DESIGN FEATURES OF THE

BCC 500 CPU

Charles F. Wall

Technical Report R-1
Issued January 3, 1974

THE ALOHA SYSTEM, Task II
Department of Electrical Engineering
University of Hawaii

ACKNOWLEDGMENTS

ABSTRACT

This report describes the design features of the BCC 500 Central Processing Unit. The CPU was designed to directly implement most of the features of a high level interactive Systems Programming Language (SPL). It directly implements in hardware the following features: a function call and return mechanism, descriptors, a powerful addressing structure and a wide variety of floating point features.

TABLE OF CONTENTS

0. Background and Overview

The BCC 500 computing system was designed to provide a large number of users with simultaneous remote access to centralized, general-purpose computing and file storage facilities. The system was developed for a wide variety of commercial and scientific applications and to support batch, remote-batch, interactive and real-time jobs. Emphasis was given to providing rapid response for large numbers of small jobs requiring moderate computational capabilities.

To economically provide a utility of this nature the hardware configuration consists of a number of specialized processors all connected to a central memory. The operating system is distributed among these independent processors, which communicate by means of the central memory. Each is dedicated to one of the following tasks:

(a) process scheduling;

(b) memory management;

(c) character input/output from terminals;

(d) system supervision and monitoring;

(e) running user programs.

The task of running user programs is performed by two independent Central Processing Units (CPUs). A module of the operating system runs on the CPU to provide users with protected access to system services. This module is divided into a monitor and a utility section. The monitor is common to all users. It may not be looked at or modified by user programs; its services are accessed by a set of monitor calls. Each monitor call checks

the user's authorization to make the call, validates the parameters passed and then proceeds to invoke the desired service. The utility can be unique to each user (although only one has been produced to date). It contains a number of useful functions that extend and individualize the users' interface to the operating system. Viewed in another way, the monitor contains a set of functions that provide the user with access to primitive operating system services, and the utility uses these basic functions to provide users with more convenient access, including the executive command language.

1. Introduction

In this paper we are concerned with describing the CPU. Many features of the CPU architecture were influenced by the ideas and structures found in the Burrough's B-5000[1] series of computers and in the MULTICS[2,3] system. The CPU was designed to implement in firmware[10] and/or hardware most of the high level features of an interactive Systems Programming Language (SPL) and to function in a multiprogrammed, multiple processor system.

SPL provides systems and applications designers with a high level development and implementation tool to provide effective applications programs. The CPU design efficiently supports these application programs by implementing the following features:

- A function call and return mechanism;
- Field Descriptors that permit full-word and part-word items in tables to be accessed efficiently;
- String Descriptors and string handling operations to speed up compiling and non-numeric processing;
- Array Descriptors that support complex array structures and permit multi-dimensional arrays to be accessed without any program multiplications;
- An addressing structure that provides for easy code relocation and supports the basic data structures of SPL;
- A simple instruction set that provides for easy mapping from SPL operations to machine instructions;
- A wide variety of floating point features;
- A virtual machine for both user and system functions.

SPL motivates programmers to organize programs into a collection of small procedures or routines called functions, with arguments and results being explicitly communicated by means of the function call and return mechanism. In this way side effects are minimized and program debugging is enhanced. Each function has a local storage area (called the local environment) that is separated from the code and is usually allocated on a stack; storage for a function may be allocated in a fixed location, however, to provide a FORTRAN-like capability.

An important feature in SPL that is directly implemented on the CPU is the use of descriptors to access various data structures. Information in the descriptor allows the CPU to check and ensure that the access is correctly specified. Descriptors are implemented on the CPU to provide efficient access to fields, strings, and arrays. These descriptors along with the address features and the instructions that support their use are discussed in Sections 3, 4 and 5, respectively.

The addressing features of the CPU, beyond the modes that are used with the various descriptors, are designed to support the basic data structures found in SPL and to facilitate incremental compiling by providing for easy code relocation. The CPU has a simple instruction set to facilitate easy translation from SPL operations to machine operations. It also provides processes with an environment called a virtual or user machine.[5,6,8] A process is considered to be a program in its virtual execution environment and is individually scheduled. Each user process sees a virtual memory of 256K words called its virtual address space.

The CPU contains a small number of registers that are an integral part of the function of each process. We list and describe them now, as we will be referring to them throughout the remainder of the paper. The information in these registers must be saved when a process is blocked and the CPU is assigned to a new process. In addition, we will refer to a few registers which the CPU uses for its activities in executing a process, but are not part of the state of the process. Figure 1.1 contains a list of these registers.

The central registers AR, BR, CR, DR are used by various arithmetic and logic operations which operate on single and double precision data. The E-register (ER) is used in floating point operations to contain the exponent. The local (L) and global (G) environment registers are base registers that point to storage for the presently active function and to a common storage area, respectively. At the start of each instruction cycle, the indexing register (IR) and the source register (R) are set to the contents of the index register (XR) and the program counter (P), respectively. Both registers may take on other values as instruction execution proceeds.

• Central Registers

(AR)	A - register
(BR)	B - register
(CR)	C - register
(DR)	D - register
(ER)	E - register

• Registers Used in Addressing

(XR)	Index register
(L)	Local environment register
(G)	Global environment register
(P)	Program Counter
(R)	Source register *
(IR)	Indexing register *

• Other Special Registers

(SR)	Status Register
(CTC)	Compute Time Clock
(IT)	Interval Timer

*Not part of the state of a process

Figure 1.1 Machine Registers

2. Function Call and Return Mechanism

The function call and return mechanism is important since SPL programs consist of a number of small functions. An instruction called BLL (Branch and Load the Local-environment register) implements this mechanism. The BLL addresses a two-word branch (function or return) descriptor which contains all the information necessary to accomplish function calls and returns.

SPL provides a very flexible function call mechanism. A function may have a number of arguments and return any number of results. The arguments may be arbitrary expressions. Results can be stored into any arbitrarily specified variables. The value of the first result, if there is any, is the value of the function. Thus a function call of the form:

PTR'TO'NODE ← SEARCH'LIST(EMP'LIST,ID);

could cause a list of employees to be searched and return a pointer to the node containing the information associated with a particular employee. SPL allows for a "failure" return from a function. We could recode the above function as follows:

PTR'TO'NODE ← SEARCH'LIST(EMP'LIST,ID//NOT'FOUND);

If the desired employee is not found in the list, then a fail return will cause control to go to the statement labeled "NOT'FOUND". We could, of course, code the search list function so that it returns the information in a node directly rather than returning a pointer to the node. The function call would appear as follows:

SEARCH'LIST(EMP'LIST,ID;AGE,DEPT'NO,SEX//NOT'FOUND);

In this case the information that might be contained in the node is directly returned in the variables "AGE", "DEPT'NO" and "SEX".

In the CPU implementation of BLL the two-word branch descriptor contains fields specifying:

- The address of the called routine's entry point;
- Whether the transfer is a function call or a return;
- Whether the storage for the function must be allocated from a stack;
- Whether arguements or results are to be copied;
- Whether the function is a FORTRAN-type function;
- A field called the environment field, used to determine the new local environment register value in a manner to be described.

All the features of the SPL call mechanism and most of the subroutine call features of FORTRAN are implemented in hardware by the BLL instruction which, in conjunction with the branch descriptor, provides for all of the following actions:

On calling it:

1) Calculates the effective address of the entry point in the called routine;

2) Acquires the new local environment and obtains storage if the function allocates space for its local environment on the stack;

3) Copies arguments and checks them for correct type and number;

4) Computes the return descriptor and saves it in the first two words of the new local environment;

5) Transfers control to the called routine.

On returning it:

1) Obtains the old local environment from the return descriptor;

2) Copies results and checks them for correct type and number;

3) Returns control.

We now describe these actions in detail.

When the BLL instruction is executed, the first step is to compute the
address of the entry point for the called routine. The new local environ-
ment is acquired. If the called function has a fixed local environment then
the environment field of the branch descriptor is taken as the new value of
the local environment register L, which we call NEWL. Space for a fixed
function's local environment is allocated at all times and its contents are
preserved between function calls. Normally, space for a function's local
environment is allocated from the stack. Two words in the global environment
describe this stack. The stack pointer (SP) addresses the first unused word
and the stack limit (SL) addresses the last word allocated for the stack. For
stacked functions the environment field in the function descriptor indicates the size
rather than the address of the new local environment. NEWL is set to the value
of the stack pointer and the stack pointer is incremented by the environment
field. (See Fig. 2.1).

Arguments are copied next if there are any. The calling function
supplies a list of parameter addresses called actual argument words (AAW)
and the called routine contains a corresponding list of formal argument
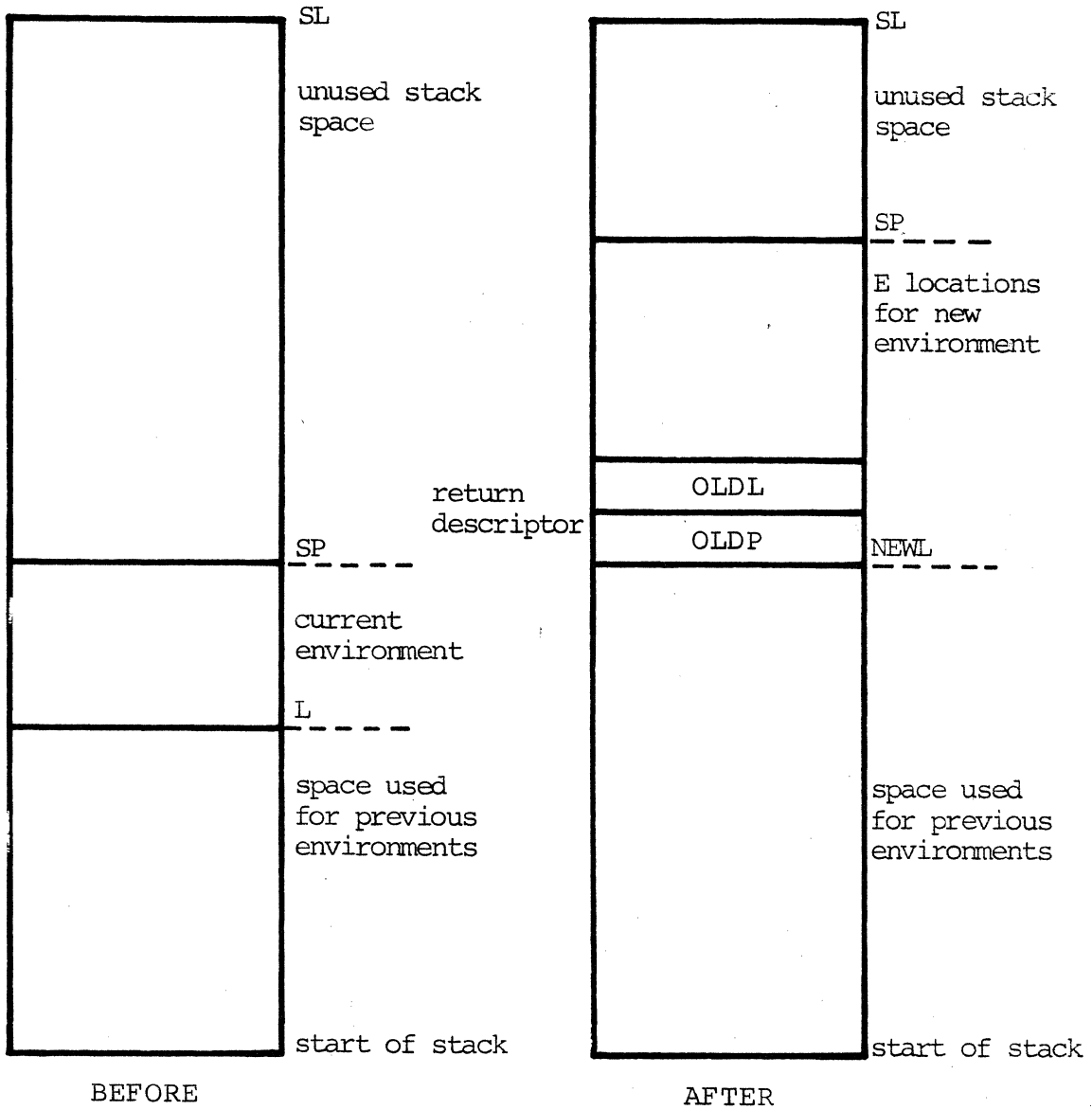
Figure 2.1 Allocating a Local Environment on the
Stack During a Call

words (FAW). An actual argument word contains the following information:

Structure of the argument:

       variable

       computed scalar

       array element

       array

Type:

integer	(1 word)
long	(2 words - a 48-bit integer)
longlong	(4 words - a 96-bit integer)
real	(2 words - a 48-bit floating point number)
double	(4 words - a 96-bit floating point number)
complex	(4 words - a 48-bit real and 48-bit imaginary)
string	(4 words - a string descriptor)
label	(2 words - a BLL descriptor)
pointer	(1 word - address of a memory location)
unknown	

End flag

Address of argument

The formal argument word contains similar information for the type, end flag and the address of the formal argument; the structure of a formal argument is specified as either scalar or array. The FAW indicates whether the address of the argument is copied or the value is copied. Arguments are copied one

at a time. An error occurs if the AAW type is not the same as the FAW type unless one and only one of them is of type unknown. The structure of the actual argument is checked with the structure of the formal argument according to the following table:

| | | FAW's | |
		Scalar	Array
	Variable	OK	Error
AAW's	Computed scalar	OK	OK if FORTRAN type fn
	Array element	OK	Error
	Array	OK if FORTRAN type fn	OK

Copying continues until an end flag occurs. If both end flags (AAW and FAW) do not appear at the same argument level then the wrong number of arguments have been supplied and an error occurs. The BLL does not provide for type conversion.

A return descriptor is computed and stored at NEWL and control is passed to the called routine. The return descriptor contains the old program counter and old L in the environment field.

On a return, if the function had its local environment on the stack, the stack is unwound by setting the stack pointer to L and NEWL to the contents of the environment field in the return descriptor. If the return is from a function with a fixed environment then OLDL was saved in the environment field of the return descriptor and is used to reset L. Failure returns are accomplished by addressing a return descriptor that may cause a return to a non-local label and may cause several stacked environments to be removed from

the stack. Results are copied in the same manner and with the same checks that are provided for the call. Control is returned to the calling routine or to the latest incarnation of the routine containing the failure return label if a fail return occurs.

3. Full-Word and Part-Word Field Accessing

SPL derives a considerable amount of its flexibility and versatility from the use of fields (the notation was adopted from Bell Labs' L^6 language). For example, assume we wanted to create the following data structure:
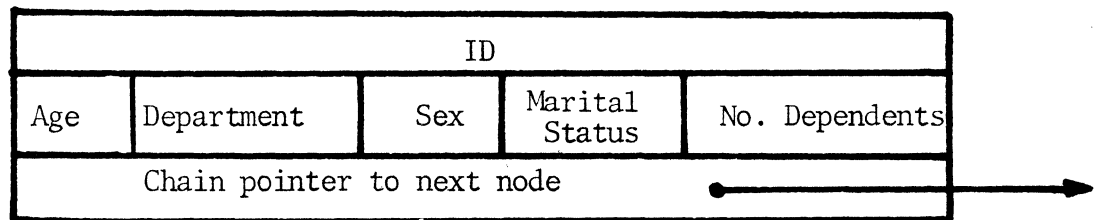
where the node contains an employee's ID (social security number), age, department, sex, marital status, number of dependents and a pointer to the next employee on the list. We can define this structure in SPL by the following declarations:

```
DECLARE FIELD ID(∅);
DECLARE FIELD AGE(1:∅,5),
               DEPT(1:6,12),
               SEX(1:13,13),
               MAR'STAT(1:14,14),
               NUM'DEP(1:15,23);
DECLARE FIELD CHAIN(2);
```

where the fields in words zero and two of the node are referred to as full-word fields and the various fields in word one are called part-word fields.

If P is a pointer to a node of this type, in SPL the statement

    P.AGE ← P.AGE+1;

will increment the age field. Component selection by a field can be used anywhere in place of a simple datum. We can extract the department number simply by coding

    DEPT'NO ← P.DEPT;

and can insert a new department number by coding

    P.DEPT ← NEW'DEPT'NO;

where the extraction operation right justifies the department field in the result and the insertion operation puts the new department number in the proper bit locations. A field may be a signed quantity and if so its sign is extended on extraction.

The field facilities of SPL are supported by a hardware implemented field descriptor and two CPU addressing modes. A field descriptor is composed of a single field indirect address word which contains the following information:

- Size of the field in bits
- Address of first bit of the field
- Sign extension flag
- Signed displacement field

If we assume we have the following structure:
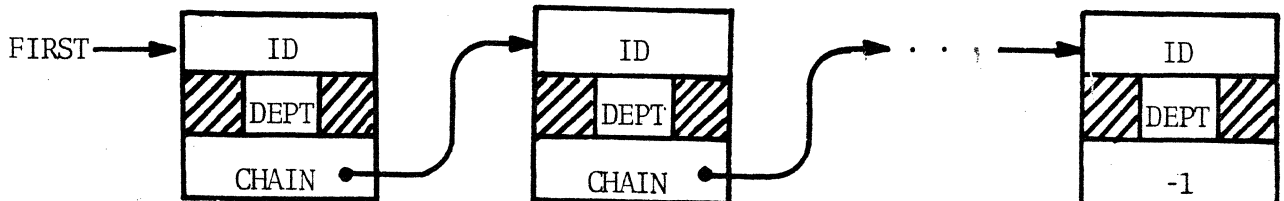
Given that FIRST is a pointer to the starting node, then the SPL code to find a node that contains a particular department number is as follows:

GOTO FOUND IF PTR.DEPT=DEPT'NO FOR PTR←FIRST,PTR.CHAIN WHILE PTR≠-1;

The machine code generated for the statement is as follows:

LDA FIRST	Load A-register with first pointer
BRU R'[2]	Branch source relative plus 2
LDA PTR.2	Pointer-Displacement addressing (PD)
STA PTR	Store the A-register in PTR
ICP -1	Compare A-register with -1
BEQ R'[5]	Branch if equal to source relative plus 5
LDA DEPT[PTR]	Base-Index addressing (BX)
ICP DEPT'NO	Compare with DEPT'NO
BEQ FOUND	Branch on equal to found
BRU R'[-7B]	Branch to LDA PTR.2

This code simply loads the A-register with the pointer to the first node on entry or loads the A-register with the pointer from the chain field of a node on subsequent traverses of the loop. The content of the A-register is stored in a variable called PTR and compared with -1 to see if the end of the list has been reached. If the A-register is equal to -1 then the machine branches out of the loop by transferring control five instructions beyond the present instruction. If the A-register is not equal to -1 the department field of the node presently pointed to by PTR is loaded into the A-register and is compared with the department number desired. If equal, then the routine branches to the instruction labeled FOUND, otherwise the routine loops and continues searching the list.

Two instructions in this sequence directly reference fields in the nodes.
The "LDA PTR.2" instruction extracts the full-word field "CHAIN" from the
presently accessed node and loads it into the A-register. This instruction
is accomplished using the pointer-displacement mode of addressing (see Fig.
3.1).
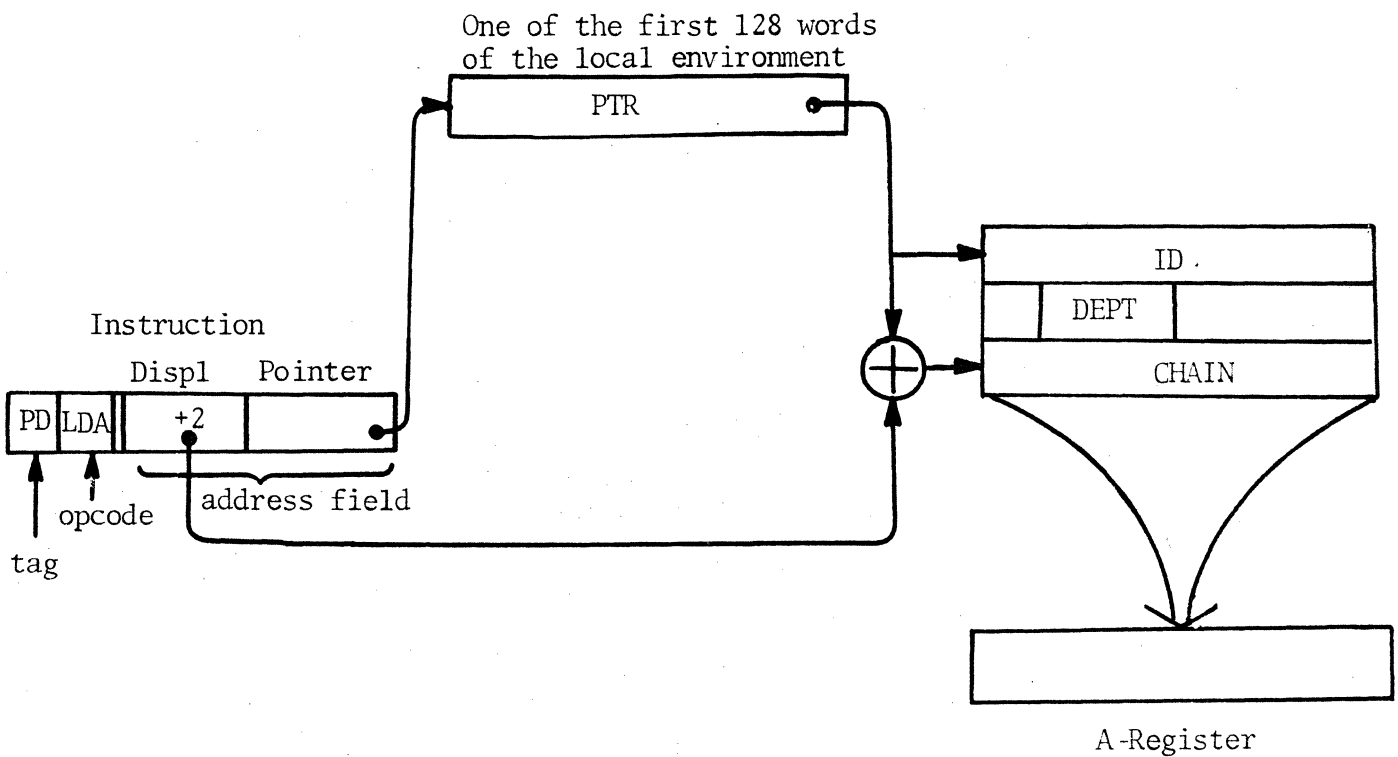
Fig. 3.1 PD Addressing for LDA PTR.2

The "LDA DEPT[PTR]" instruction extracts the part-word field "DEPT" from the presently accessed node and loads it right justified in the A-register. This instruction is accomplished using the base-index mode of addressing in conjunction with the field descriptor for "DEPT" (see Fig. 3.2).
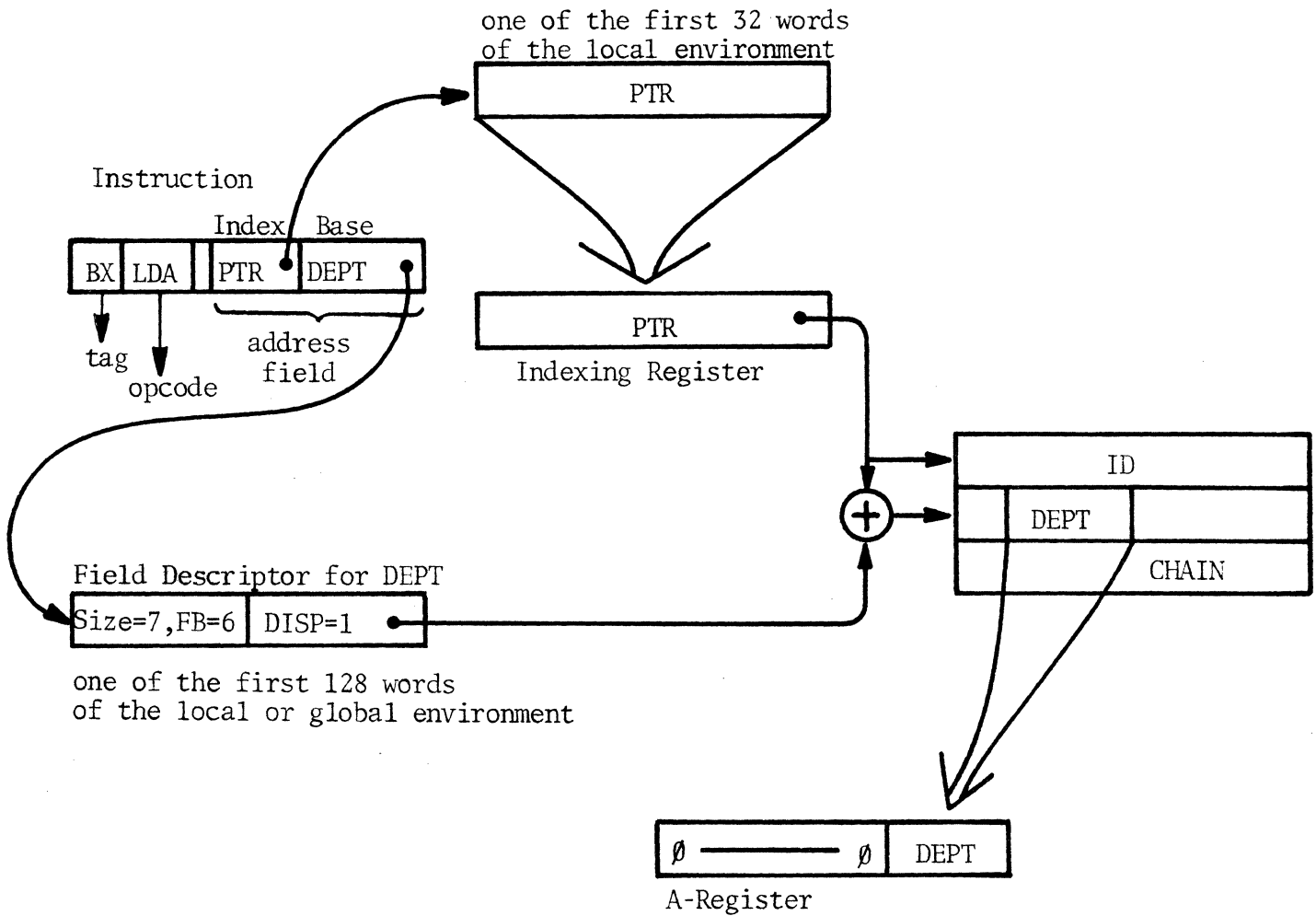
Fig. 3.2 BX Addressing in Conjunction with the Field Descriptor by DEPT

Insertion of a new department number into a node would be accomplished by storing the contents of the A-register ("STA DEPT[PTR]").

4. String Processing Features

In SPL a string is described by a four-word string descriptor of the following form:

Begin Pointer (BP) - points to character before first character in string storage area

Read Pointer (RP) - points to last character read from string

Write Pointer (WP) - points to last character written onto string

End Pointer (EP) - points to last character in the string storage area

Each pointer is a string indirect address word and contains the following information:

String type IAW

Character size: 6, 8, 12 or 24 bit characters

Character position in word

Word address

A typical example of a "new" string and a string after some "reads" and "writes" is given in Fig. 4.1 which shows where the various pointers in the string descriptor point.

A. "New" String (empty string storage area)

First C Position							Last C Position

BP
RP
WP

EP
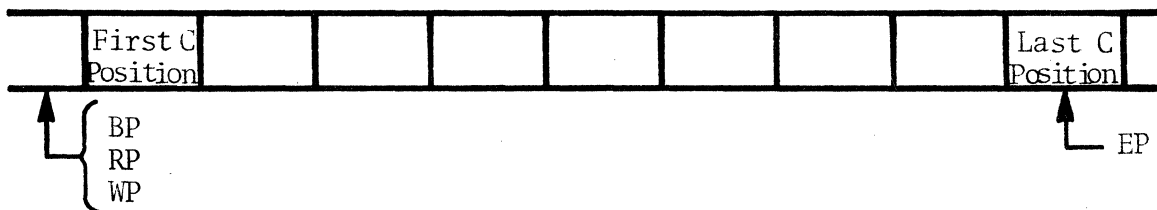
B. After a few Reads and Writes

First Char			Last C Read			Last C Written	Last C Position

BP   RP   WP   EP
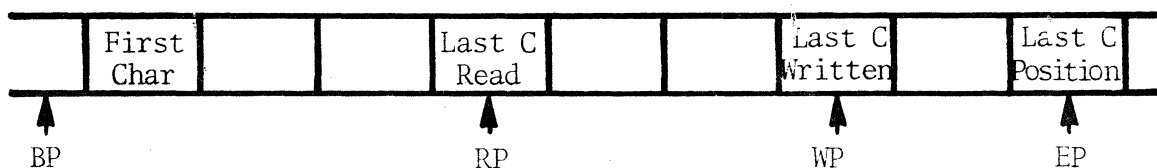
Fig. 4.1 String Descriptors and Where They Point

SPL operations on strings have low level counterparts in CPU instructions as shown in the following chart:

SPL Operations	CPU Instructions	
Read and Write Characters	Increment String Descriptor	(ISD)
	Decrement String Descriptor	(DSD)
	Add to String Pointer	(ASP)
Compute Length of String	Compute Length of String	(CLS)
Copy or Move String	Move String	(MVS)
Compare Strings	Compare String	(CPS)

The increment and decrement string descriptor (ISD and DSD) instructions work with pairs of string indirect words in the string descriptor. During a read operation the read pointer is checked in conjunction with the write pointer to make sure that any attempt to read beyond the write pointer is trapped. The ISD and DSD instructions facilitate character reading and writing while the ASP instruction facilitates accessing the Nth character in a string. The actual characters in the string are read by loading indirectly through the read pointer string IAW and written by storing indirectly through the write pointer string IAW.

5. Array Referencing

Arrays in SPL may be of any dimensionality from 1 to 7. Marginal indexing[4] is used to access arrays that are stored in row major order. For example if we declare a real array A as follows:

DECLARE REAL ARRAY A[3,4,5]

then A is an array descriptor, which points to an array of three array descriptors, which in turn point to an array of four array descriptors each of which points to the first element of a five element row of the array. Figure 5.1 illustrates this array which has 120 words of contiguous storage allocated for the real numbers.

The CPU supports the SPL array structure and array referencing by directly implementing and providing low level operations on array descriptors. An array descriptor is two words long and is composed of an array indirect address word and a pointer to the first word of the array and contains the following information:

- Lower bound (zero or one)
- A trap bit to facilitate subscript checking
- Multiplier to allow for array elements up to 64 words
- Upper bound
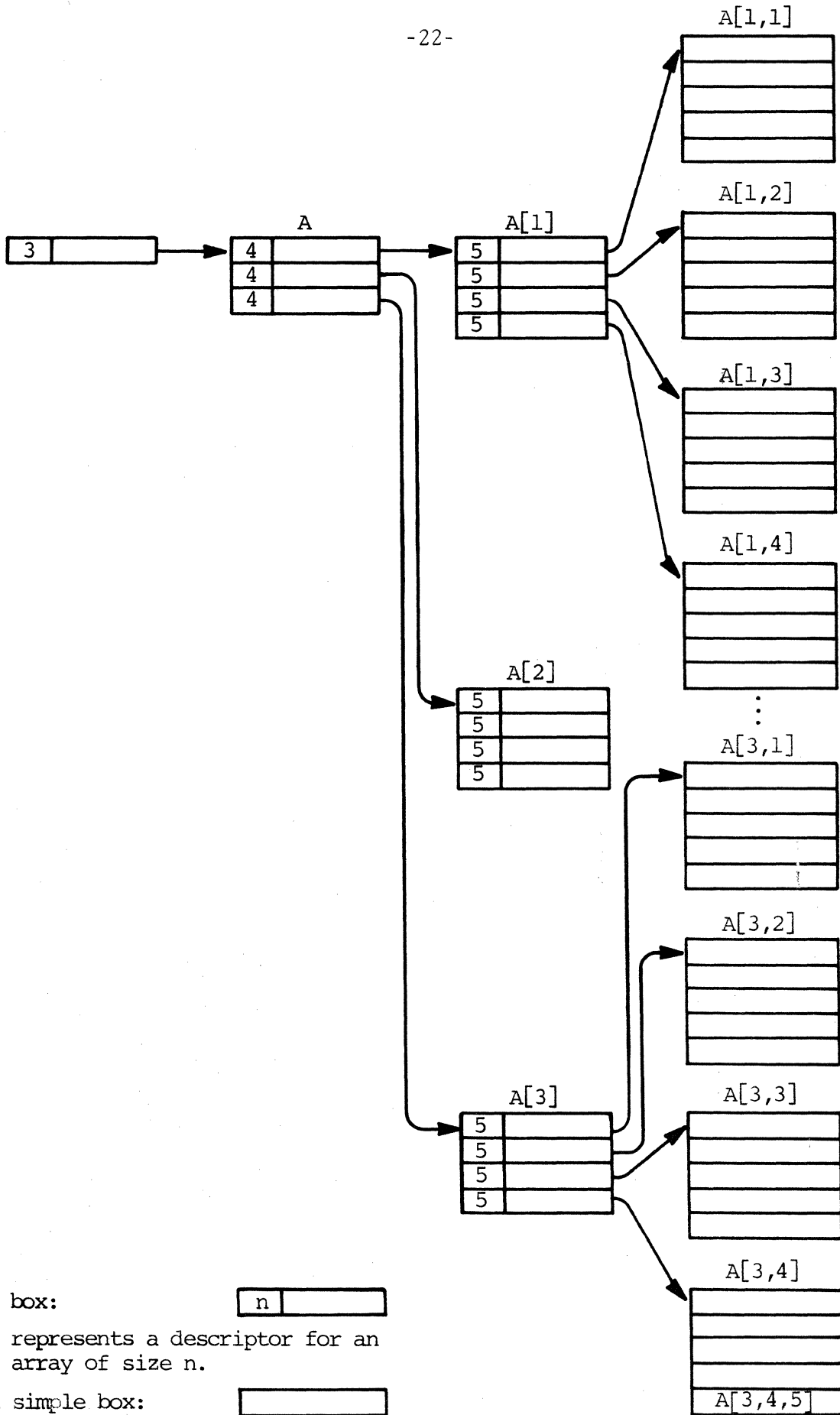
A box: `n |     |`

represents a descriptor for an
array of size n.

A simple box: `|        |`

represents a real number.

Figure 5.1 Marginally Indexed Array Structure

This information allows the following functions to be accomplished:

- Allows a zero or one lower bound

- Performs bounds check on the subscript

- Multiplies the subscript by the size of the array element, allowing for element sizes up to 64

- Checks to see that the number of subscripts supplied is the number expected

- Provides an 18-bit base address for the array

Arrays are referenced using either the base-index or base-index-displacement mode of addressing. This is similar to the method used for accessing part-word fields, only in this case we can reference elements that are full words or larger. If we consider the following 3 by 3 integer array:

$$A = \begin{bmatrix} A[1,1] & A[1,2] & A[1,3] \\ A[2,1] & A[2,2] & A[2,3] \\ A[3,1] & A[3,2] & A[3,3] \end{bmatrix}$$

Fig. 5.2 Array A

we would set up this structure in SPL by the following declaration:

DECLARE INTEGER ARRAYONE A[3,3];

This array is stored contiguously in row major order and is addressed by marginal indexing as follows:
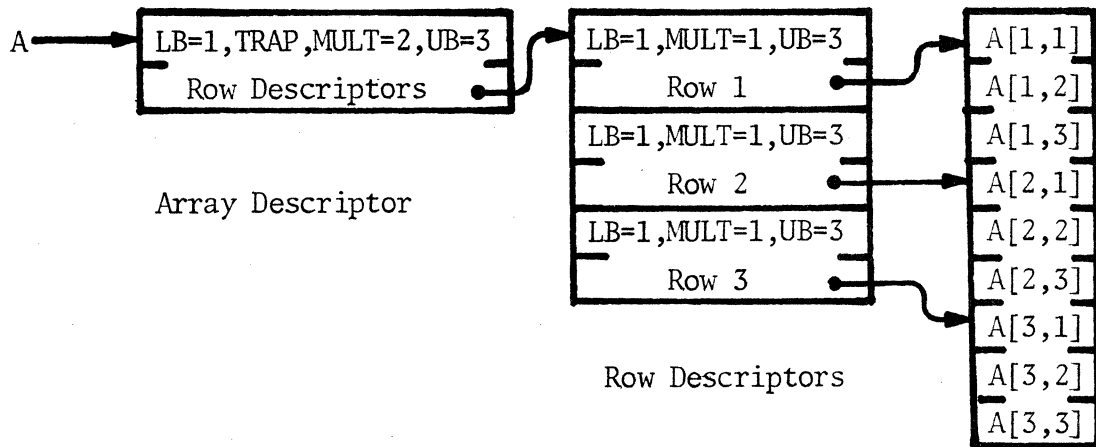
Figure 5.3 Marginal Indexing

A points to an array descriptor, which in turn points to an array of row descriptors, each of which points to the first element of a row of array A. Assume that row index K and column index L are located within the first 32 words of the local environment. Also, assume that K=3 and L=3, then the code generated for B←A[K,L] is as follows:

LAX A[K]   (BX addressing)

This Load Array indeX instruction leaves the address of the descriptor for the Kth row in the X-register
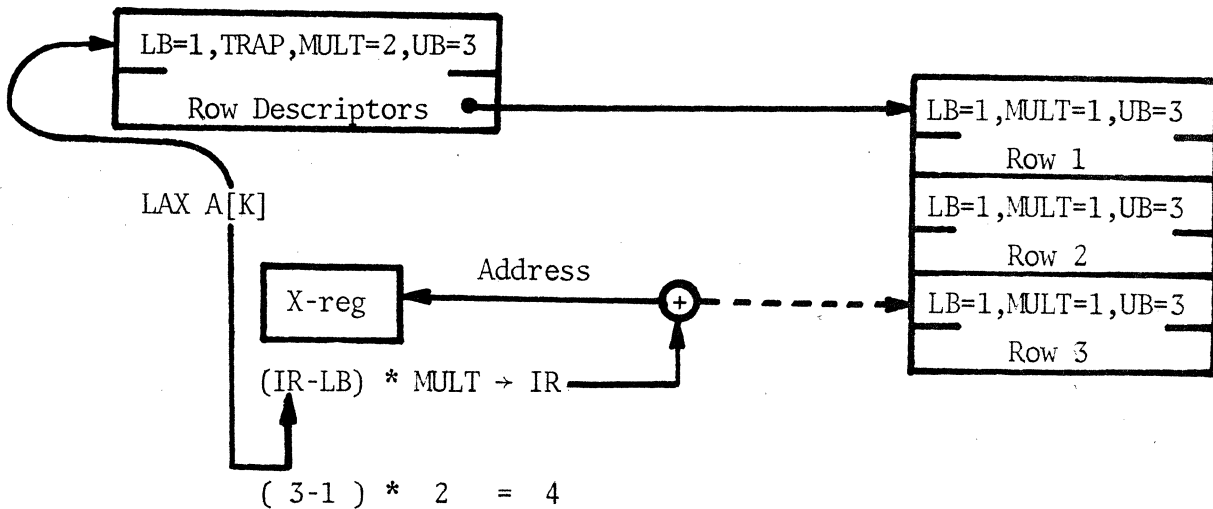
Figure 5.4 Diagram of LAX A[K] Execution

followed by:

LDA ($X') [L]   (BXD addressing)

STA   B
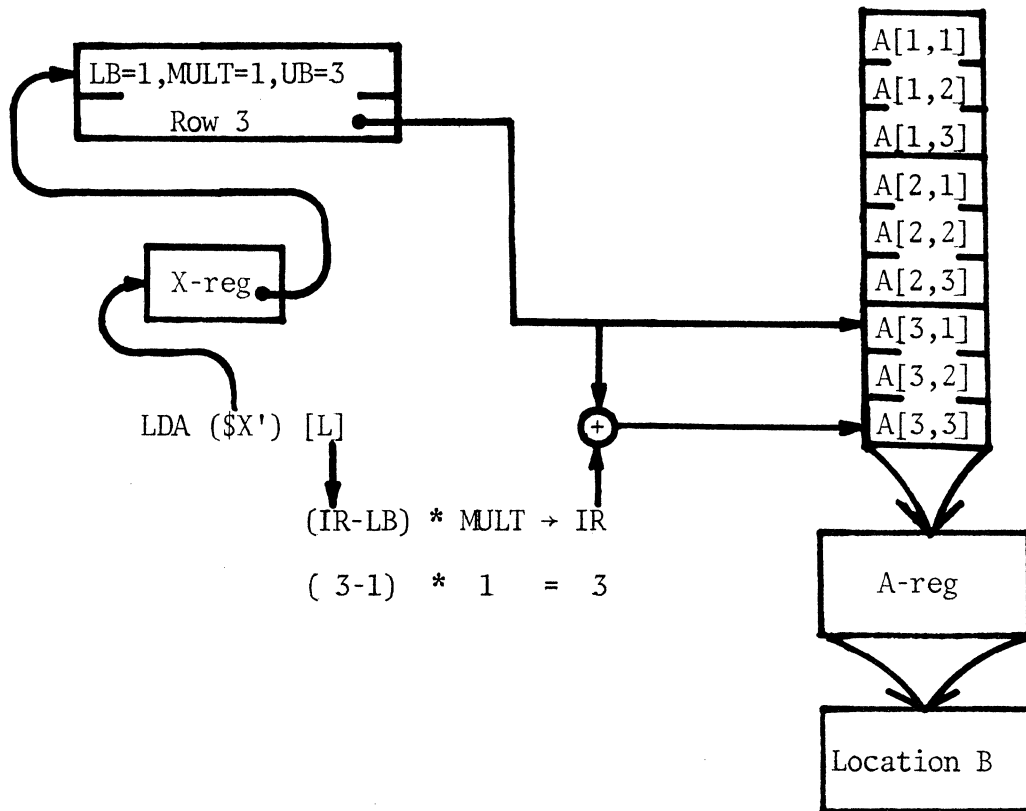
Figure 5.5 Extracting A[3,3] and Storing it in B

The LAX (Load Array indeX) instruction treats the trap bit in the array descriptor as if it were complemented in order to facilitate checking the number of subscripts as the array referencing proceeds from one level of indices to another. Bounds checking occurs at each and every indirection through a descriptor.

The CPU facilitates efficient and effective array referencing by the use of array descriptors combined with the base-index and base-index-displacement modes of addressing and a special instruction that loads an array index.

6. Addressing

The addressing modes are designed for SPL addressing requirements. In the case of descriptors we have seen three addressing modes (pointer-displacement, base-index, and base-index-displacement) that access field, string and array structures. In the discussion to follow, the various addressing modes will be considered and described in conjunction with the requirements of SPL and the virtual address space. Some of the more important factors to be considered are as follows:

- Programs are organized into a collection of small, self contained routines called functions. Each function has some local storage area of its own called its local environment. Normally a function references objects that are either in the local environment or are passed as parameters. Functions can access shared data in a global environment;

- Code must be easily relocatable;

- The data manipulation operations of SPL must be directly supported;

- To save register loading and allocation it is desirable to be able to use core locations as index and pointer values;

- It is necessary to be able to conveniently address a 256K (18-bit) address space, even though an instruction has only up to a 14-bit address field.

In order to be able to address storage in the various environments relative to the instruction or base address and to allow for easy code relocation, three relative addressing modes called G-relative, L-relative and

Source-relative are provided. The effective address in the G-relative

mode is given by a 14-bit address field in the instruction plus the contents

of the global environment register. This permits the direct addressing of

any location in the 16K global storage area (see Fig. 6.1).

Global Environment   WORD   0

G-rel | OPCODE | 14-bit address field   +

16 K

37777B
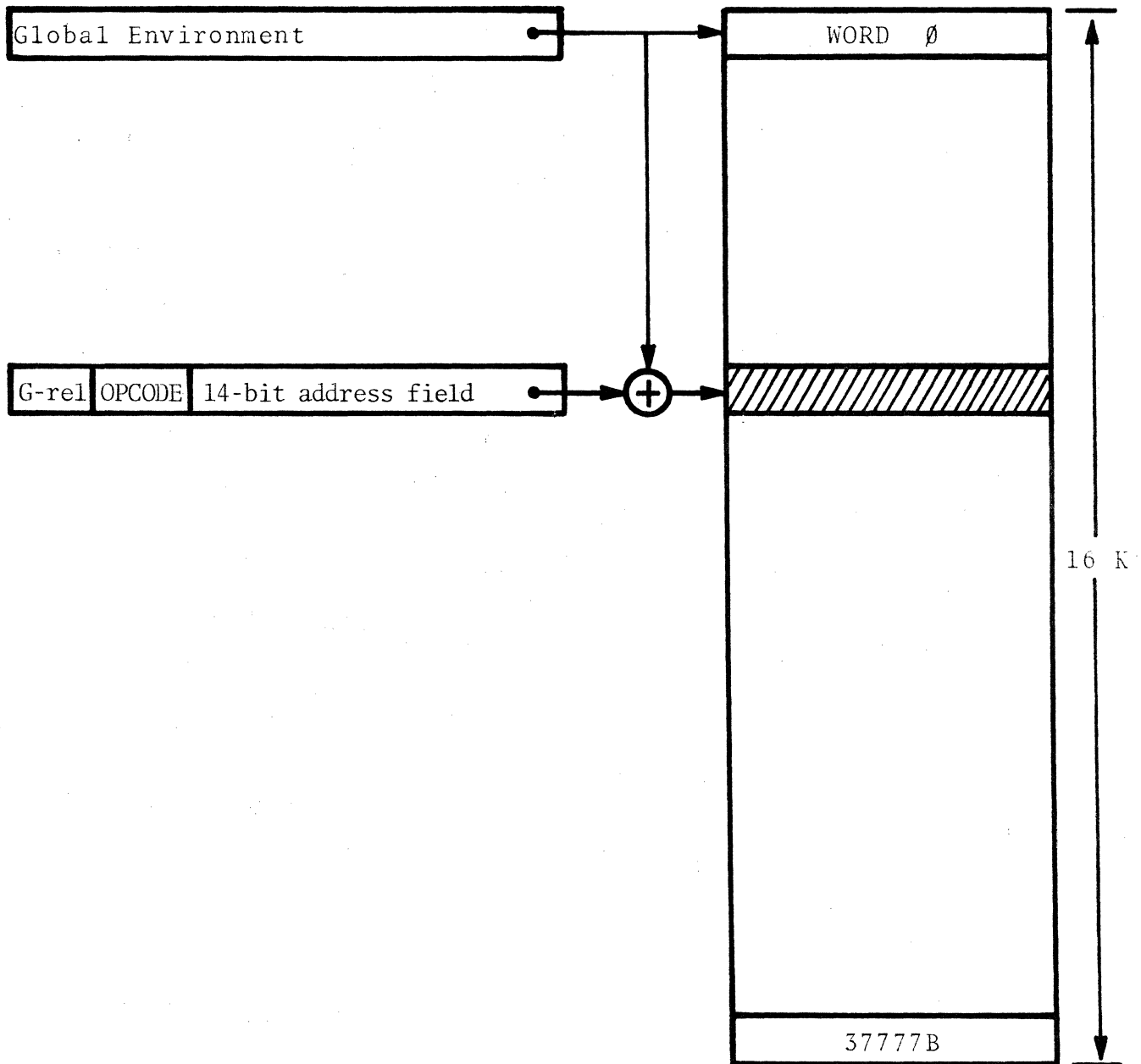
Fig. 6.1 G-Rel Addressing

The effective address in the L-relative mode is given by an 11-bit address field in the instruction plus the contents of the local environment register. This allows any location in the 2K local storage area to be addressed directly (see Fig. 6.2).
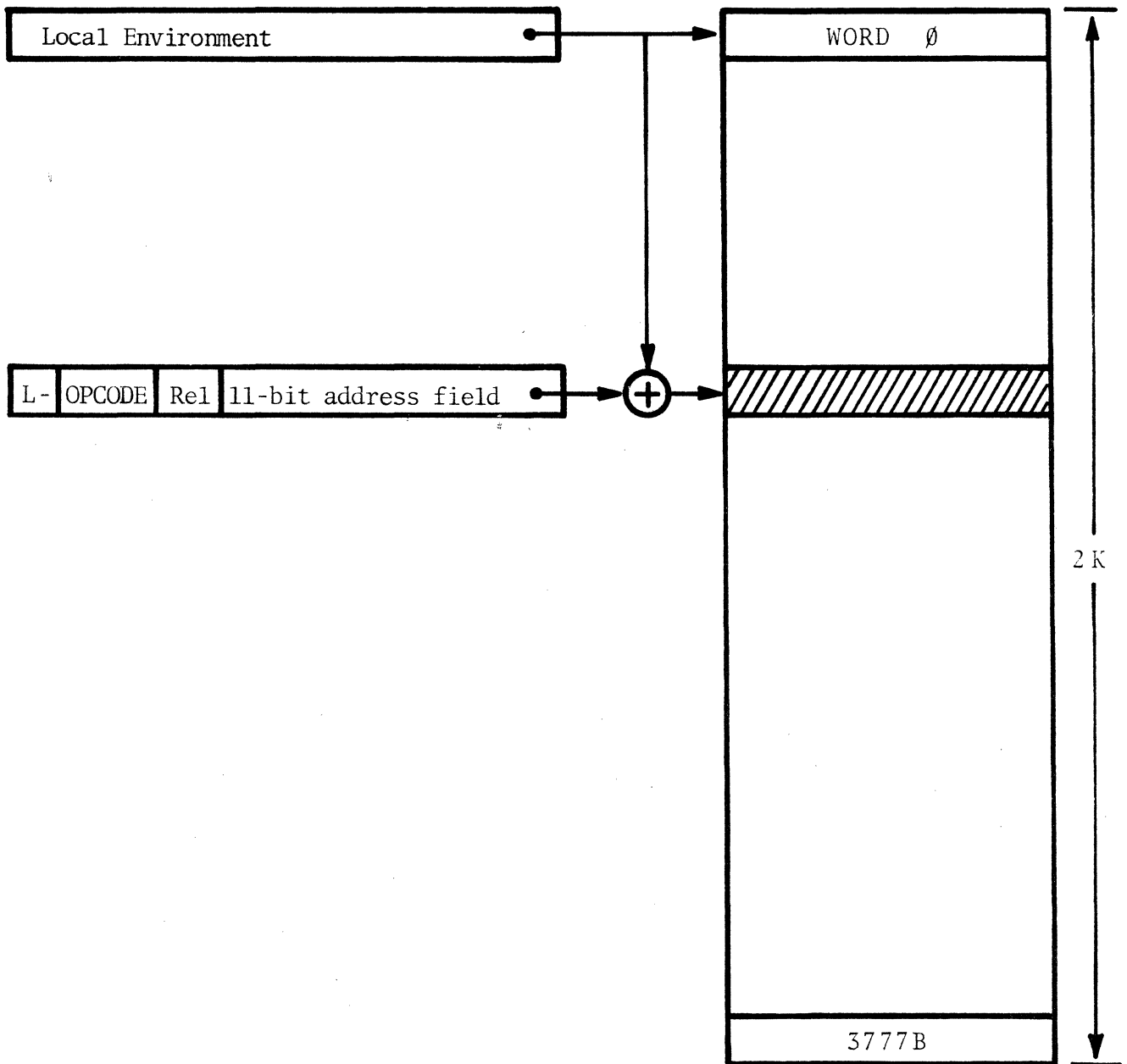
Fig. 6.2 L-Rel Addressing

The effective address in the Source-relative mode is given by the source register and the 12-bit signed address field in the instruction. This permits locations up to 2K on either side of an instruction to be addressed (see Fig. 6.3).
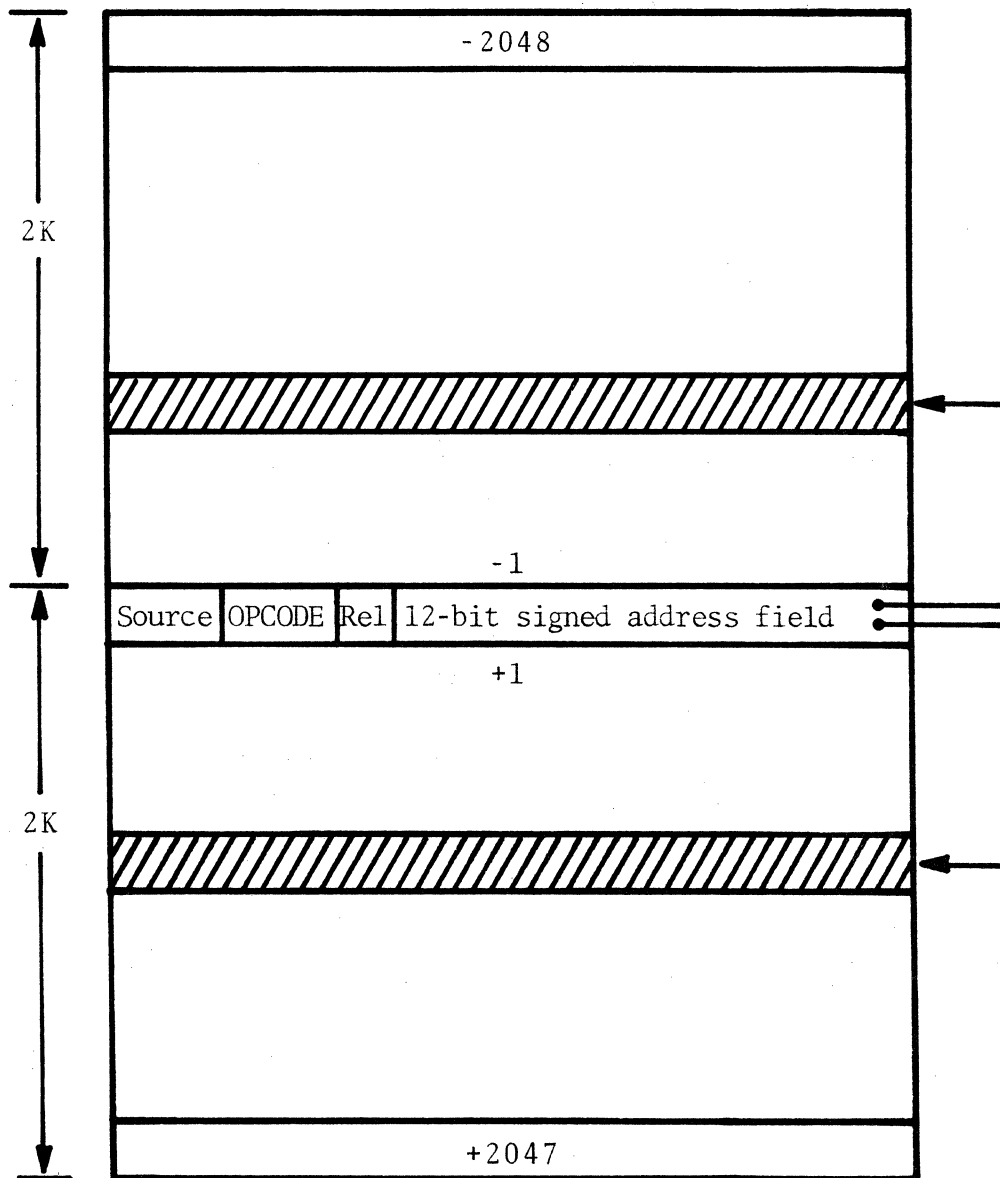
Fig. 6.3 Source-Rel Addressing

As we have seen with the field addressing example, it is desirable to be able to access data in structures such as lists, trees, and tables and other data structures common in systems code. Access to data in these structures involves obtaining a pointer to a single node or the start of the table along with a displacement to the location desired. This facility is provided by the pointer-displacement addressing mode (see Fig. 6.4).
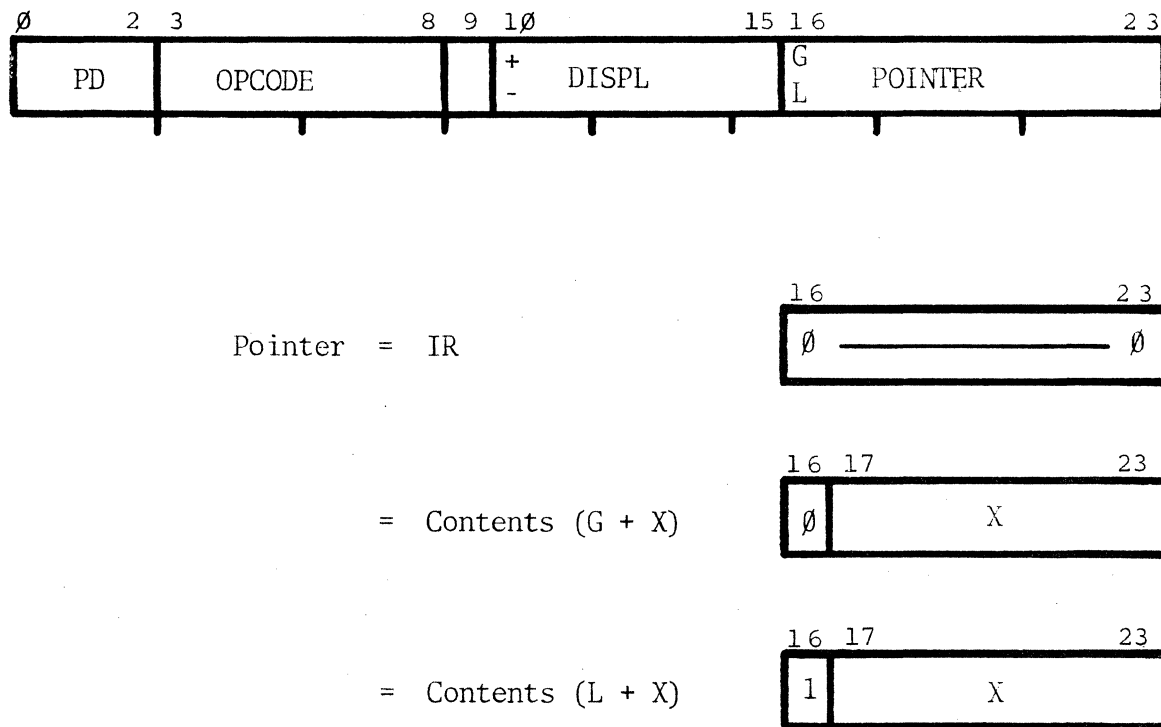
Figure 6.4 Pointer-Displacement Structure

In this mode the address field is divided into an 8-bit pointer address and a 6-bit signed displacement field. The high order bit of the pointer address field specifies the environment (1=local, 0=global) and the remaining 7-bits address a pointer in one of the first 128 words of the selected environment. If the pointer address field is zero the indexing register is used as the pointer. The effective address is simply the sum of the pointer and the displacement (see Fig. 6.5).

First 128 Words of the
Global or Local Environment

Table

Pointer

Displ   Pointer

| PD | OPCODE | | +6 | Pointer |
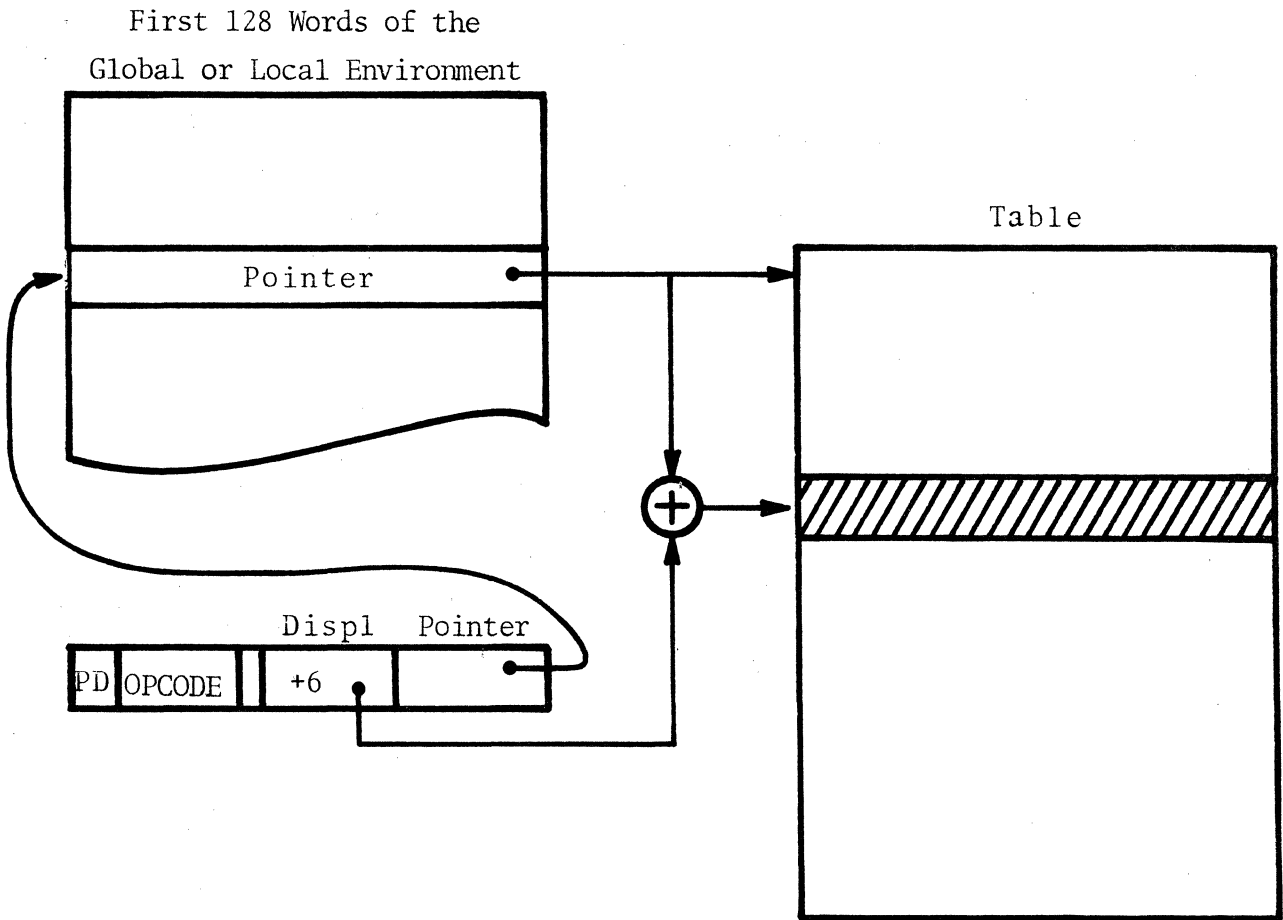
Fig. 6.5 Pointer-Displacement Addressing

Each of these addressing modes has an indirect counterpart which causes indirection through the word they address. In this case the word they address is called an indirect address word (IAW). Each IAW causes a new stage of address calculation by providing its own addressing information. An IAW can provide an 18-bit address and can point to any location in the virtual memory. An IAW specifies address modes in a manner similar to instructions, with three exceptions:

1) If the address mode is G-relative, indirect or indexed, an 18-bit absolute address is supplied and the contents of the G-register is not added;

2) If the addressing mode is L-relative, source-relative, L-relative indirect or source-relative indirect, the offsets are 3 bits longer and indexing is possible;

3) If the addressing mode is pointer-displacement or pointer-displacement indirect, the mode is taken to be read-only G-relative and read only X-relative, respectively. These behave exactly like G-relative or indexed modes except that any attempt to store will cause an error and will be trapped.

This is the normal indirect address word (IAW). We have already seen the three other types of indirect address words (field, string and array) in the descriptor sections.

To enable direct access to the entire address space an indexed addressing mode is provided. The effective address is formed by adding the contents of the X-register to the 14-bit address field in the instruction to generate an 18-bit address. Also, an instruction can contain an immediate operand field.

As we have seen, it is necessary to be able to address the various data descriptors (field, string and array) and to provide them with run-time indexing information. Two similar addressing modes called base-index (see Fig. 6.6) and base-index-displacement (see Fig. 6.7) are provided to accomplish this task.
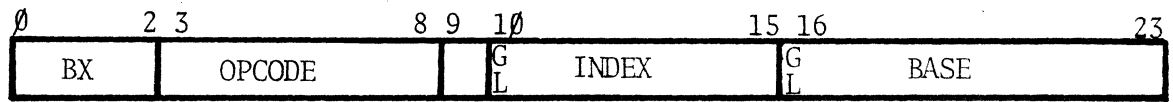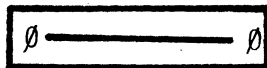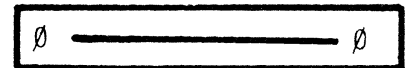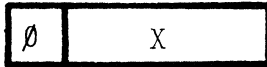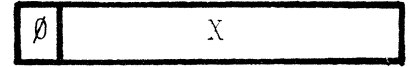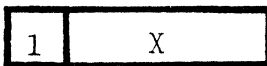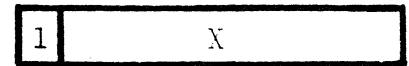
Fig. 6.6 Base-Index Structure

Ø	2	3	8	9	1Ø		15	16		23

```
   Ø        2 3              8 9  1Ø              15 16                 23
 ┌───────┬──────────────┬───┬───┬──────────────┬───┬──────────────────┐
 │  BXD  │   OPCODE     │   │ + │   D I S P L  │ G │      INDEX        │
 │       │              │   │ - │              │ L │                   │
 └───────┴──────────────┴───┴───┴──────────────┴───┴──────────────────┘
```

Index = Ø

```
 ┌──────────────────────────────────┐
 │ Ø ──────────────────────────── Ø  │
 └──────────────────────────────────┘
```

= Contents (G+X)

```
 ┌───┬──────────────────────────────┐
 │ Ø │              X               │
 └───┴──────────────────────────────┘
```
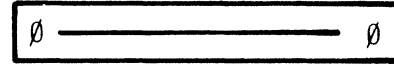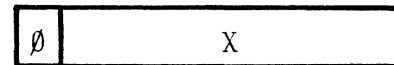
= Contents (L+X)

```
 ┌───┬──────────────────────────────┐
 │ 1 │              X               │
 └───┴──────────────────────────────┘
```
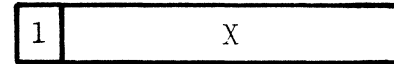
● Base ← IR

● IR ← Index + Displacement

● IA (Base)

Fig. 6.7 Base-Index-Displacement Structure

The base-index mode provides an address field that is divided into an 8-bit base field and a 6-bit index. The high order bit of each field specifies the environment (1=local, 0=global). The remaining 7-bits in the base field address a location in the first 128 words of the selected environment that in turn points to a descriptor. The remaining 5-bits in the index field address a location in the first 32 words of the selected environment that is used to initialize the indexing register. If the index field is zero, then the X-register is used to initialize the indexing register. With all these actions taken, indirection through the descriptor is caused (see Fig. 6.8). The base-index-displacement is similar except that the base is assumed to be in the Indexing Register.

First 32 Words of the
Global or Local Environment

Index

Indexing Register

Object

| BX | OPCODE | | G L | Index | G L | Base |

Field or Array IAW
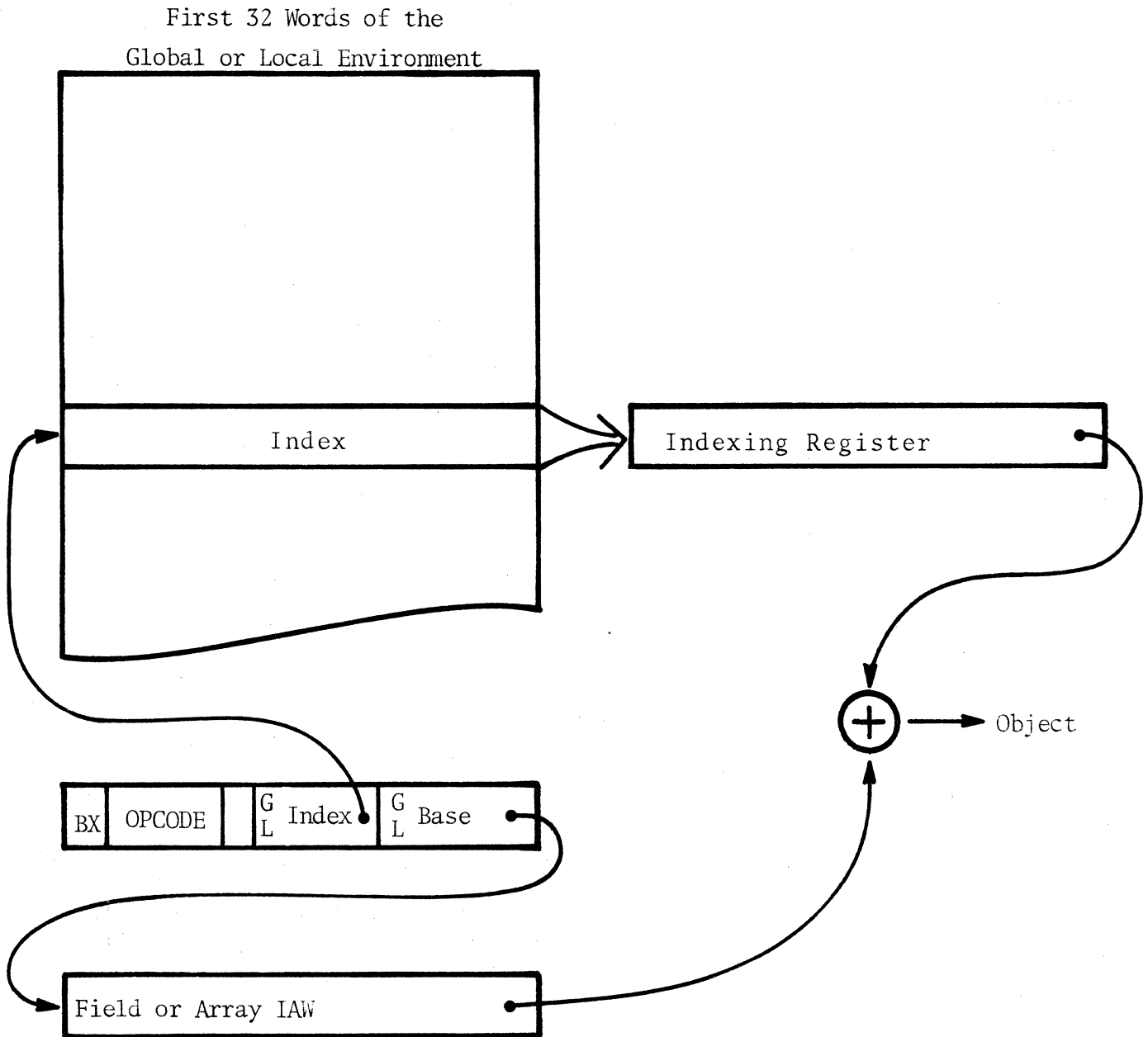
Fig. 6.8 Base-Index Addressing Example

A summary of all the addressing modes appears in Appendix I.

7. Instructions

The BCC 500 instruction set was designed to provide for easy translation of SPL operations into machine instructions. Of course SPL supports a wide variety of data types so the machine instructions generated depend on both the operation to be performed and the type of data being accessed. Nonetheless it is reasonable to illustrate at least a partial mapping of SPL operations to machine instructions as follows:

SPL operation class	Machine instruction class
Assignment	Data Transfer
Arithmetic	Arithmetic
Logical	Logical
Predicate	Test, Branch and Shift
Data Manipulation	(Handled by descriptors and addressing modes)
Control	Test and Branch

The detailed lists of SPL operations and CPU machine instructions are contained in Appendices II and III, respectively.

An instruction is formatted as follows:

0 2	3 8	9	10 23
TAG	OPCODE	P O P	Address Field
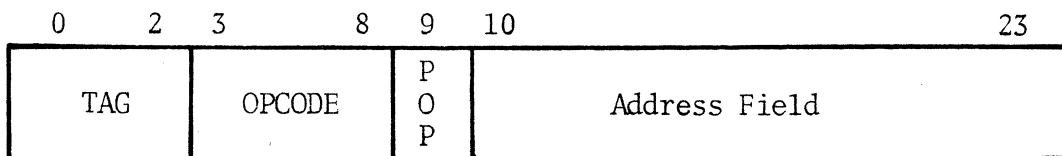
The TAG field defines the addressing mode of the instruction. The OPCODE field specifies the machine instruction. There are 61 opcodes that are defined. One opcode, called an operate (OPR) instruction provides for various register operations, special purpose operations, privileged operations and system calls.

The OPR instruction is given an immediate operand in bits 13-23. If the operand is negative, the instruction is a system call. If the operand is positive, it is decoded to determine what operation is to be done. If the POP bit is on, the instruction is interpreted as a rather peculiar kind of subroutine call rather than an ordinary machine instruction. This facility is similar to the Programmed Operator (POP) used in the XDS 940 system.[9]

8. Floating Point Features

The CPU provides for single precision (48-bit) and double precision (96-bit) floating point numbers, hardware (firmware) implemented operations, program controllable traps, a soft underflow option and five program selectable rounding modes. A single precision number is composed of an 11-bit binary exponent (numbers up to approximately 10^{300}) and a 36-bit fraction (11 decimal digits). Double precision numbers have an 84-bit fraction (25 decimal digits). A special undefined floating point number is provided for all real variables that have not been defined. This is provided to assist the programmer in debugging. The following instructions are provided by the hardware (firmware):

FLD - Floating Load

STF - Floating Store

FAD - Floating Add

FSB - Floating Subtract

FMP - Floating Multiply

FDV - Floating Divide

FCP - Floating Compare

FLX - Convert floating point to fixed and load X with result

FNA - Floating Negate

FIX - Convert floating point to fixed and load A with result

FLOAT - Convert to floating point

All floating operations have single (SP) and double (DP) precision variants, bit TDFLAG in the status register selects the mode to be used. The user's

program can elect to handle overflow, underflow and division by zero traps. The program can specify soft underflow which allows numbers to drift toward zero rather than causing an underflow trap. Finally, there is a 3-bit field in the status register that allows the program to select one of the following rounding modes.

Nearest number

Floor (toward 0)

Ceiling (away from 0)

Away from ∞

Toward ∞

9. Physical Characteristics and Operating Environment

The CPU is a 24-bit, word oriented, two's complement processor whose only task is to operate on user processes. It is implemented on a slightly modified version of the BCC microprocessor, a processor having a basic cycle time of 100 nanoseconds. The basic microprocessor design provides for inter-processor communication,[9,11] access to the central memory, and includes an arithmetic and logic unit and a control unit. There are 64 hardware testable branch conditions that allow for testing the state of various busses, registers and flip-flops and 64 special functions that are used by the microprocessor to speed up the execution of certain functions. The processor contains, in addition to a number of registers, a control store of 2K words of 90-bit read only memory and a 64 word scratchpad (200 nanosecond) memory. Modifications to the basic microprocessor for the CPU include an instruction fetch unit, which gets the next sequential instruction while the current instruction is being decoded and executed, a hardware multiplier, a set of 128 physical MAP registers, an interval timer and a compute time clock.

The CPU at any particular moment is either running a user process, switching from one user process to another or is idle until the scheduling processor assigns the CPU to a new user process. We can illustrate the general actions of the CPU with respect to the user process as follows:

IDLE: until the scheduling processor assigns a new process

THEN: clear the physical map registers

LOAD'STATE: vector from the context block of process

RUN'PROCESS: until it blocks or until a "pirate ship appears on the horizon"[12] †

SAVE'STATE: vector in the context block and go to IDLE

---

†Pirate ships rarely appear in the system, so for all practical purposes you need not worry about them causing your process to stop running. Pirate ship appearances were first reported in the character input/output processor by Paul Heckel.

The state vector for a process is composed of the following 12 elements:

Program Counter

4 Central Registers: A, B, C and D

Floating point exponent

Index register

Local environment (base) register

Global environment (base) register

Status register

Compute time clock

Interval timer

The central memory is but a portion of a hierarchical memory system.[9,11] The memory system is designed to be composed of up to 512K words of core storage, 16 million words of drum storage and 1 billion words of disk storage. This multi-level or hierarchical memory system is organized into 2048 (2K) word blocks called pages (more correctly page slots). The CPU may access information in pages only when they are in core storage. It is the memory manager's job to put pages into core storage to be used by the CPU and to remove them from core storage when the CPU is finished. Another processor called the scheduling processor is responsible for assigning a CPU to a user process. The memory management processor and the scheduling processor work together to put a process into core and wake it up[6] by assigning it a CPU. Two other tasks that the distributed operating system handles for a process are all character input/output to a terminal and file transfers to physical storage devices such as tapes, printers, etc. Basically then, the operating

system (which is distributed over several independent, concurrently running processors) provides for the services common to all users, while the CPUs service only the individual needs of each user process.

10. Virtual Machine Environment

Each programmer using SPL has access to an environment provided by the system called a user machine or a virtual machine environment.[5,6,7] The virtual machine is composed of a set of virtual operations and a virtual memory structure. The set of operations includes all the user operations provided by the physical CPU and all the services provided by the monitor or utility portion* of the operating system. The virtual memory structure for a user is defined by a directory that connects a user to all objects in the system he can access. An object is one of the following whatnots:[†]

- File

- Process

- Resource allocation

- Access keys

- Free object

Free objects are simply present to allow the system to be open-ended. Files and processes are the basic objects that the user performs actions on and with, respectively. Resource allocations provide the user with the ability to control various resources such as response time, number of terminal lines, etc., while the access keys allow for protection of various objects. Basically, the virtual memory structure for a user consists of all the pages in the set of objects he can access (see Fig. 10.1).

---

*Note that the variability of the utility allows the system to support different virtual machines.
†"Whatnots" were first proposed by Jack Freeman and are simply a "something or other."

VIRTUAL MEMORY STRUCTURE FOR A USER

```
┌──────────────────────────────────────────────────────────────┐
│ All pages accessible to the user:  This includes files and processes. │
│                                                                │
│      ┌──────────────────────────────────────────────┐        │
│      │ Pages in a single process                     │        │
│      │     ┌──────────────────────────────────┐      │        │
│      │     │ Pages in a drum working set       │      │        │
│      │     │   ┌──────────────────────────┐    │      │        │
│      │     │   │ Pages in core working set │    │      │        │
│      │     │ ┌─────────────────────────┐  │    │      │        │
│      │     │ │░░VIRTUAL ADDRESS SPACE░░░│  │    │      │        │
│      │     │ └─────────────────────────┘  │    │      │        │
│      │     │   └──────────────────────────┘    │      │        │
│      │     └──────────────────────────────────┘      │        │
│      └──────────────────────────────────────────────┘        │
│                                                                │
└──────────────────────────────────────────────────────────────┘
```
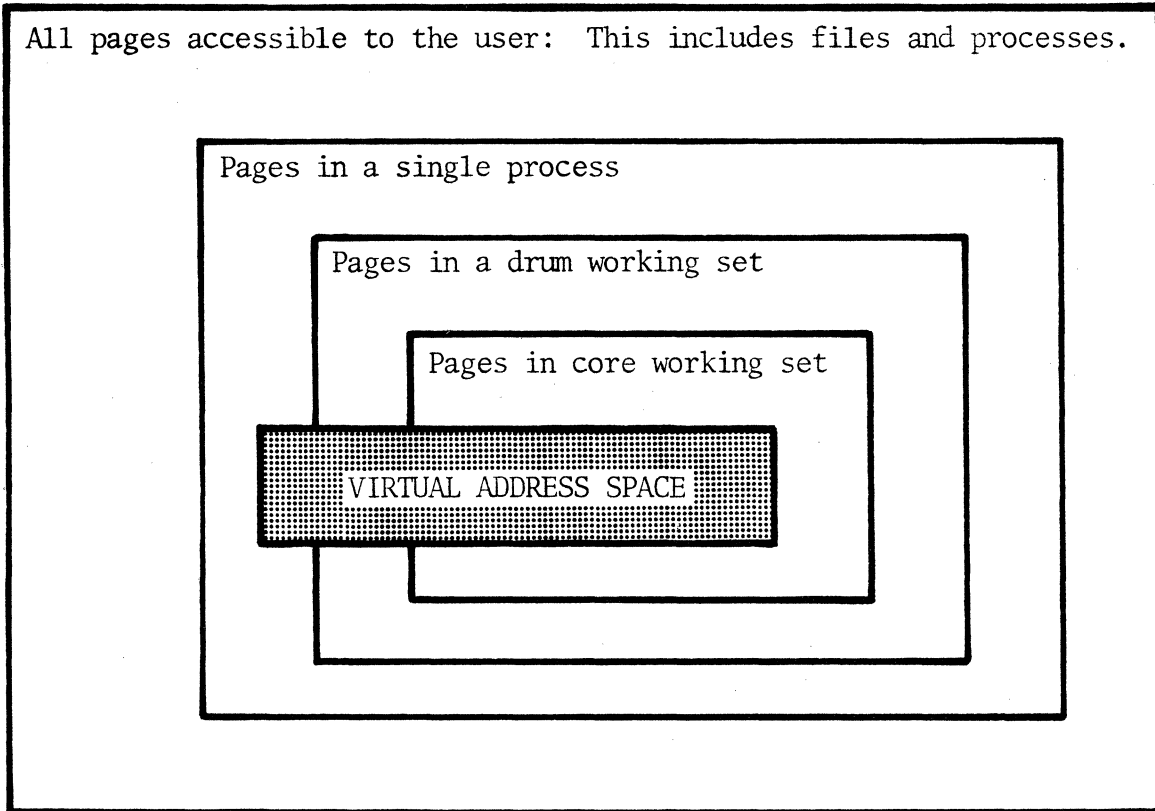
Fig. 10.1 Virtual Memory Structure for a User

All the pages a process can reference are named in the process memory table in the context block of the process. A process calls on the services provided by the monitor either directly or indirectly through an individualized utility to get more pages from a file to the process memory or to create new pages. Each and every page that is created is given a 48-bit location-independent or unique name. No two pages will ever have the same name and that unique name is used to reference the page wherever it may be located in the multi-level physical

memory system. Each active process has direct access to a memory of

256K words called its virtual address space which is organized into 128 (2K

word) pages and is logically divided into a user, utility and a monitor area.

These three areas are protected from one another (see Fig. 10.2) and may be

conceptualized as a ring structure. The user ring is considered to be the

VIRTUAL ADDRESS SPACE
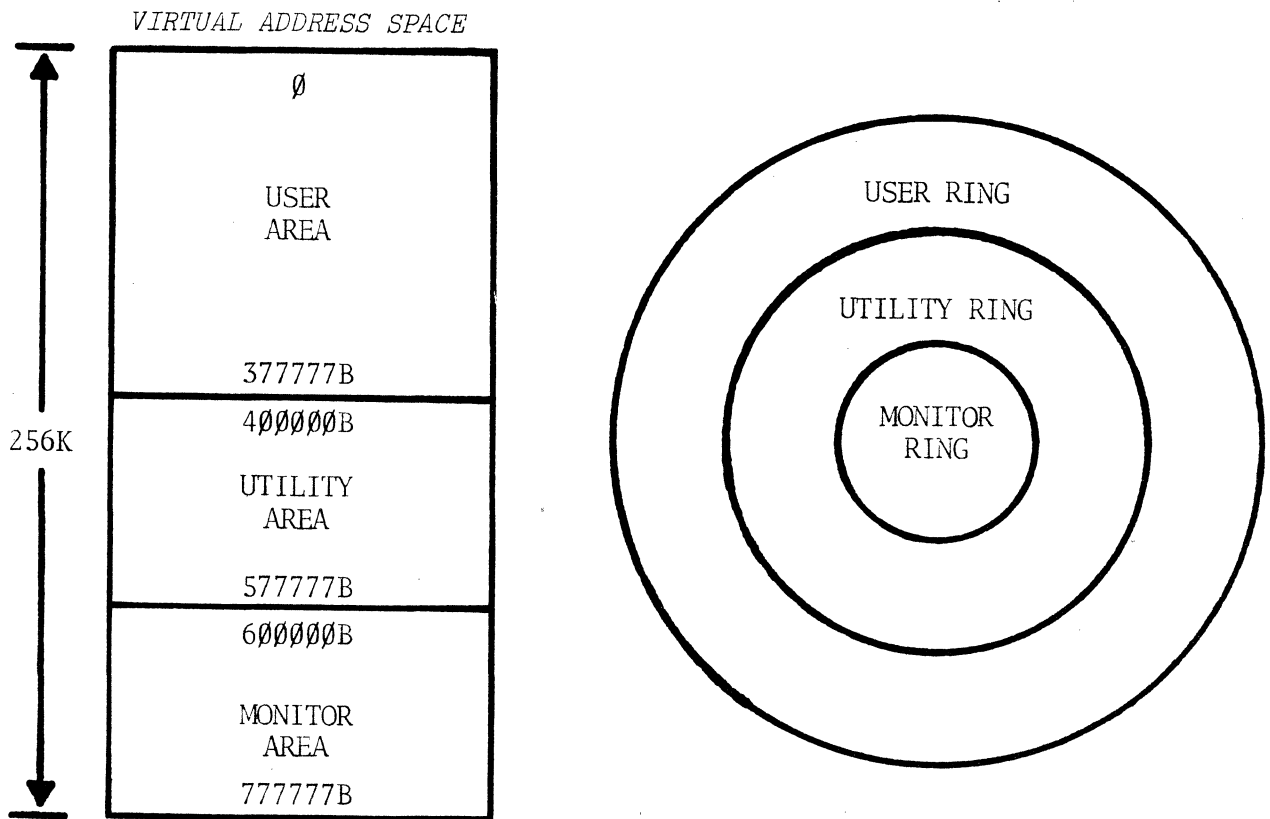
Fig. 10.2 Virtual Address Space and Protection Rings

lowest ring and the monitor ring the highest. Services provided by either

the monitor or the utility may only be accessed through protected entry system

calls. Any other references from a lower ring to a high ring are illegal

and cause a memory access trap.

In order to increase the efficiency of a process a user can specify that frequently accessed pages in the virtual address space be assigned to core when the process is active. These pages are the so called "core working set" of the process. Pages that are accessed less frequently may be assigned to the "drum working set" of the process.

The system does not practice demand paging.[†]

[†]But it is not bad at it even though it doesn't practice.

11. Mapping Facilities

Every reference a process makes to an address in the virtual address space is mapped into a physical address in core storage. A reference into the virtual address space consists of an 18-bit address composed of a virtual page number (the top 7-bits) and a word number (the low order 11-bits). THe CPU maps this 7-bit virtual page number into a physical page number in a real core of up to 256 pages. First the CPU uses the process map (which defines the virtual address space) and the process memory table (which contains the "unique names" of every page known to the process) to translate the virtual page number into a location-independent name. Now, since the CPU is only able to directly address information in core it must determine if the page is in core. To do this the CPU references a system table that contains a list of all pages in core. If the desired page is in core this table, called the core hash table, supplies an 8-bit physical page number which locates the page in core. Appended to this 8-bit page number is the 11-bit word number providing CPU with the 19-bit physical address it needs to reference the desired word in a core storage that can contain up to 512K words. If the desired page is not in core, the process is blocked and the CPU is assigned to a new process. The memory manager will insure that the next time the original process becomes active the desired page will be in core.

Every memory reference a process makes then requires a mapping from:

virtual page number → location-independent name

location-independent name → physical page number

This mapping process facilitates the checking needed to ensure that a virtual

reference does not address a page that does not exist and that the page is in core. Once this mapping and checking process has been accomplished for a particular virtual page it is possible to simply map from virtual page number to physical page number. This facility is provided by a hardware map that contains 128 registers. The hardware map is cleared each time the CPU is assigned to a new process. When a virtual page is referenced the mapping function loads the physical page number into the hardware map register which corresponds to the virtual page number.

The various mechanisms for performing the mapping will now be described in detail. First, we will describe the data structures and hardware registers used and then the mapping process performed by the CPU for each reference to the virtual address space. It is convenient to consider the mapping process performed by the CPU as being composed of a mapping function, a hardware map and a hardware map loader. The mapping function together with the tables that support it provide all the mechanism necessary to perform the mapping. The hardware map facilitates rapid access to pages once they have been mapped and the hardware map loader's function is to load these registers.

Two tables in the context block of the active process provide the CPU with all the information it needs to translate the virtual page number into a location-independent name. These two tables are called the process map and the process memory table. The process map defines the virtual address space for the process and is composed of 128 12-bit entries (see Fig. 11.1). Each

```
 0    1 - 3   4                                 11
┌──┬────────┬───────────────────────────────────┐
│R │        │                                   │
│  │   ⨉    │              PMTI                 │
│O │        │                                   │
└──┴────────┴───────────────────────────────────┘
```
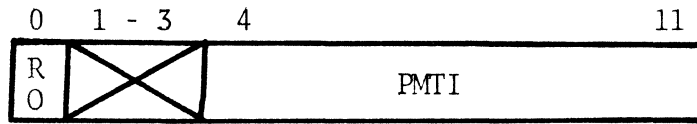
RO - Read Only Bit

PMTI - Process Memory Table Index

Fig. 11.1 Map Entry

entry of the process contains a read-only (RO) bit and an index to an entry in the process memory table or is empty (i.e., its value is zero) indicating that a particular virtual page is not being used. If the read-only bit is set the corresponding page may not be modified. The process memory table contains a list of all the pages that the process can reference. Currently this table contains 128 entries, but is expandable to 255 entries. Each entry is 4 words long (see Fig. 11.2) and contains the following information

```
┌───────────────────────────────────────────────┐
│                                               │
│──              UNIQUE NAME                  ──│
│                                               │
├──────────┬────────────────────────────────────┤
│   ⨉      │          Disk Address              │
├──┬──────┬┴────────────┬───┬───────────────────┤
│R │ P    │             │ S │                   │
│  │ R    │             │   │                   │
│O │ E  F │             │ F │                   │
└──┴──────┴─────────────┴───┴───────────────────┘
```
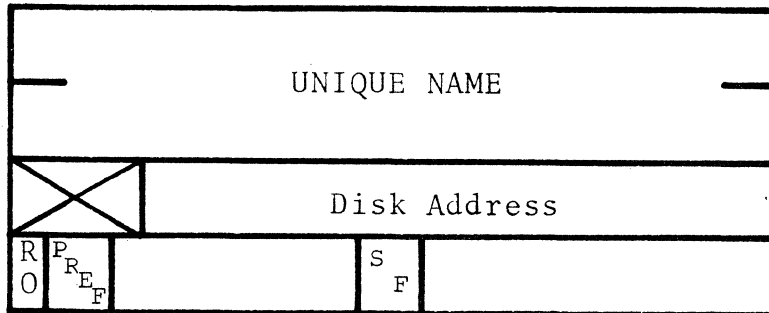
PREF - Page has been referenced flag

RO - Read only flag

SF - Page is scheduled for the process

Fig. 11.2 Process Memory Table Entry

UNIQUE NAME: The location-independent name for the page.

DISK ADDRESS: The address at which the disk copy of the page is stored.

READ-ONLY FLAG: This flag is set for read-only pages by the basic file system when a process places file pages in the process memory table.

REFERENCED FLAG: The CPU's hardware map loader sets this flag whenever it loads the associated page into its map, thus providing an indication as to how frequently the page is referenced.

SCHEDULED FLAG: The memory management system sets this bit if a process is authorized to access this particular page. That is, the page is in the core working set of the process.

Information about the current contents of core storage is maintained in a core resident table called a core hash table. The table is composed of a set of 256 index elements and a list of entries.

The index elements, called CHT1, are an array of 256 pointers to lists of CHT entries. Each index element is either an end marker or contains a pointer to an entry α with the property that HASH(UN(α)) is the address of the index element. If there are several pages in the core hash table with the same value of HASH(UN), the index points to one entry, which points to the next entry using a collision pointer field, and so on until all are chained onto the list. The last entry in the list has an end flag in its collision pointer field. The hashing function HASH is to take the exclusive or of the 6 8-bit bytes of the unique name (UN) of the page and then the exclusive or of this result with 264B.

The core hash table entries are contained in an array which has one entry per page of real core. This array of entries is called CHT2. The format of

an entry is given in Fig. 11.3.

```
┌─────────────────────────────────────────────┐
│                                               │
│─              UNIQUE NAME                  ─ │
│                                               │
├──────┬────────────────────────────────────────┤
│ ╳    │          Disk Address                  │
├──┬──┬┼──────────────────┬──────────────────────┤
│D │U │╳ │   Core Page    │        SCHED         │
│B │  │  │    Number      │                      │
├──┴──┴──┼──────────────────────────────────────┤
│   ╳    │     Free Core List Pointer           │
├────────┼──────────────────────────────────────┤
│   ╳    │        Collision Pointer             │
└────────┴──────────────────────────────────────┘
```
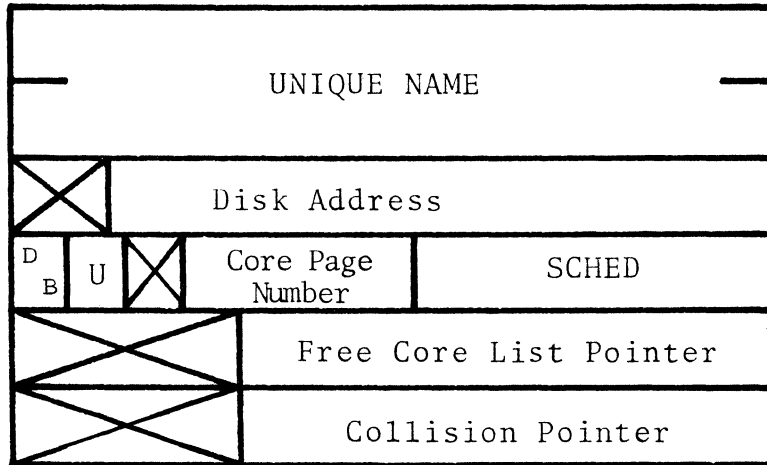
DB - Dirty Bit

U - Unavailable Bit

SCHED - Number of occurrences of this page in loaded working sets

Fig. 11.3 Core HASH Table Entry

Each entry is six words long and contains the following information:

- The unique name of the page;

- The disk address of the page;

- A dirty bit which is set if the page in core is potentially different from the copy on the drum. That is, a store into the page has occurred.

- An unavailable bit that prevents CPU access to the page when it is set.

- Core page number. This is also an index into CHT2.

- The scheduled count which gives the number of occurrences of this page in loaded working sets.

The hardware map is composed of 128 11-bit registers, one register for each of the 128 pages in the virtual address space. Each register contains an empty flag which is set if the register has not been loaded, a dirty bit which is set if the page is modified, a read-only bit and an 8-bit real page number of a page in a core storage of up to 256 pages.

We now describe in detail the actions the CPU takes for each and every virtual memory reference. These actions and the checking they support are outlined in Fig. 11.4 and it will be useful to refer to this figure while reading this description.

When a process becomes active by being assigned a CPU, the empty flag is set in each of the 128 hardware registers of that CPU. Each and every address generated by a program in the process must be mapped to convert it from a virtual address to a real address in core storage. This is done by extracting the virtual page number (top 7-bits) from the 18-bit virtual address and using it to index one of the 128 hardware map registers.

If the empty flag of the selected hardware map register is off then the remainder of the register is returned. The physical page number (8-bits) is prefixed to the word number (last 11-bits) of the virtual address to make a 19-bit real address. If the read-only flag is on and the access is a store, the store is not allowed and "Read-only trap" is caused. If the read-only flag is off, the dirty bit is off and the access is a store, the dirty bits in the core hash table entry for the page and in the hardware map are set on. The read-only flag is saved.

If the empty flag is on, the CPU must execute its mapping function and will load the hardware map when finished. In this case the virtual page number is used to index an entry in the process map. If this entry is zero,

MAPPING PROCESS

PAGE NO. → LOCATION-INDEPENDENT NAME | LOCATION-INDEPENDENT NAME → PHYSICAL PAGE NO.

VIRTUAL ADDRESS (18-BIT)

| 7-BIT PAGE NO. | 11-BIT WORD NO. |

MAP

PROCESS
MEMORY TABLE

PAGE-NOT-IN-CORE

UNIQUE NAME

HASHING
ALGORITHM

| RO | REF | |

CORE HASH TABLE

| RO | PMT INDEX | |
| EMPTY |

UNIQUE NAME

DISK ADDRESS

PAGE NO.

PAGE-NOT-IN-MAP

| 8- BIT PAGE NO. | 11-BIT WORD NO. |

REAL ADDRESS (19-BIT)

-54-
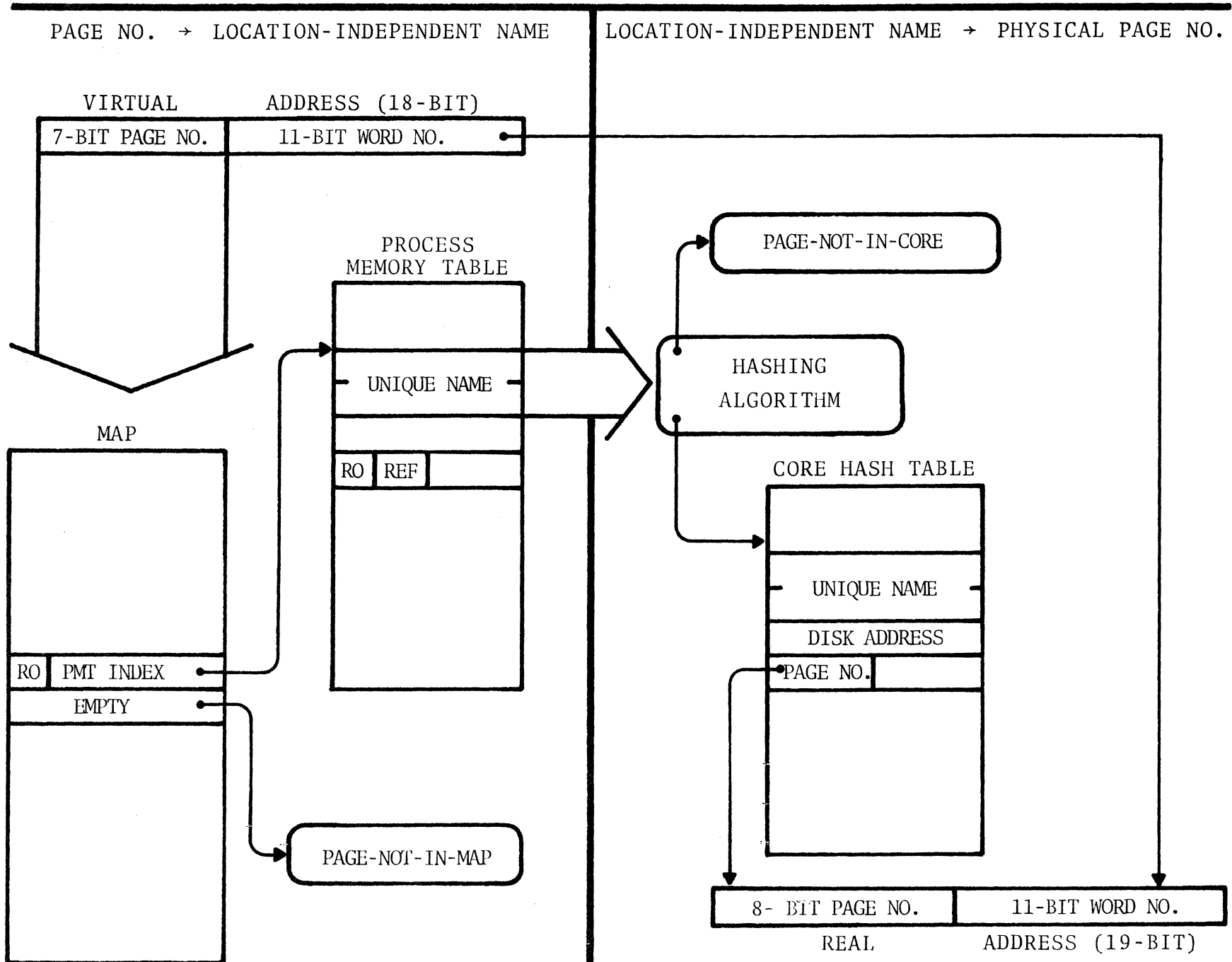
Fig. 11.4 Mapping Process

the page does not exist and a "page-not-in-map trap" is caused.

If the entry is not zero, the index into the process memory table is extracted. The process memory table entry specified is accessed. If the scheduled flag is off, the referenced page is not in the core working set and a "page-not-in-core trap" is generated. The read-only flag is saved so it may be merged with the read-only flag from the map and loaded into the associated hardware map register. The referenced flag is set on. The unique name is extracted from the process memory table entry. The core hash table is searched using the HASH(UN) function. If the page is not in the core hash table (this condition should not happen, but is checked for anyway) then the memory manager made an error and a "page-not-in-core trap" is caused and the process is blocked. Otherwise an 8-bit page number is supplied by the core hash table entry and appended to the top of the 11-bit word number to provide an address in core. The 8-bit page number is also loaded into the appropriate physical map register.

References

[1] "Operational Characteristics of the Processors for the Burroughs B5000," Burroughs Corp., Detroit, Mich., (Sept. 1961).

[2] Corbató, F. J., Daggett, M. M. and Daley, R. C., "An Experimental Time Sharing System," AFIPS Conference Proceedings, Vol. 21, (1962, FJCC), pp. 335-344.

[3] Corbató, F. J. and Vyssotsky, V. A., "Introduction and Overview of the Multics System," AFIPS Conference Proceedings, Vol. 27, (1965, FJCC), pp. 185-194.

[4] Iliffe, J. K. and Jodeit, J. G., "Dynamic Storage Allocation," Computer Journal, Vol. 5, (Oct. 1962), p. 200.

[5] Lampson, B. W., "Dynamic Protection Structures," AFIPS Conference Proceedings, Vol. 35, (1969, FJCC), pp. 27-35.

[6] Lampson, B. W., "A Scheduling Philosophy for Multi-Processing Systems," CACM, Vol. 11, No. 5, (May 1968), p. 347.

[7] Lampson, B. W., Lichtenberger, W. W., and Pirtle, M. W., "A User Machine in a Time-Sharing System," Proc. IEEE, Vol. 54, No. 12, (Dec. 1966), pp. 1766-1774.

[8] Lampson, B. W., "Scheduling and Protection in Interactive Multi-Processor Systems," Thesis, Project Genie Document No. P-11, (Jan. 1967).

[9] Lichtenberger, W. W., and Pirtle, M. W., "A Facility for Experimentation in Man-Machine Interaction," AFIPS Conference Proceedings, Vol. 27, (1965, FJCC), pp. 589-598.

[10] Opler, A., "Fourth Generation Software," Datamation, Vol. 13, (Jan. 1967), pp. 22-24.

[11] Pirtle, M. W., "Intercommunication of Processors and Memory," AFIPS Conference Proceedings, Vol. 31, (1967, FJCC), pp. 621-634.

[12] Snoopy, "It Was a Dark and Stormy Night--Part I," Snoopy's Publisher, N. Y., 1970.

APPENDIX

I.   ADDRESSING MODES

Notation used in defining addressing modes.

W[i,j]   means bits i to j of W (the address field of
                      the instruction) considered as a 24-bit number.
                      W[i,i] is represented by W[i].

CONTENTS(N)   means the contents of the memory location with
                      address N. Ring checking is performed with
                      R as source and N as target.

IA(N)   means that the indirect addressing sequence
                      is initiated by:

                           FUNCTION IA(N);
                              IAW ← CONTENTS(N);
                              R ← N;
                           *PROCEED TO PROCESS IAW

                      By the time it is finished, the IA function
                      will set the value of the address (Q) or the
                      operand (OP).

All instructions start with IR ← XR & R ← P;

Addressing Modes (continued)

Abbr	Name	Notation	Address Computation
D	DIRECT	OPC G'[W];	Q ← W + G; OP ← CONTENTS(Q);
I	INDIRECT	OPC $G'[W];	IA(W + G);
X	INDEXED	OPC X'[W];	Q ← W + IR; OP ← CONTENTS(Q);
PD	Pointer-Displacement	OPC P[D];	PTR ← IR IF W[16,23] = Ø ELSE PTR ← CONTENTS(G + W[17,23]) IF W[16] = Ø ELSE PTR ← CONTENTS(L + W[17,23]); DISP ← SIGNED(W[1Ø,15]); Q ← PTR + DISP; OP ← CONTENTS(Q);
PDI	Pointer-Displacement Indirect	OPC $P[D];	Q ← PTR + DISP; * AS FOR PD MODE IA(Q);
BX	Base-Index	OPC B[I];'	BASE ← IR IF W[16,23] = Ø ELSE BASE ← G + W[17,23] IF W[16] = Ø ELSE BASE ← L + W[17,23]; IR ← IR IF W[1Ø,15] = Ø ELSE IR ← CONTENTS(G + W[11,15]) IF W[1Ø] = Ø ELSE IR ← CONTENTS(L + W[11,15]); IA(BASE);
BXD	Base-Index-Displacement	OPC ($X')[I+D];	BASE ← IR; INDEX ← Ø IF W[16,23] = Ø ELSE INDEX ← CONTENTS(G + W[17,23]) IF W[16] = Ø ELSE INDEX ← CONTENTS(L + W[17,23]); DISP ← SIGNED(W[1Ø,15]); IR ← INDEX + DISP; IA(BASE);

Revision 3/4/74

Addressing Modes (continued)

Abbr	Name	Notation	Address Computation
LR	L-Relative	OPC L'[D];	DISP ← W[13,23]; Q ← L + DISP; OP ← CONTENTS(Q);
LRI	L-Relative-Indirect.	OPC $L'[D];	DISP ← W[13,23]; Q ← L + DISP; IA(Q);
	Field	SE(2), SIZE(3,7), FB(8,12), DISP(13,23)	Q ← IR + DISP; U ← CONTENTS(Q); OP ← U[FB,FB+SIZE-1]; OP ← OP - 2**(24-FB) IF SE = 1 AND OP[FB,FB] = 1;
	String	CSIZE(2,3), CPOS(4,5), WA(6,23)	Select byte CPOS of CSIZE from word WA of string.
	Array	LB(2), ATRAP(3), LEB(4), MULTS(5,6), MULTL(5,1∅), UBS(7,23), UBL(11,23)	TRAP'ABE(R) IF IR < LB; IATRP(R) IF (ATRAP=1) AND (INSTR≠LAX); IATRP(R) IF (ATRAP=∅) AND (INSTR=LAX); IF LEB = ∅ DO; TRAP'ABE(R) IF IR > UBS; IR ← (IR-LB) * (MULTS+1); ELSE DO; TRAP'ABE(R) IF IR > UBL; IR ← (IR-LB) * (MULTL+1); ENDIF; T ← R + 1; NORMAL'IA(T);

Appendix II: SPL Operations

Arithmetic: + - * / ** (exponentiation)

                      - (negation) MOD

Logical: A' (bitwise AND)   V' (bitwise OR)

                      E' (bitwise exclusive OR)

                      N' (1's complement)

                      LSH RSH (logical shifts)

                      LCY RCY (cyclic shifts)

Predicate: = # < > >= <=

                      AND OR NOT

Data Manipulation:[ ] (subscripting)

                      . $ (field operations)

                      $ (unary, indirection)

                      @ (pointer)

                      @ (binary, field placement)

Control: GOTO

                      RETURN, FRETURN

                      & WHERE

                      IF ELSE

                      FOR WHILE

Appendix III: Machine Instructions

A. Data Transfer

    LDA - Load A register

    LDB - Load B register

    LDX - Load X register

    LDD - Load double

    EAX - Effective address to X

    LAX - Load Array index

    LNX - Load Negative to X

    STA - Store A register

    STB - Store B register

    STX - Store X register

    STD - Store Double

    XMA - Exchange memory and A

B. Integer Arithmetic

    ADD - Add memory to A register

    SUB - Subtract memory from A register

    ADC - Add memory and carry to A register

    SUC - Subtract memory from A register + carry

    MIN - Memory increment

    MDC - Memory decrement

    ADM - Add to memory

    ADX - Add to X

    MUL - Multiply memory and A register

    DIV - Divide memory into A and B registers

C. Test

   ICP - Integer compare A register and memory

   CPZ - Compare A register with zero

   CMZ - Compare A register AND memory with zero

   ISD - Increment string descriptor

   DSD - Decrement string descriptor

D. Logical

   ETR - AND A register and memory

   IOR - OR A register and memory

   EOR - Exclusive OR A register and memory

E. Shift

   ASHD - Arithmetic shift double (A and B register)

   ASHA - Arithmetic shift A register

   LSHD - Logical shift double

   LSHA - Logical shift A register

   CYD - Cycle double

   CYA - Cycle A register

F. Branch

   BRU - Branch unconditionally

   BLT - Branch on result less than zero

   BLE - Branch on result less than or equal to zero

   BEQ - Branch on result equal to zero

   BNE - Branch on result not equal to zero

   BGE - Branch on result greater than or equal to zero

   BGT - Branch on result greater than zero

BRX - Branch on index less than zero

BSX - Branch and set index register to p-counter + 1

BLL - Branch and load the local environment register

G. Miscellaneous

HLT - Halt, causes the TI trap

EXU - Execute

EAC - Effective address computation

SRS - Set or reset status bits

TSB - Test status bits

H. Opr

If the OPR operand is negative, the instruction is a system call, otherwise it is decoded as one of the following:

CAB - Copy A to B

XAB - Exchange A and B

CBA - Copy B to A

CBX - Copy B to X

XXB - Exchange X and B

CXB - Copy X to B

CAX - Copy A to X

XXA - Exchange X and A

CXA - Copy X to A

CNA - Negate A

CNX - Negate X

ZOA - Clear A

ZAB - Clear A and B

ZOB - Clear B

CGA - Copy G to A

XGA - Exchange G and A

CLA - Copy L to A

XLA - Exchange L and A

CSA - Copy SR to A

XSA - Exchange SR and A

CTA - Copy interval timer to A

CCA - Copy Compute time clock to A

NOP - No operation

MVB - Move block

MVC - Move constant

MVS - Move string

CPS - Compare string

CLS - Compute length of string

ASP - Add to string pointer

LLT - Locate leading transition

COB - Count one bits

LOADS - Load state

STORS - Store state

LSC - Load string constant

*The following OPR's are privileged and may only be executed in the monitor ring.

     SLOK - Set CPU lock

     RLOK - Reset CPU lock

      ALD - Absolute load A

      AST - Absolute store A

      AAX - Absolute address to X

      PRO - Protect

   UNPRO - Unprotect

     ATTN - Attention

     USCL - Micro-scheduler call

     CMAP - Clear physical map

      CAT - Copy A to interval timer

      CAC - Copy A to compute time clock

      RUN - Read Unique Name

I. Floating point instructions and OPR's

      FAD - Floating Add

      FMP - Floating multiply

      FDV - Floating divide

      FCP - Floating compare

      FLX - Fix and load X

      FNA - Floating negate

      FIX - (OPR) Similar to FLX, but operand taken from floating point accumulator and put in A-register

    FLOAT - (OPR) Produces floating point number from fixed in A-register

FLD - Floating load

STF - Floating store

IV. SPL PROGRAM TO DEFINE BLL

```
*   SPL PROGRAM TO DEFINE BLL

BLL:    N←Ø; SPEC←Ø; MCAL←Ø; NEWG←G; GOTO BLL1;
BLLN:   N←1; SPEC←Ø; MCAL←Ø; NEWG←G; GOTO BLL1;

*   OPR WITH NEGATIVE OPERAND:
OPR:    OP← -OP;
        N←OP $ BIT15; SPEC←Ø;
        MCAL←OP $ BIT14+1;
        (NEWG←4ØØØØØB & R←4ØØØ14B) IF MCAL=1 ELSE
        (NEWG←6ØØØØØB & R←6Ø4ØØØB);
        IR←OP $ BIT16THRU23; IA(R); GOTO BLL1;
*
POP:    POPW←CONTENTS(P); IR←POPW $ FOPC; N←Ø
        SPEC←1; MCAL←Ø; NEWG←G;
        IA(G); TI() IF IMMEDIATE=1; GOTO BLL1;
*
BLL1:   NEWPW←CONTENTS(Q);
        BLLERR(1) IF NEWPW $ BIT5;
        NEWP←(NEWPW $ FLW IF NEWPW $ BIT4=Ø
            ELSE Q+NEWPW $ FSRW);
        BRD←CONTENTS(Q+1) FTNATF←Ø;
        CLL←BRD $ BITØ; STK←BRD $ BIT1;
        CPA←BRD $ BIT2;
        CPR←BRD $ BIT3 IF CLL=1 ELSE UWSTK←BRD $ BIT3;
        REL←BRD $ BIT4; FTN←BRD $ BIT5;
        NEWL←E←BRD $ FE;
        IF RING(NEWP)<RING(P) DO;
            NEWG←G[14]; RET←1;
        ENDIF;
*
*   OBTAIN NEW LOCAL ENVIRONMENT
*
        IF STK=1 DO;
            IF CLL=Ø DO;
                IF UWSTK=Ø; SP←L;
                ELSE DO; SP←E; NEWL←E·FE;
                ENDIF;
            ELSE DO;
                SP←NEWG[2]+E; STKOV() IF SP>=NEWG[3];
                NEWL←NEWG[2];
            ENDIF;
        ELSE DO;
            NEWL←L IF NEWL=Ø;
        ENDIF;
*
        RINGCHECK(NEWP);
```

```
*
*    COPY ARGUMENTS
*
        BLLERR(2) IF N=CPA;
        NAW←P+1;
        IF CPA#∅ DO;
            FOR NFW←NEWP BY 1 DO;
                R←NEWP; FP←CONTENTS(NFW);
                FTYPE←FP $ TYPE;
                IF SPEC=1 DO;
                    SPEC←∅; AP←POPW; NAW←NAW-1;
                    ATYPE←FTYPE; ASTR←FP $ FSTR; AENDF←FP $ ENDF;
                ELSE DO;
LØ:                 R←P; AP←CONTENTS(NAW);
                    ATYPE←AP $ TYPE; ASTR←AP $ STR;
                    AENDF←AP $ ENDF;
                ENDIF;
                IF ATYPE=∅ DO;
*    JUMP IN ACTUAL ARGUMENT LIST
                    R←P; IR←XR; EA(NAW);
                    BLLERR(5) IF IMMEDIATE;
                    NAW←Q;
                    GOTO LØ;
                ELSE DO;
                    BLLERR(2) IF AENDF#FP $ ENDF;
                    IF ATYPE#FTYPE DO;
*    TYPES DISAGREE.  ERROR UNLESS ONE IS JOKER, JOKER IS CHECKED
*    FOR BELOW UNLESS CADDR=1 OR FSTR=ARRAY, IN WHICH CASE IT IS
*    NOT CHECKED.
                        IF ATYPE#14 DO;
                            BLLERR(3) IF FTYPE#14;
                            FTYPE←ATYPE;
                        ENDIF;
                    ENDIF;
                    NAWP←NAW;
                    IF ASTR=∅ OR ASTR=2 DO;
                        NAW←NAW+1 IF ASTR=2;
                        IF FP $ FSTR=∅ AND ASTR=2 OR FP $ FSTR=1
                            AND ASTR=∅ DO;
                            BLLERR(3) IF FTN=∅; FTNATF←1;
                            TEMP←NAW+1B6;
                            GOTO L1;
                        ENDIF;
                    ELSE DO;
                        BLLERR(3) IF FP $ FSTR=∅;
                    ENDIF;
*    CHECK FOR ACTUAL ARG IN ACCUMULATOR
                    IF (AP AND 7∅∅37777B)#∅ DO;
                        R←P; IR←XR; EA(NAWP); ARGADR←Q;
                        IF FP $ CADDR=1 DO;
                            IF IMMEDIATE=1 DO;
*    CONSTRUCT IMMEDIATE IAW
                                TEMP←OP AND 3777B OR 1634B4;
                            ELSE DO;
                                RINGCHECK(ARGADR); TEMP←ARGADR;
```

```
*  MAKE THE IAW READ-ONLY IF NECESSARY
                        TEMP←TEMP+1B7 IF READONLY=1 OR ASTR=3;
                    ENDIF;
*  FIX UP SO THE COPY VALUE CODE WILL COPY THE ADDRESS IN TEMP
L1:                     FTYPE←1; FP $ FSTR←1;
                    ELSE DO;
                        IF IMMEDIATE=1 DO;
                            BLLERR(5) IF FTYPE#1 OR FP $ FSTR=∅;
                        ENDIF;
                        TEMP←(OP IF FTYPE=1 ELSE CONTENTS(ARGADR));
                    ENDIF;
                    OLDR←R;
                    CPYADR←((FP AND 3777B)+NEWL IF FP<∅ ELSE
                        (FP AND 37777B)+NEWG);
                    GOTO ARRAY IF FP $ FSTR=∅;
                    COUNT←(1 IF FTYPE=1 OR FTYPE=9 ELSE
                           2 IF FTYPE=2 OR FTYPE=3 ELSE
                           4 IF FTYPE=4 OR FTYPE=5 OR FTYPE=6
                             ELSE GOTO STRING IF FTYPE=7
                             ELSE GOTO LABEL IF FTYPE=8
                             ELSE BLLERR(4));
                    UFN'TRAP() IF(FTYPE=3 OR FTYPE=4)
                        AND UNDEFINED(TEMP);
L2:                 R←NEWP; $CPYADR←TEMP; COUNT←COUNT-1;
                    IF COUNT#∅ DO;
                        R←OLDR; Q←Q+1;
                        CPYADR←CPYADR+1;
                        TEMP←CONTENTS(Q); GOTO L2;
                    ENDIF;
                ELSE DO;
                    BLLERR(5) IF FP $ CADDR=1 OR FP $ FSTR=∅;
                    CPYADR←((FP AND 3777B)+NEWL IF FP<∅ ELSE
                        (FP AND 37777B)+NEWG);
                    IF TYPE=3 OR TYPE=4 DO;
                        STF(CPYADR);
                    ELSE DO;
                        COUNT←(1 IF FTYPE=1 OR FTYPE=9 ELSE
                               2 IF FTYPE=2 ELSE
                               4 IF FTYPE=5 OR FTYPE=6 ELSE
                               BLLERR(4));
                        R←NEWP;
                        STORE(CPYADR, A);
                        IF COUNT#1 DO;
                            STORE(CPYADR+1, B);
                            IF COUNT#2 DO;
                                STORE(CPYADR+2, C);
                                STORE(CPYADR+3, D);
                            ENDIF;
                        ENDIF;
                    ENDIF;
                ENDIF;
            ENDIF;
            NAW←NAW+1;
```

```
L3:            ENDIF;
               INTERRUPT'CHECK();
               GOTO L4 IF FP $ ENDF=1;
           ENDFOR;
L4:        NEWP←NFW+1;
       ENDIF;
*
*  COMPUTE RETURN DESCRIPTOR
       IF CLL=1 DO;
           R←NEWP;
           NEWL[Ø]←NAW;
           NEWL[1]←L+2B7*STK+1B7*CPR;
           NEWG[14B]←G IF MCAL>Ø AND RING(NEWP)>RING(P);
       ENDIF;
       IF STK=1 DO;
           IF CLL=1 DO
               R←NEWP; NEWG[2]←SP;
           ELSE DO;
               R←P; G[2]←SP;
           ENDIF;
       ENDIF;
       IF MCAL=2 DO;
MENTER:    PROTECT(4);
           SET'LOCK();
       ENDIF;
       SR $ TDFLAG←SR $ PDFLAG←Ø IF MCAL>Ø;
       L←NEWL; G←NEWG; OLDP←P; P←NEWP;
       IF RET=1 DO;
           IF OLDP>=6B5 DO;
MEXIT:         UNPROTECT(4);
               RESET'LOCK();
               XMON'TRAP() IF SR $ XMONT;
           ELSE DO
               XUTIL'TRAP() IF SR $ XUTILT;
           ENDIF;
       ENDIF;
       P←P+1 IF FTN=1 AND FTNATF=Ø;
*
*  EXIT FROM BLL
       GOTO NEXT'INSTRUCTION;
*
STRING: COUNT←4; GOTO L2 IF MCAL=Ø
       FORM←TEMP AND 14B6 OR 4B7; OLDT←Ø;
       FOR I←Ø BY 1 DO;
           R←P; RINGCHECK(TEMP);
           BLLERR(6) IF OLDT $ WA>TEMP $ WA OR
                         OLDT $ WA=TEMP $ WA AND
                         OLDT $ CPOS>TEMP $ CPOS;
           R←NEWP; $(CPYADR+I)←TEMP AND NOT 74B6 OR FORM;
           GOTO L3 IF I=3; R←OLDR; OLDT←TEMP;
           TEMP←CONTENTS(ARGADR+I+1);
       ENDFOR;
```

```
*
LABEL:  Q←(TEMP $ FLW IF TEMP $ BIT4=∅
            ELSE ARGADR+TEMP $ FSRW);
        RINGCHECK(Q) IF MCAL>∅;
        R←NEWP;
        STORE(CPYADR, Q AND NOT 75B6 OR TEMP AND 75B6);
        R←OLDR; BRD←CONTENTS(ARGADR+1);
        IF BRD $ FE=∅ AND BRD $ FSTK=∅ DO;
            BRD←BRD AND NOT 4B7 IF MCAL>∅;
            BRD←BRD OR (L IF STK=∅ ELSE NEWL+2B7+4B6);
        ELSE DO;
            BLLERR(6) IF MCAL>∅;
        ENDIF;
        R←NEWP;
        STORE(COPYADR+1,BRD); GOTO L2;
*
ARRAY:  R←NEWP; $CPYADR←TEMP;
        BLLERR(6) IF TEMP $ IAT#3;
        IF MCAL>∅ DO;
            IF←(TEMP $ UB1 IF TEMP $ LEB=∅ ELSE TEMP $ UB2);
            IA(ARGADR+1); RINGCHECK(Q);
        ENDIF;
        IR←∅; R←ARGADR; IA(ARGADR+1);
        BLLERR(6) IF IMMEDIATE=1;
        RINGCHECK(Q) IF MCAL>∅;
        R←NEWP;
        $(CPYADR+1)←(Q+(4B6 IF READONLY=∅ ELSE 12B6));
        GOTO L3;
```

V. FIXED TRAPS

Number	Name	Caused by	Parameter
1	MACC	Memory access error - attempted access to monitor from below M or utility from below U	$Q+(RING(R)-1)*1B6$
2	PRO	attempted write of RO page	Q
3	PNIM	attempted reference to page not in map	Q
4	PNIC	attempted reference to page not in core	Q
5	TO	timer overflow - not in monitor mode	---
6	PI	privileged instruction	---
7	TI	trapped instruction	---
8	XMON	on exit from monitor via any BLL or LOADS if XMONT is set in the state	---
9	XUTIL	on exit from utility via any BLL or LOADS if XUTILT is set in the state	---
11	ILIM	indirect limit exceeded	address of IAW
12	MAB	map abort	---