Design features of the BCC 500 CPU

C.F. Wall

Presented at The Seventh Hawaii International Conference on System Sciences, 8 January 1974, Honolulu

DRAFT

1. Introduction

The BCC 500 computing system was designed to provide a large number of users with remote access to centralized, general-purpose computing and file storage facilities. The system was intended to be used both in batch mode and interactively. Considerable emphasis was given to providing rapid response for large numbers of relatively small processes. Among some of the foreseen primary uses of the system were such applications as reservations systems, banking systems and various data entry and information retrieval systems. In order to provide this class of service economically the hardware configuration chosen for the system consists of a number of processors connected to a central memory. The processors are not of uniform design, being dedicated rather to specific purposes for reasons of efficiency, integrity, and system security.

The BCC 500 operating system is distributed over several of these processors which are otherwise independent and which communicate with one another by means of the central memory. These processors are assigned the tasks of process scheduling, memory management, character input/output for terminal and network access devices, system supervision and monitoring, and running of user programs. The task of running user programs is performed by two general purpose processors called CPUs. Since most of the operating system tasks are performed by other processors, the CPUs are free to service user programs. A portion of the operating system, called the monitor, runs on the CPU to provide users with protected access to operating system services. The monitor is common to all users of the system. It may not be looked at or modified by user programs; it may be accessed only 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. Another portion of the operating system that runs on the CPU, called the utility, contains a number of useful and frequently used functions which each user program would otherwise have to include. It thus provides an interface between the monitor and the user program and extends the basic services of the monitor. The system was designed so that individual user programs can be supplied with alternative utilities that extend or individualize their interface to the operating system.

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 constructs of an interactive systems programming language (SPL) and to provide each user with a virtual machine.[5,6,7]

SPL provides systems and applications designers with a high level development and implementation tool that is capable of providing users with an effective environment. The CPU design efficiently supports this environment 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 encourages programs to be organized into a collection of small routines called functions. It is expected that systems designers will structure their programs in a modular fashion and that arguments and results will be 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. Functions are normally recursive, but it is possible to allocate storage for a function in a fixed location 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. This feature of the CPU was greatly influenced by the Burroughs B5000[1] series of computers. Basically descriptors in the BCC 500 provide the programmer with the ability to easily access various data structures. Information in the descriptor allows the CPU to check and ensure that the access is correctly specified. Three types of descriptors are implemented on the CPU to allow 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 address 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 provide for easy code relocation. The CPU provides a simple instruction set to facilitate easy translation from SPL operations to machine operations. It also provides an environment for programs called a virtual or user machine.[5,6,8] A single copy of this user machine on which programs are executed is called a user process.[5,6,8] Each user process contains a virtual address space of 256K words.

There are a small number of register that are an integral part of the various CPU functions. It is useful to list and describe them here since we will be referring to them throughout the remainder of the paper. The information in most of these registers must be saved when a process is blocked and the CPU is assigned to a new 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 and for loading and storing 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

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

• Registers Used in Addressing

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

• Other Special Registers

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

*not part of state

Fig. 1.1 Machine Registers

2. Function Call and Return Mechanism

Since SPL programs will be very modular and will consist of a large number of small functions, the function call and return mechanism is quite important. This mechanism is implemented in hardware by an instruction called BLL (Branch and Load the Local-environment register) that addresses a two word branch descriptor (or function descriptor). The branch descriptor contains all the information necessary to describe the environment involved and to facilitate all the checking needed to accomplish function call and return actions in a wide variety of possible situations.

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

```
T'INV ← INVERSE(MATM(TRANS(T),T):,T'DET);
```

causes the matrix inverse of T transpose times T to be stored in T'INV, which is the value of the inverse function, and the determinant to be stored in T'DET. A function may have no arguments and/or return no results. The number and type of arguments and results are checked by the hardware at run time. The SPL function call and return mechanism also allows for a "failure" return from a function. The failure return can return results of its own and may return control to the caller's failure return label. Thus a function call of the form

```
T'INV ← INVERSE(MATM(T(TRANS),T):,T'DET//T'SINGULAR:T'LOWER'TRI);
```

causes the partial results of the inverse function to be stored in T'LOWER'TRI if the matrix formed by multiplying the transpose of T by T singular and control to be returned to the statement labeled T'SINGULAR. Besides these ordinary function calls, SPL provides for operating system calls and intrinsic function calls.

The BLL instruction address a two word branch descriptor which contains the following information:

- Address of the called routine's entry point;
- Whether the branch descriptor is a function descriptor or a return descriptor;
- Whether the storage for the function (called its local environment) must be allocated from a stack;
- Whether arguments 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:

1) Obtain the effective address of the entry point in the called routine;
2) Acquire the new local environment and obtain storage if the function allocates space for its local environment on the stack;
3) Copy arguments;
4) Compute a return descriptor and save it in the first two words of the new local environment;
5) Transfer Control;
6) Obtain the old local environment from the return descriptor;
7) Copy results;
8) Return 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. Next, the new local environment 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 is preserved between function calls. Normally, space for a function's local environment is allocated from the stack. Two words in the global environment describe the stack. The stack pointer (SP) addresses the first unused word and the stack limit (SL) addresses the last word allocated for the stack. In this case 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).

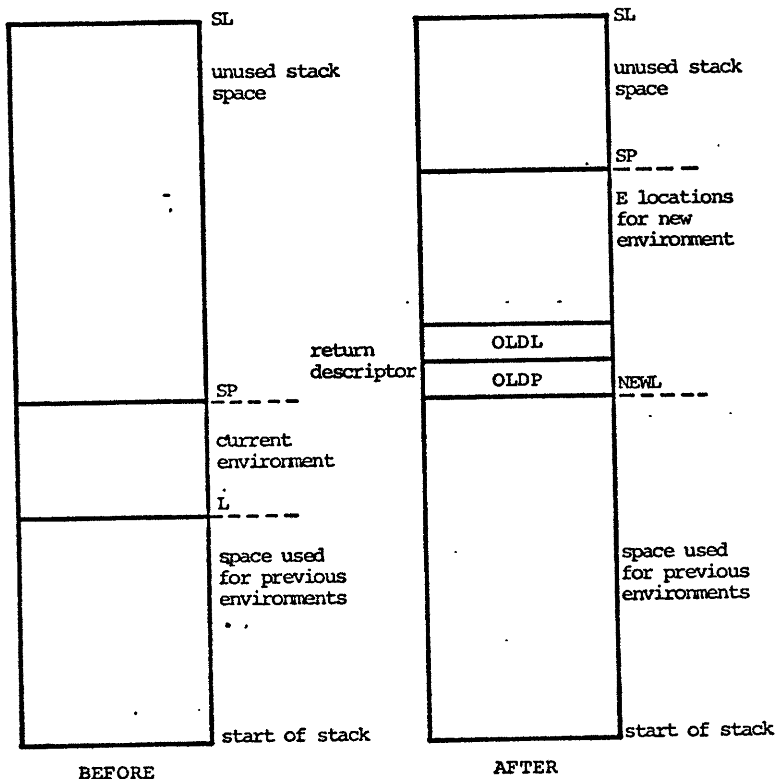


Fig. 2.1 Allocating a Local Environment on the Stack During a Call

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 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)
real	(2 words)
double	(4 words)
complex	(4 words)
longlong	(4 words)
string	(4 words)
label	(4 words)
pointer	(1 word)
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, although the structure specified is only 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	FAW's Array
AAW's	Variable	OK	Error
	Computed scalar	OK	Error if not a FORTRAN type fn
	Array element	OK	Error
	Array	Error if not a FORTRAN type fn	OK

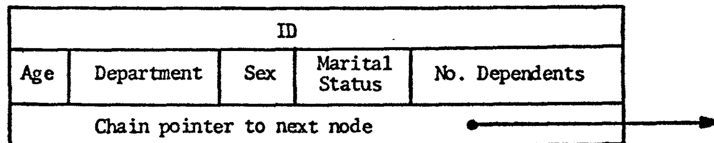
Copying continues until an end flag occurs. If both end flags (AAW and FAW) do not appear at the same argument level then the wrong number of arugments have been supplied and an error occurs. The BLL does not provide for type conversion.

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) that provide for the selection of words or partial words from a block of packed data. 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(Ø);
DECLARE FIELD AGE(1:Ø,3),
              DEPT(1:4,12),
              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 we obtain a pointer P to a node of this type, in SPL we can increment the age field by coding

```
P.AGE ← P.AGE+1;
```

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

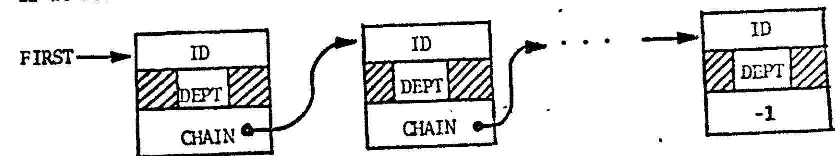
```
P.DEPT ← NEW'DEPT'NO;
```

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 the pointer-displacement and base-index modes of addressing and a hardware implemented field descriptor. 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:



where the nodes are as we have previously defined them, then the SPL code

to find a node that contains a particular department given that FIRST is a pointer to the starting node is as follows:

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

The machine code generated is as follows:

```
LDA FIRST        Load A register with first pointer
BRU R'[2]        Branch source relative plus 2
LDA PTR.2        (Pointer Displacement)
STA PTR
ICP -1           Compare A register with -1
BEQ R'[5]        Branch if equal to source relative plus 5
LDA DEPT[PTR]    (Base Index)
ICP DEPT'NO
BEQ FOUND
BRU R'[-10B]
```

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 contents of the A-register are stored in a variable called PTR and is 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 we seek. If it is equal to this department number 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. 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. The first instruction is accomplished using the pointer-displacement mode of addressing (see Fig. 3.1).

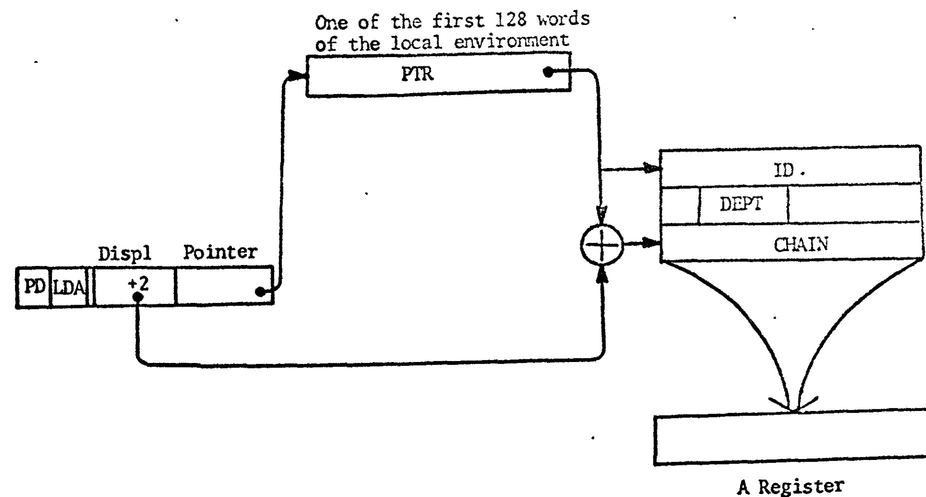


Fig. 3.1 PD Addressing for LDA PTR.2

The second instruction is accomplished using the base-index mode of addressing in conjunction with the field descriptor for "DEPT" (see Fig. 3.2).

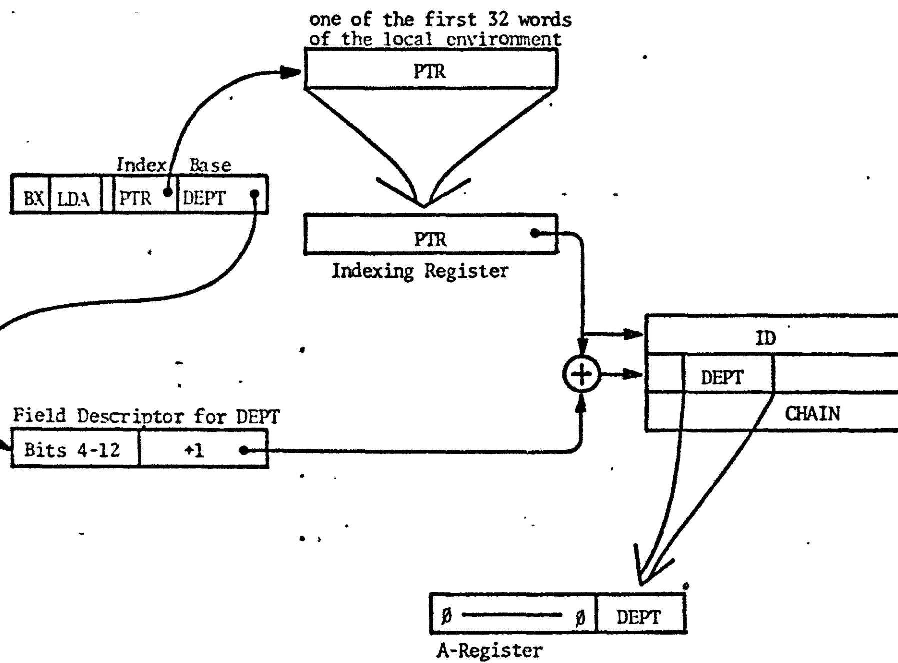


Fig. 3.2 BX Addressing in Conjunction with the Field Descriptor for DEPT

Insertion of 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

Read Pointer (RP) - points to last character read

Write Pointer (WP) - points to last character written

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

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

String type IAW

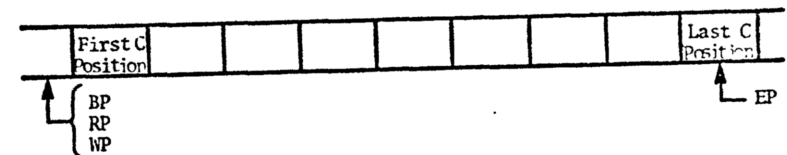
Character size: 6, 8, 12 or 24 bit characters

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



B. After a few Reads and Writes

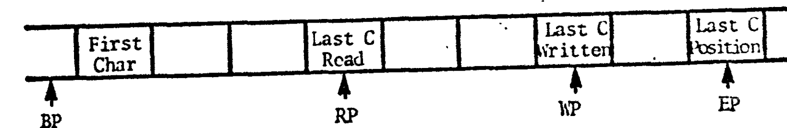


Fig. 4.1 String Descriptors and Where They Point

SPL Operations	CPU Instructions	
Create string descriptor	Load String Constant	(LSC)
Read and Write characters	Increment String Descriptor	(ISD)
	Decrement String Descriptor	(DSD)
	Add to String Pointer	(ASP)
Computer length of string	Computer Length of String	(CLS)
Copy or Move string	Move String	(MVS)
Compare Strings	Compare String	(CPS)

The increment and decrement descriptor (ISD and DSD) instructions and the add string point (ASP) instruction work with pairs of string indirect words in the string descriptor. Thus during a read operation the read pointer can be checked in conjunction with the write pointer to make sure that any attempt to read beyond the write pointer is trapped. This provides all the checking necessary. The ISD and DSD instructions facilitate character reading and writing while the ASP instruction facilitates accessing the Nth character in a string. The LSC instruction generates string descriptors. The actual characters in the string are read and written by loading indirectly through the read pointer string IAW and stored indirectly through the write pointer string IAW.

The string descriptors and the various string operations provided the basis for efficient, flexible and versatile string processing.

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 the value of A is an array descriptor for an array with three entries. Each entry of this array is an array descriptor for an array with four entries. Each entry of this array is a row descriptor for a row with five entries, each of which is a real number. 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 to be referenced. An array indirect address word 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

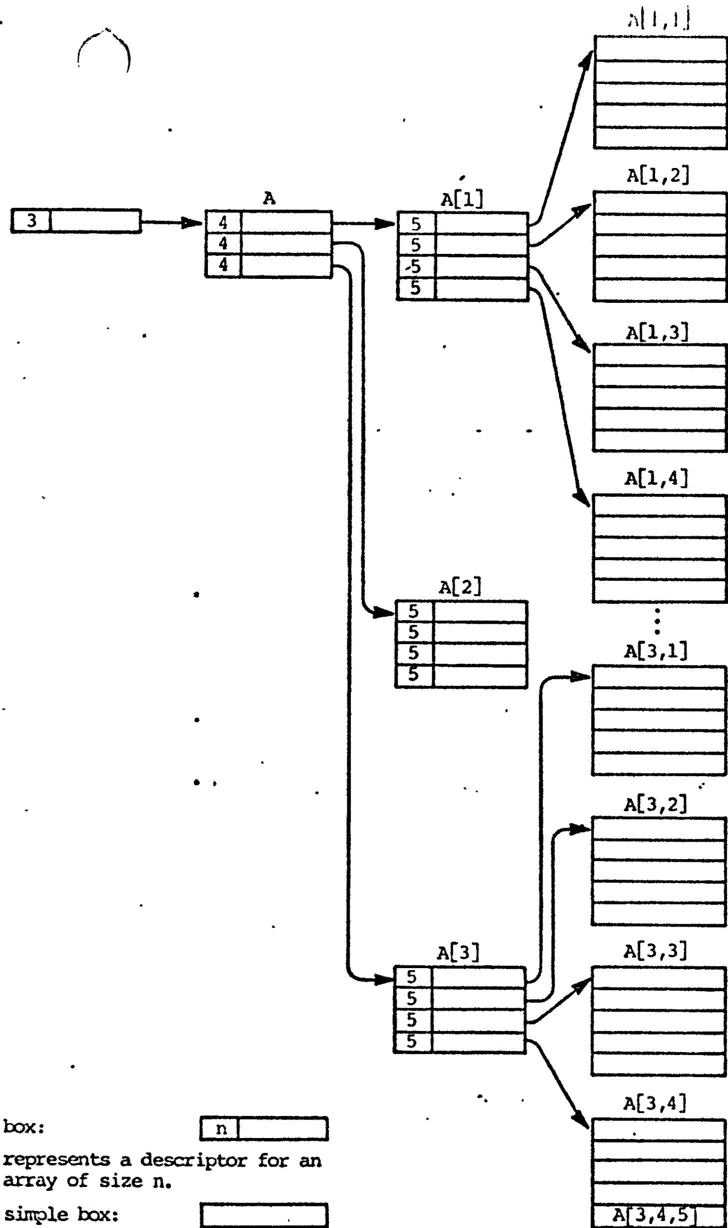


Fig. 5.1 Marginally Indexed Array Structure

This information allows the following functions to be accomplished for array referencing:

- Allows a zero or one lower bound
- Perform bounds check on the subscript
- Multiply the subscript by the size of the array element, allowing for element sizes up to 64
- Check to see that the number of subscripts supplied is the number expected
- Provide an 18-bit base address for the array

Array referencing is accomplished by the array descriptor being referenced by 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 declarations:

```
DECLARE INTEGER ARRAYONE A[3,3];
```

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

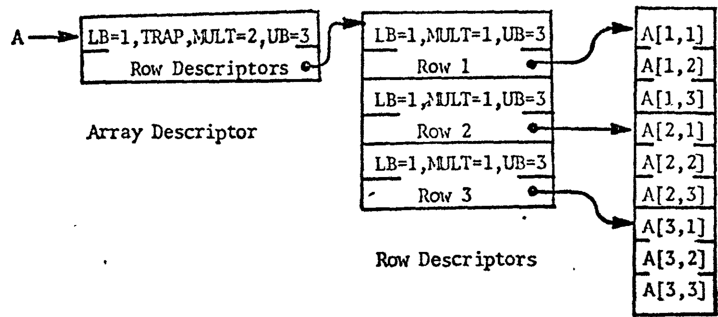


Fig. 5.3 Marginal Indexing

Where 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)

leaves the address of the descriptor for the Kth row in the X-register

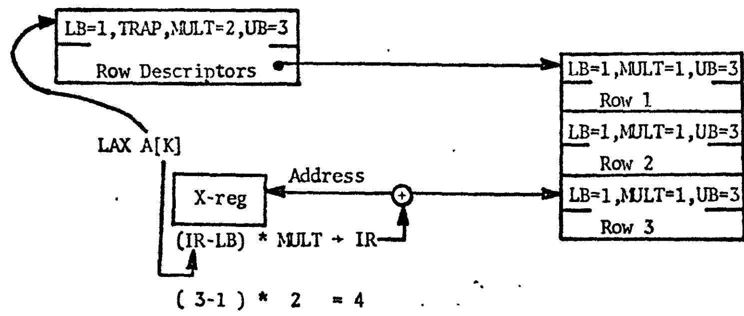


Fig. 5.4 Diagram of LAX A[K] Execution

followed by;

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

STA B

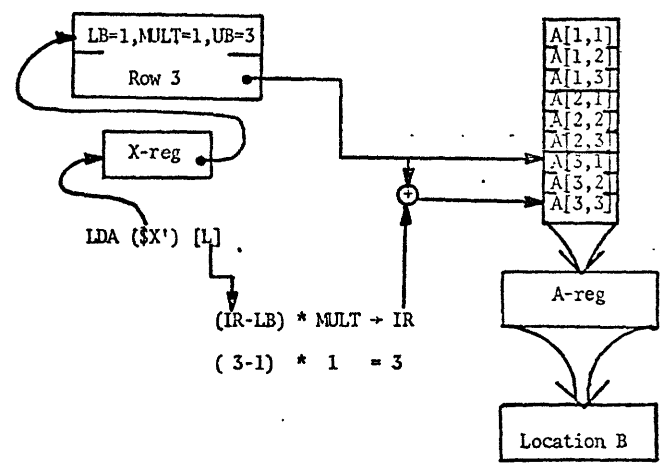


Fig. 5.5 Extracting A[3,3] and Storing it in B

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.

Thus, 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 implemented on the CPU are designed to support the SPL addressing requirements. In the case of descriptor we have seen two addressing modes pointer-displacement and base-index--that are designed to work in conjunction with descriptors to access field 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 will normally be organized into a collection of relatively small, self contained routines called functions. Each function has some private or 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 objects that are contained in a global environment that can be shared by several functions;
- 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).

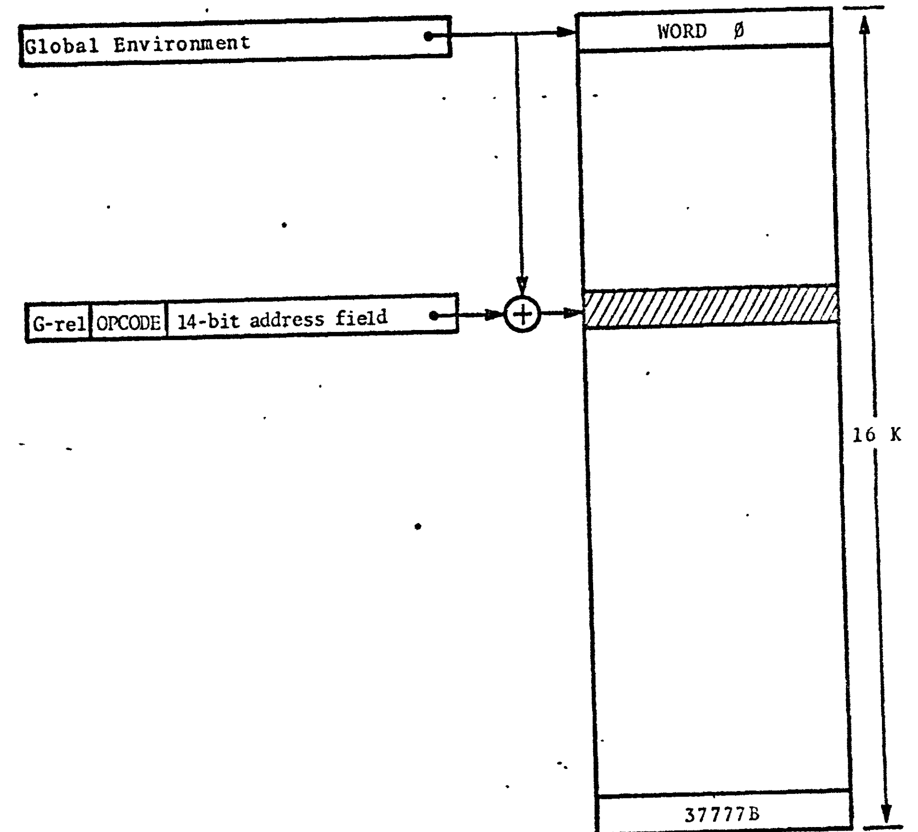


Fig. 6.1 G-Rel Addressing

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).

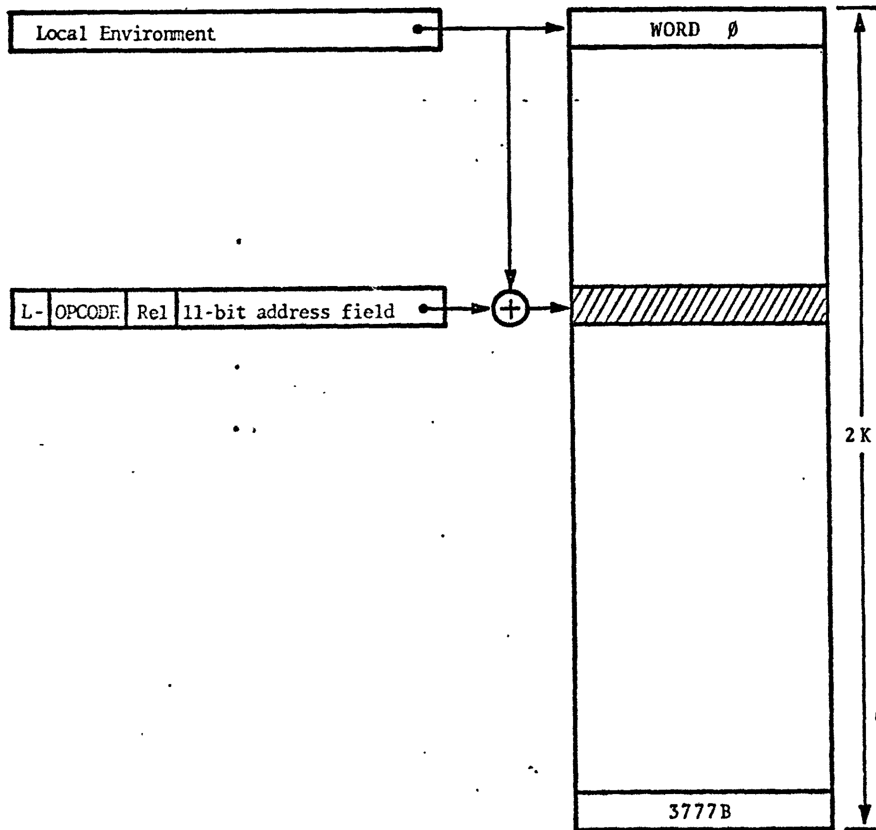


Fig. 6.2 L-Rel Addressing

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).

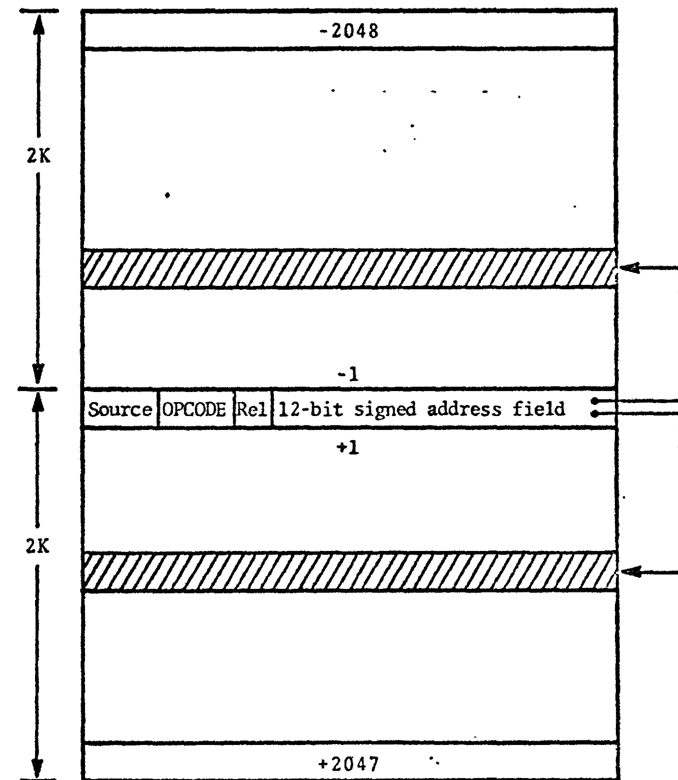


Fig. 6.3 Source-Rel Addressing

As we have seen with the field addressing example, it is very desirable to be able to access various objects in structures such as lists, trees, and tables and other types of data structures that are very common in systems code and in compiling. In general, access to objects or locations in these structures involves obtaining a pointer to a single node or the start of the table along with a displacement to the actual object or location desired. This facility is provided by the pointer-displacement addressing mode (see Fig. 6.4).

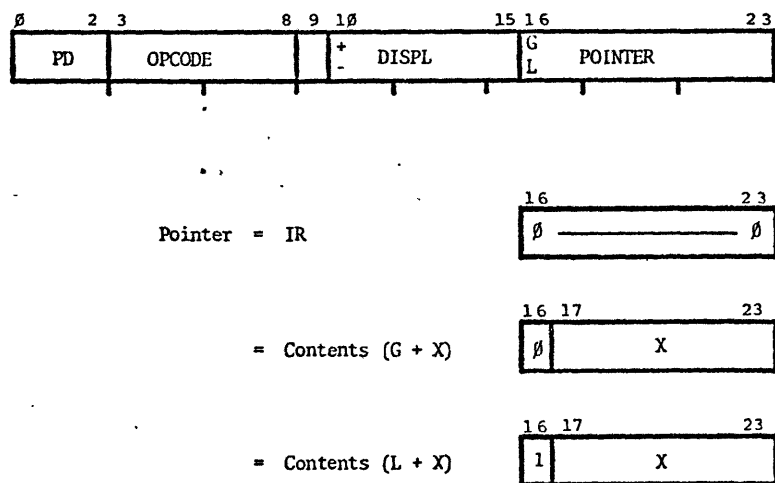


Fig. 6.4 Pointer-Displacement Structure

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).

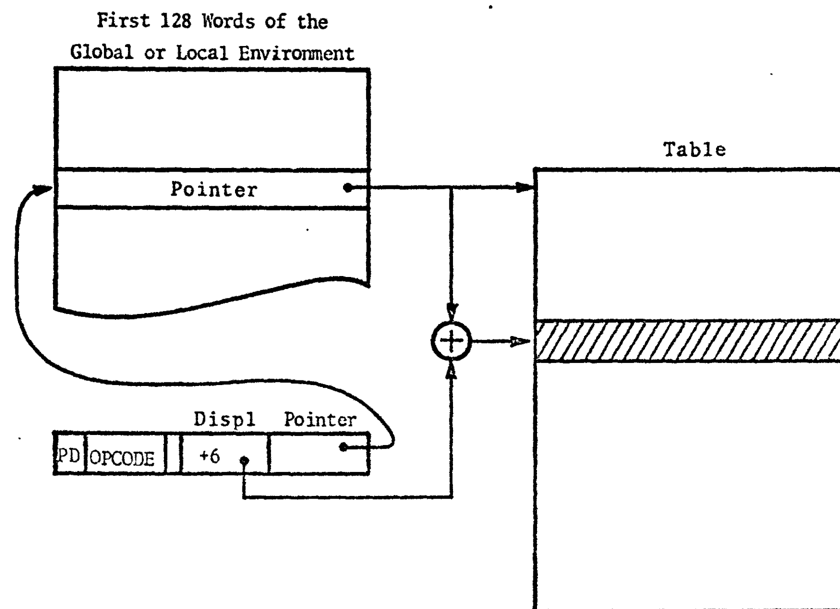


Fig. 6.5 Pointer-Displacement Addressing

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) which causes a new stage of addressing by providing its own addressing information. An IAW can provide an 18-bit address and can access any location in the virtual memory. An IAW specifies address modes in a manner similar to instruction, 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 type of IAW is called normal indirection. 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.

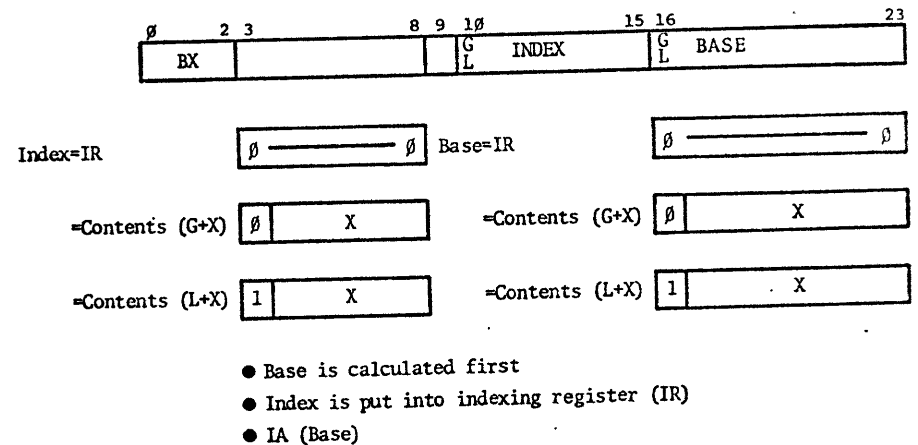
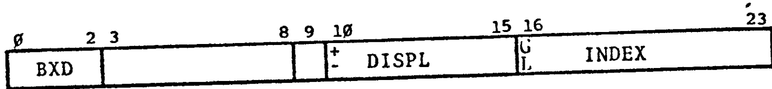
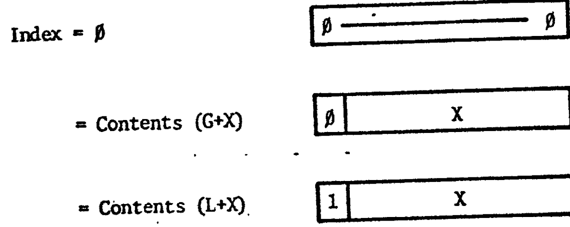


Fig. 6.6 Base-Index Structure





- 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

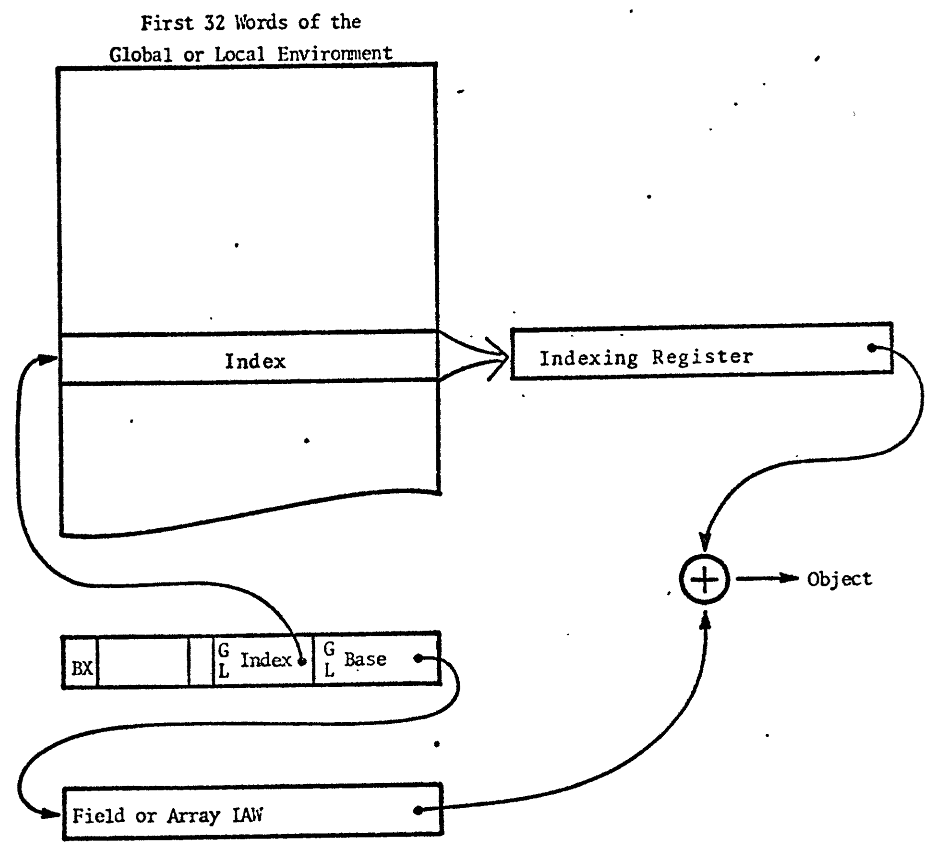


Fig. 6.8 Base-Index Addressing Example

A summary of all the addressing modes appears in Appendix 1.

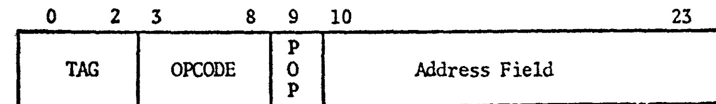
7. Instructions

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 Appendix II and III, respectively.

An instruction is formatted as follows:



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
FLC - Floating Compare
FLX - Convert floating point to fixed and load X
FNA - Floating Negate
FIX - Convert floating point to fixed and load A
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 users

program can select to handle overflow, underflow and division by zero traps. The program can specify soft underflow which allows numbers to draft 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 Environment

The CPU is a 24-bit, word oriented, twos 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, 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 user process 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 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] and functions within an operating system that is distributed over several other microprocessors. 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 or more correctly page slots. The CPU may access information in pages only when they are in core storage. It is the memory managers 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 togehter 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 is 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 asynchronous 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 systems or applications 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 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 monitor is common to all virtual machines and provides programs with access to the services of the distributed operating system. There may be any number of utilities provided by various user groups that extend and individualize the virtual machine environment. 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 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 factors such as response time, number of terminal lines, etc., while the access keys allow for protection of various objects. Basically then the virtual memory structure for a user consists of all the pages in the set of objects he can access (see Fig. 10.1).

†"Whatnots" were first discovered in the system by Jack Freeman and are self referential in that a whatnot is a type of "whatnot".

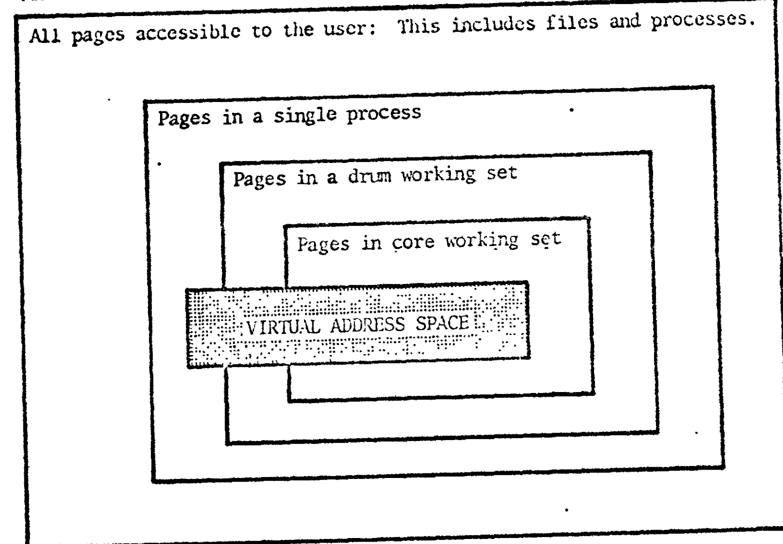


Fig. 10.1 Virtual Memory Structure for a User

A process is one copy of the virtual machine. All the pages a process can reference are contained 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. That is 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

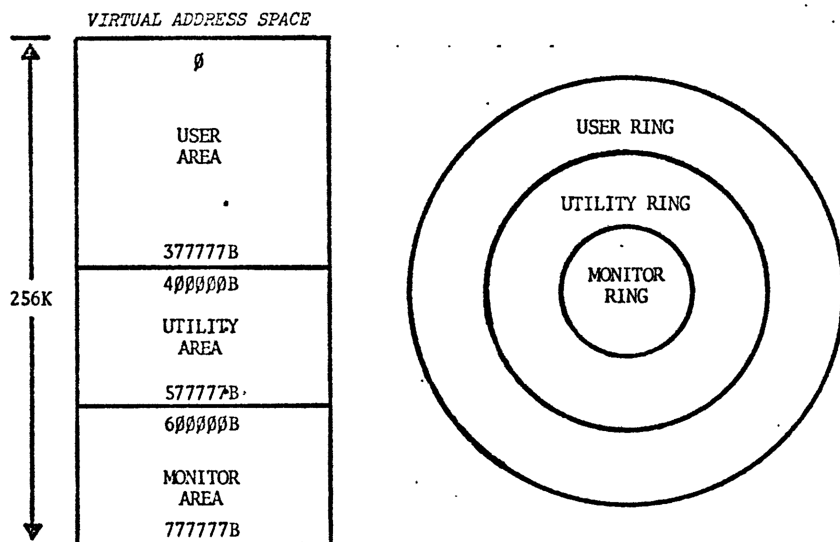


Fig. 10.2 Virtual Address Space and Protection Rings

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 must be mapped into a physical address in core storage. A reference into the virtual address space consists of an 18-bit address which is composed of a virtual page number (the top 7-bits) and a word number (the low order 11-bits). The CPU must map this 7-bit virtual page 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. This 8-bit page number together with the 11-bit word number provides the 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

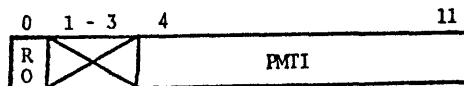
location-independent name → physical page number .

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 latter 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

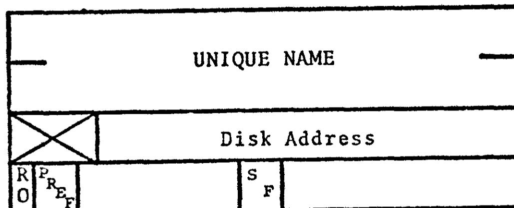


RO - Read Only Bit

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



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 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

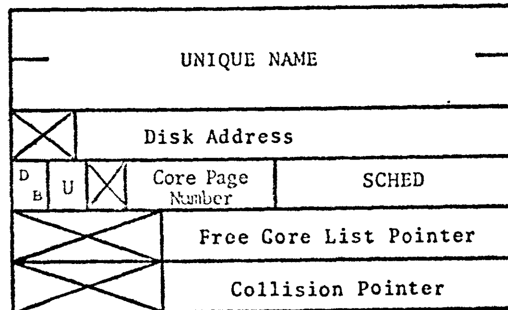
SCHEDULED FLAG: The memory management system sets this bit if a process is authorized to access this particular page. That is, the page is in the core working set of the process.

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.



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. This bit is set when it is determined that a write onto the disk may take place.
- 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 register, 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,

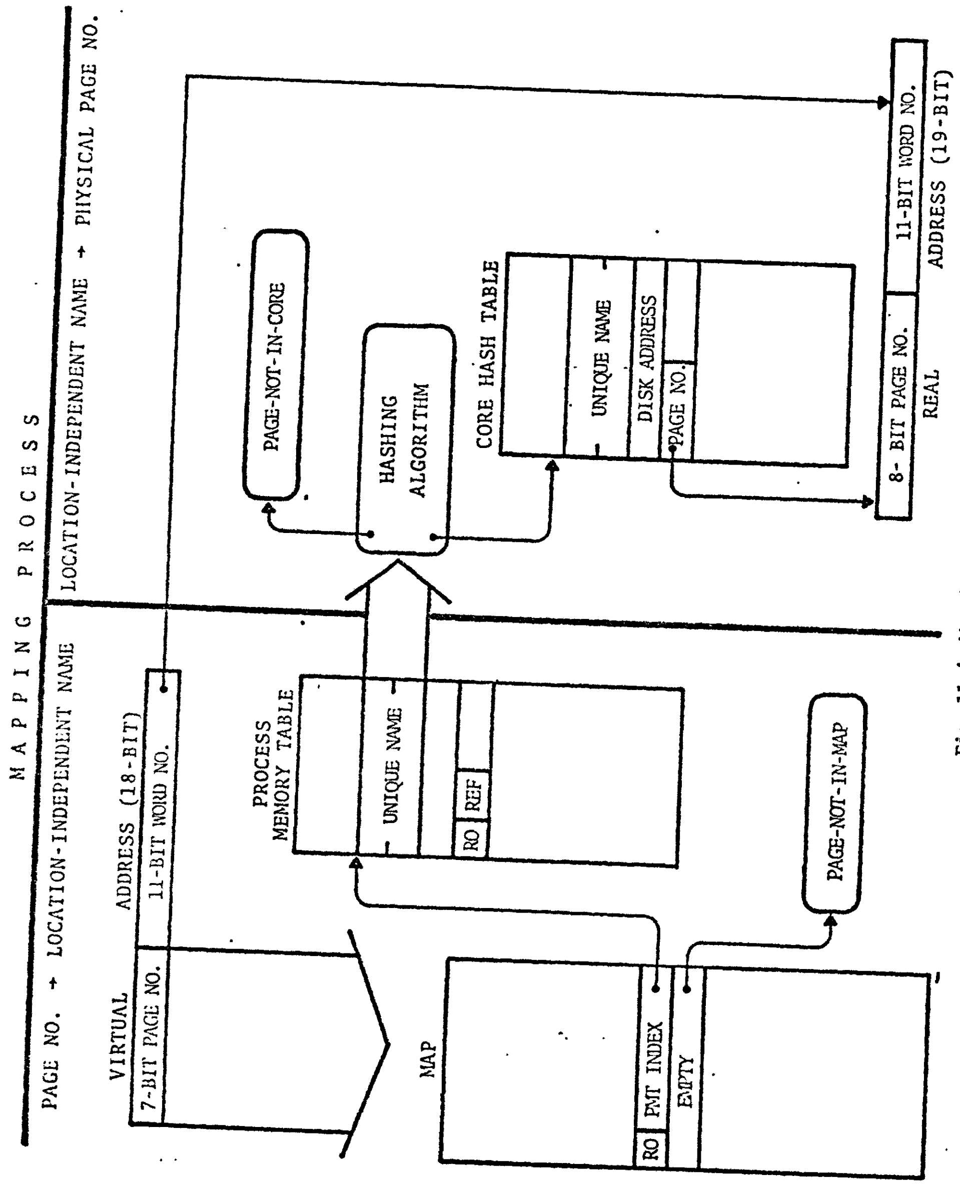


Fig. 11.4 Mapping Process

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

II. SPL Operations

III. Machine Instructions

Summary of Abbreviations

Data Transfer

Integer Arithmetic

Test

Logical

Shift

Branch

Miscellaneous

OPR

Floating Point

IV. SPL Definition of BLL

V. Traps