Associative Memory by Caxton C. Foster (December,1970).

TN/CS/00024 A Five Tape Algorithm for the Instant Playback Problem by Caxton C. Foster (December,1970).

TN/CS/00025 A Comparison of Simulation Languages by Albert W. Zukatis (January,1971).

TN/CS/00026 A Stack Oriented Computer by Caxton C. Foster (April,1971).

TN/CS/00027 Certain Formal Properties of the Vienna Definition Language by John A.N. Lee.

TN/CS/00028 When the Chips are Down by Caxton C. Foster (January,1972).

TN/CS/00029 RAUPEDATA-11 -- A Sophisticated Debugging Program for the PDP-11 by Edward G. Fisher (February,1972).

TN/CS/00030 A Tutorial on Cobol Extensions to Handle Data Bases: The Data Base Group Report by Robert W. Taylor (February,1972).

TN/CS/00031 System Design: Process Models by Richard H. Eckhouse, Jr. (April,1972).

TN/CS/00032 Automated Accounting Systems by Richard H. Eckhosue, Jr. (April,1972).

TN/CS/00033 A Generalization of AVL Trees by Caxton C. Foster (June,1972).

TN/CS/00034 Conditional Syntactic Specification by J. Dorocak and John A.N. Lee (September,1972).

TN/CS/00035 A Formal Definition of Mini Language Number 7,"Dynamic Type Checking" by Edward G. Fisher (October,1972).

TN/CS/00036 Data Administration, Data Independence, and the DBTG Report by Robert W. Taylor (October 1973).