CS 90

A MULTI-LEVEL COMPUTER ORGANIZATION DESIGNED TO SEPARATE DATA-ACCESS ING FROM THE COMPUTATION

BY

VICTOR R. LESSER

TECHNICAL REPORT NO. CS 90
March 11, 1968

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

CGTM 37
January, 1968

A Multi-Level Computer Organization Designed to Separate Data-Accessing From the Computation

by Victor R. Lesser*

ABSTRACT

The computer organization to be described in this paper has been developed to overcome the inflexibility of computers designed around a few fixed data structures, and only binary operations. This has been accomplished by separating the data-accessing procedures from the computational algorithm. By this separation, a new and different language may be used to express data-accessing procedures. The new language has been designed to allow the programmer to define the procedures for generating the names of the operands for each computation, and locating the value of an operand given its name.

* Supported in part by the National Science Foundation

CONTENTS

I. Introduction

In a conventional computer organization, procedures for accessing non-standard*[1] data structures are expressed in the same language (same instruction repertoire and instruction sequencing) as, and are intermixed with, the computational algorithm. If data-accessing can be decoupled from the computational algorithm, one can then implement them in different languages. This permits greater economy of representation and ease of programming for both the computation and data-accessing procedures. This is particularly true for sophisticated problems requiring data representations not anticipated by the computer designer.

Computational operations performed on these complex data structures usually involve multiple operands.*[2] Therefore, the accessing mechanism must be able to locate both single elements and ordered sets of elements contained in the data structure. This capability can be accomplished *through* two formalisms; one which generates a list of names of operands, and the other which locates the value of an operand given its name. Using the above concepts hierarchically, a list of operand values can be extracted from the data structure to be used as arguments for a computational operation. ,

In most computer organizations, the machine language instructions are usually binary or unary operations, and the names of the operand for instruction' are determined by the particular instruction format. In a single address computer, one operand name (an address) is contained in the instruction, and the other operand is implicitly defined as the accumulator; while in a computer with a stack mechanism, the operands are a few elements' at the top of the stack.

*[1] In most computers such commonly used data-structures as lists, matrices, and stacks are considered non-standard.

*[2] For example, most operations performed on matrices can be most easily represented as operation on'rows (columns) rather than on individual elements, of matrix,

The important point of these examples is that the programmer has been given no flexibility in defining the names of the operands for a computation except through the use of different instruction formats. The built-in mechanisms for locating the value of an operand, such as indirect address, indexing, and B5500 program reference table, are useful, but only for a small class of data' structures which are anticipated by the computer designer,

The computer organization to be described in this paper has been developed to overcome the inflexibility of computers designed around a few fixed data structures, and only binary operations. This has'been accomplished by separating the data-accessing procedures from the computational algorithm. By this separation, a new and different language may be used to express data-accessing procedures. The new language has been-designed to allow the programmer to define the procedures for generating the names of the operands for each computation, and locating the value of an operand given its name.

II* Data Accessing

The program for data-accessing has been separated from the computational instruction stream by integrating the function of data--accessing into the computer's memory organization. The key idea in this computer organization is the ability of each memory register to be more than just a place to hold a. value. Each memory register K (there are no special purpose registers) can be thought of as representing two different entities depending on how the register is accessed in the computation:

1) the name of an opernad K;
2) the name of a computation K.

If the register is accessed as the name of an operand, then we are interested in obtaining the value of the operand, while if the register is the name of the computation then we are interested in generating the names of the operands used by the computation. It is important to note the hierarchy: first we generate the names of the operands for the computation, and then we determine' the values of the operands" These two rfunctions of the register K are accomplished by auxiliary information attached to the register (in the implementation to be described later, this will be through a pointer) which defines the

first step in an algorithm for performing the desired task.

A. Value Generation

The mechanism (formalism) for describing how the value of an operand is located, given its name, will be discussed first. The mechanism for generating the names of the operands for a computation is more complex but it essentially is just an extension of the value generation mechanism. In locating the value of an operand, it is desired to generate an access path between the name of the operand and its value. This is accomplished by attaching to each register K the following auxiliary information:

1) f_k - the name of a function;
2) N_k - the name of a register whose value defines the number of primary levels of indirectness;
3) F_k - the name of a register whose value is used as one of the arguments for f_k.

Let us define C(K) as the contents of register K. If the name of an operand is considered to be an address of a memory register, then the value of operand K, V(K), is defined by the following recursive definition:

$$V(0)=0$$

$$V(1)=1 \quad .$$

$$V(K)=B(V(N_k),K)$$

$$B(M,K)=V(B(M-1,K))=V^M(B(0,K))$$

$$B(0,K)=T_{f_k}(K,C(K),V(C(K)),V(F_K))$$

-This recursive definition for the value of a register is a generalization and combination of the concepts of indirect addressing, and multi-leveling indexing. At each level in an indirect chain, a computation may be performed to determine the address of the next register in the chain, and the number of further levels of indirectness is a data parameter contained in the auxiliary information, N_k. In the normal implementation of indirect addressing either the number of levels of indirectness is fixed, or the length of 'chain is completely controlled by the indirect bits contained in the registers of the chained list. Both of these schemes do not allow operations such as retrieving the Ith element of a chained list where the number I may vary each time the'

chain is accessed. The value generation mechanism previously described allows the above operation, and much more sophisticated data accessing operations through the use of the parameter N_k. If N_k=0, then the recursive mechanism resembles an addressing schema with no indirection, while N_k=1 corresponds to the normal concept of indirect addressing. The computational capability represented by the above recursive definition for V(K) is very similar to the representation of an algebraic expression in Cheatham's pseudo code when N_k=0. Let us consider an example of the pseudo code resulting from the algebraic expression:

A + 25×(B+C)

Line	Operator	Operand Descriptors
1	D_+	D_A, D_2
2	$D_\times$	D_{25}, D_3
3	D_+	D_B, D_C

If we define the following functions for T:

$T_+(X,Y,Z,W)=Y+W$

$T_\times(X,Y,Z,W)=Y\times W$

then one possible representation of the above algebraic expression in the recursive structure defined for value generation is:

Register K	C(K)	f_K	N_K	F_K
2	Value of A	+	0	3
3	25	×	0	4
4	Value of B	+	0	address of C

Let us consider V(2):

$V(2)=B(V(N_2),2)=B(V(0),2)=B(0,2)$

$B(0,2)=T_{f_2}(2,C(2),V(C(2)),V(F_2))$

$=T_+(2,C(2),V(C(2)),V(3))$

$=C(2)+V(3)=$Value of A+V(3)

$V(3)=B(0,3)=T_\times(3,C(3),V(C(3)),V(4))$

$=C(3)+V(4)=25\times V(4)$

V(4)=Value of B+V(address of C)

if V(address of C)=Value of C

then V(2)=Value of A+(25×(Value of B+Value of C))

If we define two additional functions:

$T_{ZW+}(X,Y,Z,W)=Z+W$

$T_{ZW\times}(X,Y,Z,W)=Z\times W$

the computation of the algebraic expression could be represented in a different manner:

Register K	C(K)	f_K	W_K	F_K
2	address of A	ZW+	0	3
3	25	×	0	4
4	address of B	ZW×	0	address of C

if V(address of A)=Value of A then

V(2)=V(address of A)+V(3)

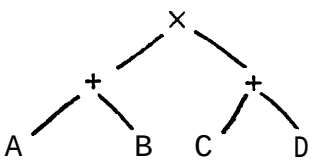
V(3)=25×V(4)

V(4)=V(address of B)+V(address of C)

therefore V(2)=A+25×(B+C)

These two formulations of the algebraic expression both have their advantages. The first method is faster since memory references to find the value of A and B are unnecessary, but by having the value of A and B interconnected with the computation there is no way to isolate the value of A unless the auxiliary information is altered. The second method does not have this problem since the value of A is not part of the computation but only an operand. This alternate format also allows us to exploit parallelism.* Let us consider the following computation (A+B)·(C+D). Then it can be represented by the following tree structure which is duplicated in the register configuration.

* This type of local parallelism introduced in computing the data-accessing mechanism is easily implemented since there is not the problem of side effects, since no registers are modified in the process.



Register K	C(K)	f_K	N_K	F_K
2	3	ZW×	0	4
3	address of A	ZW+	0	address of B
4	address of C	ZW+	0	address of D

The three examples which follow indicate how the computational capability represented by the above, combined together with the level of indirectness, can be used to locate elements in data structures. Before considering the examples, let us define the following additional functions:

$I_Y(X,Y,Z,W)=Y$

$T_{I_X}(X,Y,Z,W)=X$

$T_{Pl}(X,Y,Z,W)=Y+1$

EX 1: We would like register K to represent the matrix element [I, J]. Let us suppose the value of A [I, J] for particular I, J is the value of the following cell:

$V(BASE_A + I \times D_A + J)$

Let us define the following memory configuration:

Register L	C(L)	f_L	N_L	F_L
K	$BASE_A$	+	1	K1
K1	J	+	0	K2
K2	I	X	0	K3
K3	D_A	I_Y	0	

Then

$V(K) = B(V(1),K) = B(1,K)$

$B(1,K) = V(B(0,K))$

$B(0,K) = BASE_A+V(K1)$

$V(K1) = B(V(N_{K1}),K1)) = B(V(0),K1) = B(0,K1)$

$B(0,K1) = J+V(K2)$

$V(K2) = B(0,K2)$

$B(0,K2) = I \times V(K3)$

$V(K3) = B(0,K3) = D_A$

$V(K2) = I \times D_A$

$V(K1) = J + I \times D_A$

$B(0,K) = BASE, + J + I \times D_A$

$V(K) = V(BASE_A + J + I \times D_A)$

Ex 2: We would like register K to represent the Jth element of an ordered list. Each element of the list is represented by a consecutive pair of registers; the first register contains the address of the first register of the next pair, and the second register contains the value of the list element. The register configuration is more difficult than in (EX 1) since there are two separate actions which need to be performed:

1) getting the address of Jth element of the list;
2) using the address of Jth element to get its value: the value is the second element in the register pair.

Let us suppose that L = address of the first elements of the list, and each pointer element of the list contains the following auxiliary information: f=Iy, N=0

Let us define the following memory configuration:

Register P	C(P)	f_P	N_P	F_P
K	1	+	1	K1
K1	L	Iy	K2	
K2	J	Iy	0	

Then

$V(K) = B(1,K) = V(B(0,K))$

$B(0,K) = 1+V(K1)$

Note V(K1) = address of the Jth element of list

$V(K1) = V(B(V(K2),K1))$

$V(K2) = J$

$V(K1) = V(B(J,K1)) = V^J(B(0,K1))$

$B(0,K1) = L;\ V(K1) = V^J(L)$

but V(M) = C(M), for each M = the first element of list pair

$V(K1) = C^J(L) = \underbrace{CCC\ --\ C(L)}_{J \text{ times}}$

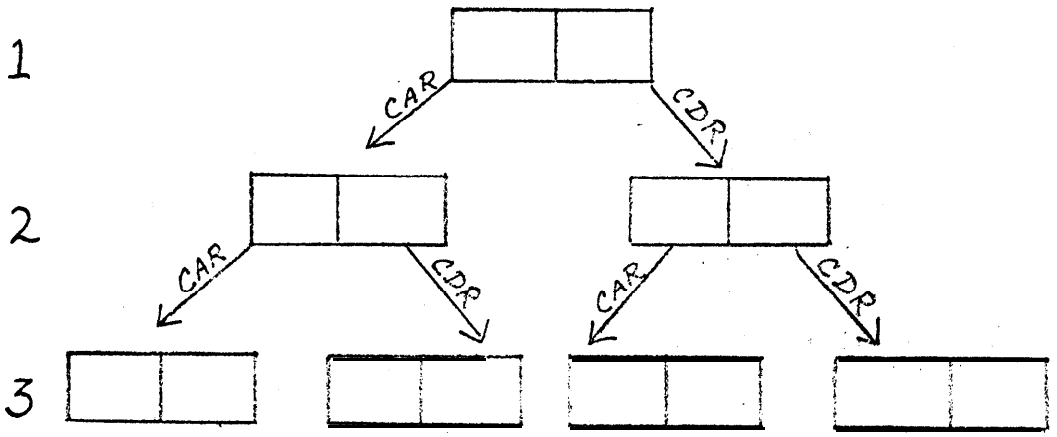
so we get that

$V(K) = V(1 + C^J(L))$

which is the value of the Jth element of list.

Notice that if the second register was not an operand whose value was the contents of the register then the recursive iteration would continue. If the second register was itself a pointer to a list structure, we could, for instance, get the value of the Ith element of the Jth sublist.

EX 3: Let us consider the accessing of a LISP data structure (a binary tree) by an arbitrary string of CAR'S and CDR'S.



Performing a CAR operation at a node means to take the left branch while a CDR operation takes the right branch. If we have a sequence of these instructions, we then have a sequence of left and right branchings starting at the top node. Let us represent each node as in (EX 2), where the content of, the first register is now a pointer to the left node,. and that of the second register a pointer to the right node. Let theauxiliary information at the Mth level of the *tree* be the following:

$f_M = +,\ N_M = 0,\ F_M = M + D$

Then if at the M level we want to perform a CAR or CDR we set:

$$V(M + D) = \begin{cases} = 0 \text{ CAR} \\ = 1, \text{CDR} \end{cases}$$

Then we can perform a string of N CAR'S and CDR'S on any LISP data structure by specifying L - the position of the first node -, setting the V(M + D) appropriately, and setting up the following memory configuration:

Register P	C(P)	f_P	N_P	F_P
K	L	+	Kl	D
Kl	N	Iy		

Limitations

The basic limitation in the value generation mechanism is that it "cannot"* be used to express data-accessing procedures which contain in some form the concept of conditional branching. An example of a data structure which cannot be accessed efficiently is that of a symmetric matrix:

$$A[I,J] = \begin{cases} \text{if } I \geq J,\ C\ (BASE_A + I \times D_A + J) \\ \text{if } I < J,\ C\ (BASE_A + J \times D_A + I) \end{cases}$$

There is no way to define this data-accessing mechanism without comparing I with J and making a decision on the comparison or without using an extremely large amount of excess storage which defeats the whole purpose.

We could program accessing of a symmetric matrix by the algorithm described above, but the main computational program would have to make the decisions about which data-accessing mechanism should be activated. There is a way of adding conditional branching to the value generation mechanism, but this addition is dependent on using the auxiliary information required by the name generation scheme. Therefore, a discussion of this new addition will be delayed until the name-generation mechanism is introduced.

B. Name Generation

The concept of a name generation mechanism has appeared in many higher level languages. Language formalism such as the FOR statement in ALGOL, the generator function in IPL-V, and the mapcar function in LISP are examples of the implementation of the concept of name generation. This section describes one way in which the facility of name generation can be incorporated into a

* Cannot is a little too strong here since in many cases conditional branching, may be replaced by the use of pre-stored data in the memory. The trouble is that in most cases the extra storage is so prohibitively large that this approach is ruled out.

computer organization and be combined with the value generation mechanism previously described. The mechanism of name-generation gives the programmer the definitional capability to specify the operands for the computation (including the operands to contain the result). Implicit in the above definitional capability is the ability to specify the number of operands.

The name-generation mechanism is especially advantageous in a microprogram computer where macro instructions involving non-binary operators are easily constructed, In addition, it would be unnecessary to have variability in a macro-instruction format since information concerning data-accessing is not part of the instruction format: there is no need for complex decoding of the instruction format. Pipeline computers also provide a place where name-generation-can be employed advantageously since creating streams of operands is very useful in this type of computer organization.

In the introduction, it was discussed how separating data-accessing from the computation allowed for greater economy of representation (higher code density) in the program for the computation. This statement can be verified by considering the use of the name generation mechanism in a conventional computer organized around binary operations. It has been found that for most problems it is unnecessary to have the three operands for a binary operation explicitly specified in the instruction. Therefore, in order to increase code density, instruction formats have been developed in 'which one or more of the operands are implicitly specified: no address instructions for stack computers,, and single address instructions for computers with accumulators. The incorporation of a name generation mechanism gives the programmer the ability to specify the names of the operands implicitly rather than as part of the instruction. Therefore, the programmer can construct (simulate) through the name generation mechanisms the instruction format or formats which give the highest code density for the particular problem.

Name Generation Mechanism

Based on the previous discussion the name generation must be able to generate a list of argument operand names and result operand names. In order to generate a list of names, there must be a parameter which specifies the number of operands, and a parameter which is modified after each name is generated to'

prepare for the generation of the next name. In addition, it is felt that the mechanism should handle the degenerate case of the three address instruction format, and inner product type calculations. The following name generation mechanism was developed based on the above requirements, and the desire for this mechanism to be similar to the value generation mechanism.

As in the value generation scheme, auxiliary information is attached to each register K:

g_k** - a function used in parsmeter adjustment after each cycle of iteration;

Sk - a register which generates the name of the second operand;

D_k - a register which generates the name of the result operand;

f_k^1, F_k^1 - defined analogously to f_k, F_k

N_k^1 - defines the number of operands generated by register K.

Let us define $OP^1(M)$ as the ith operand *name (address)* generated by the register M. If $i > V(N^1_M)$, then the operand name $OP^1(M)$ is considered to be null.. Let the register K be accessed as the name of a computation, then the following sequential string of names* is generated:

$$OP^1(K),\ OP^1(S_k),\ OP^1(D_k),\ OP^2(K),\ OP^2(S_k),$$
$$OP^2(D_k),\ .\ .\ .\ .,\ OP^i(K),\ OP^i(S_k),\ OP^i(D_k)$$

where the string continues until

$$i = \max\ (V(N^1_k),\ V(N^1_{s_k}),\ V(N^1_{Dk}))$$

We define $OP^i(K)$ in the following way:

$$OP^i(K) = T_{f^1_k}(K, C(K), V(C(K)), V(F^i_k))$$

** Note the 6 pieces of auxiliary information can be grouped into 2 groups of 3, such that each group has the same format as value-generation information.

* There is a difficult problem in deciding where the result operand names appear in the generated list of names: intermixed or at the end of the string. This really depends on the nature of the computations to be performed on the argument operands. It is believed the best solution is for the memory organization to generate a result operand name only when the computation desires to store a result. For the sake of example, the intermixed case is represented.

where after each stage

$$C(K) \leftarrow T_{g_k}(K, C(K), V(C(K)), V(K))$$

so that if C* is the original C(K) then

$$OP^i(K) = T_{f_k^L}(K, L, V(L), V(F_K^1))$$

where

$$L = T_{g_k}^{i-1}(K, C^*, V(C^*), V(K))$$

The name-generation scheme for one step is exactly the same as value-generation where there is no indirection. Instead of using the parameter N_K^1 to specify the level of indirectness it is used to specify the number of operands to be generated. The contents of register K is the parameter which is modified after each cycle, and is modified by the same scheme used to generate the operand name at each cycle.

The following examples illustrate how the name-generation mechanism can be used to generate the names of the operands of some commonly used computational operations:

EX 1: A stack Address Mechanism. Let us define a stack by 2 registers. The first register holds the contents of the top of the stack, and the second contains the address of the second element of the stack. (It is assumed the remaining stack entries are the sequential cells following the second element). Consider the following register configuration:

Reg P	C(P)	f_p	N_p	F_p
K	value of the top element of stack	Iy	O	
Kl	address of the second element of stack	Iy	0	

Reg P	g_p	S_p	D_p	f'_p	N'_p	F'_p
K	Iy	Kl	K	Ix	1	
Kl	Pl			Iy	1	

Let us reference register K as the name of the computation; then the following operand names are generated:

$$\underline{OP^{1}(K) = TI_{x}\ (K, C(K), V(C(K)), V(F^{1}(K)) = K}$$

$$C(K) = TI_{Y}\ (K, C(K), V(C(K)), V(K)) = C(K)$$

$$OP^{1}(S_{k}) = OP^{1}(K1)$$

$$OP^{1}(K1) = T_{I_{Y}}\ (K1, C(K1), V(C(K1)), V(F'(K1))) = C(K1))$$

$$\underline{OP^{1}(K1) = C(K1) = \text{address of second element of stack}}$$

$$C(K1) = T_{P1}\ (K1, C(K1), V(C(K1)), V(K1)) = C(K1) + 1$$

$$C(K1) = \text{address of third element of stack}$$

$$OP^{1}(D_{k}) = OP^{1}(K) = K$$

We get that computation K is a binary operation in which its argument operand names are K and the address of the second element of the stack, and the result operand is also register K. If the value function is now applied to the argument operands, we get V(K), V(address of second element of stack), but V(K) = value of the top element of stack. The value of the second operand cannot be predicted since we don't know the auxiliary information attached to the second element of stack, but note that it could be itself an operand name which points into a complex data structure: it' could be in the form of the name of a matrix element, as discussed in example 1 in the value generation section.

So we have seen that the correct operand names are generated for stack operations,, and the addressing mechanism is set up properly for future stack operations.

Ex 2: Let us generate the names of the elements of a vector A of length N. There are two ways that this can be done, one in which the base address of A is destroyed, and the other in which a cell must be initialized to zero at the beginning of the procedure. Let us consider the former case; consider the following register configurations:

Register P	C(P)	f_p	N_p	F_p
K	$BASE_A$			
Kl	N	Iy	0	

Register P	g_p	S_p	D_p	f'_p	N'_p	F'_p
K	Pl	0	?	Iy	Kl	

Then $OP^1(K) = T_{Iy}(K, C(K), V(C(K)), V(F^1_k))$

$= C(K) = BASE_A$

remember $C(K) = T_{Pl}(K, C(K), V(C(K)), V(K))$

$= C(K) + 1 = BASE_A + 1$

so

$$OP^i(K) = TI_Y(K, T_{Pl}^{i-1}(K, C(K), V(C(K)), V(K)), V(C(K)), V(K))$$

$$= T_{Pl}^{i-1}(K, C(K), V(C(K)), V(K))$$

$$= C(K) + (i - 1)$$

$$= BASE_A + i - 1$$

So we generate the names

$BASE_A$, $BASE_A + 1$, ...e $BASE_A + N - 1$

which are the elements of the vector A.

This examples is not exactly right since the string of names generated does not include any names generated for the result operands. As mentioned in a previous footnote, where the result operand names appear is a function of the type of computation performed, and therefore in order to simplify the example they have been ignored.

EX 3: Let us generate the names of the diagonal elements of a matrix A of dimension N × N. Let A [I, J] = loc ($BASE_A$ + (I - 1) N+'J-1). We then have A [I, I] = loc ($BASE_A$ + (1-l) × (N + 1)). For setting up this computation there is a very important point which must be considered: whether the quantity (N + 1) should be computed each time, or whether it is a constant computed only once. This problem

points up a limitation of the recursion computation capability since there is no way to generate intermediate results and save them for future steps in the computation, except for the contents of register K, but if the value is changed at one step then it is changed at all steps. It appears that for complex data structures to be done efficiently the computational algorithm must interact with the name generation mechanism to set up constants. Two register configurations will be set up to illustrate both of the ways the problem can be attacked:

Case 1: (N + 1) is defined previously

Register P	C(P)	f_p	N_p	F_p
K	$BASE_A$	Iw	0	KN1
KN	N	Iy	0	
KN1	N+1	Iy	0	

Register P	g_p	S_p	D_p	f'_p	N'_p	F'_p
K	+	0	?	Iy	KN	

We get $OP^1(K) = C(K) = BASE_A$

$C(K) = T_+ = .C(K) + V(K)$

$V(K) = T_{Iw} = V(KN1) = (N + 1)$

so $C(K) = C(K) + (N + 1)$

Case 2: (N + 1) is computed.

The-only change is register KN1:

$c(KN1) = 1$

$f_{KN1} = +,\ N_{KN1} = 0\ ,\ F_{KN1} = KN$

then $V(KN1) = C(KN1) + V(KN) = (N + 1)$

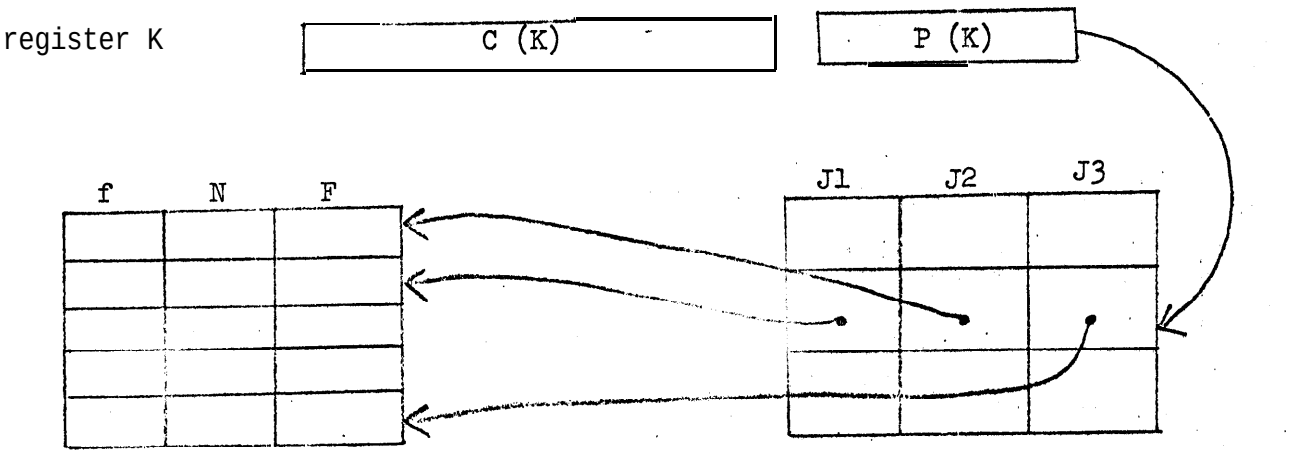
The problem of computing the names of a row of a matrix: A [I,,*] are similar to the above problem since A [I, *] = loc of $(BASE_A + (I - 1)\ N + J)$, J = 1, . . N. The quantity (I - 1) × N would need' to be pre-computed or generated at each stage.

III. Computer Organization

The next sections are concerned with how the value generation and name generation mechanism already discussed can be integrated into the framework of a total computer organization. The first section discusses a possible technique for adding the auxiliary information to a memory structure; the second section discusses an instruction format which can advantageously use the value end name generation mechanisms; the final section is concerned with possible extensions.

A. Memory Layout

Before discussing the instruction format, it is worthwhile to consider a possible schema for the memory organization:



Attached to each register K, we have a pointer, P (K), which points to a table entry which contains 3 additional pointers. . Each of'these pointers $(J_1, J_2, J_3)_{p\ (K)}$ points into table containing, for each entry, a function and two operands. We then get that -

$$(f_k, N_k, F_k) = (f, N, F)_{J1\ p(K)}$$

$$(f^1_k, N^1_k, F^1_k) \quad (f, N, F)_{J2,\ p(K)}$$

$$(g_k\ S, D) = (f, N, F)_{J3,\ p(K)}$$

This double level technique for determining the auxiliary information has been used since the 9 pieces of auxiliary information can be broken up into 3 groups, each having the same format.

B. Instruction Format

The instruction formats have been designed to make full use of the implicit name generation capability and therefore get high code density whenever possible, There are three pieces of data which must be either implicitly or explicitly stated in each instruction format:

1) OP - operation code;

2) K - the name of the cell generating the names of the operands;

3) J - the entry in the table (J1, J2, J3) which specifies the auxiliary information.

Let us define KI, and JI as registers which contain values for K and J, respectively. There are four basic instruction formats:

1) OP - this instruction format is used when the name generation mechanism is fixed like a stack address mechanism. 'The values of K_I, J_I are used for K, J.

2a,b) OP-K - this instruction format is essentially a single address instruction in which J = P(K). There could be another variation with J = J_I.

3) OP-J-where K = KI. This instruction is advantageous when for one instruction a different set of auxiliary information is desired: this is useful when the contents of a register is to be altered, and the value of the register is not its contents.

4) OP-K-J, The justification for this format is similar to that of format 3.

There Will be other formats but they will be concerned with modifying pointer variables P(K), KI, JI, entries (J_1, J_2, J_3)* and (f,N,F)*.

Let us consider how the addressing schemes of conventional computers can be represented in this computer organization.

* It'is really not necessary to modify these entries, therefore, they could be located in a slow write fast read memory.

Ex 1: Stack computer - a computer where instructions are just operation codes and can be accomplished through the format OP, where K_I and J_I point to a stack address mechanism previously discussed.

Ex 2: Single address computer - in this organization one operand is contained in the instruction and the second and result operands are the accumulator. Let us simulate this instruction format with the format OP-K where $J = J_I$. Let us define register A to be the accumulator. Consider the following auxiliary information for every register:

$$S_K = A, \qquad D_K = A \qquad g_k = Iy$$
$$F^1_K = ?, \qquad N^1_K = L \qquad f^1_K = I_x$$

then $OP^1(K) = T_{I_x}(K, , ,) = K$, first operand name

$$OP^1(S_k) = OP^1(A) = TI_x(A, , ,) = A, \text{ second operand name}$$

In a similar manner a computer with an instruction format of single address plus increment can be implemented using the format OP-K-J. 'If, in most cases, the index register associated with a variable is fixed, then the format OP-K can be used. The format of the /360 with base registers can be accomplished by using the format OP-J. The purpose of these examples is to show how a programmer could structure the generation of the names of the operand of a computation so as to be most efficient (high code density) for the particular application.

IV. Extensions and Further Research

In the section on value generation a limitation of this mechanism was discussed. This problem arises due to the inability to alter the value generation algorithm based on the comparison between two operands.. It is believed that this problem can be remedied if the auxiliary information tied to the name-generation mechanism is used where comparisons are necessary. There are 3 triplets of auxiliary information; let one of these triplets be used to make the comparison, and, based on the comparison, use one of the two remaining triplets to generate the value of the register.

Another extension is oriented towards high code density. In the instruction format previously mentioned the variable K which defined an address in memory was assumed to be large enough so as to address all of memory. In the B5500, it was shown that through the use of a program reference table, the operand address in an instruction could be made to be much smaller than the length of memory, thereby increasing code density. The program reference table is an address generation mechanism easily simulated in this organization. Therefore it would seem reasonable to have instruction formats where the size of K was smaller than the length of memory. This smaller size K could also be used as increment quantity which would allow a simulation of one of the IBM-360 instruction formats: RX.

There are two important questions so far ignored, whose answers will eventually decide the utility of the concepts developed in this paper:

1) How can the name and value generation mechanisms be implemented in computer hardware, and with what speed?
2) Can a compiler for a higher level language produce machine code which takes "advantage" of this data accessing mechanism?

Further research is intended to answer these questions.

Bibliography

1. McKeeman, W.M. Language directed computer design. AFIPS Cont Proc 1967 FJCC pp 413-417.

2. Iverson, K.E. A Programming Language Wiley 1962.

3. Holland, J.H. A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously. Proc. EJCC (1959).

4. Murtha, J.C. Highly Parallel Information Processing Systems. Advances in Computers, Vol. 7.

5. Standish, T.A. A Data Definition Facility for Programming Languages. Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University.

6. Barton, R.S., A New Approach to the Functional Design of a Digital Computer Proc. WJCC 19, (1961) pp. 393-396.

7. Cheatham, T.E. The introduction of definitional facilities into higher level programming languages Proc. AFIPS FJCC(1966), 623-637.

8. Galler, B. and Perles, A.J. A proposal for definitions in ALGOL CACM 10(April 1967).