

# BCC 500 CPU REFERENCE MANUAL

Chuck Wall  
Jack Freeman

THE ALOHA SYSTEM  
University of Hawaii

Document No. BCC/M-1  
December 31, 1973

Contract NAS2-6700  
Department of Defense  
Advanced Research Projects Agency  
ARPA Order No. 1956

ABSTRACT

This document provides a complete description of the BCC-500 CPU from a machine language programming point of view. It is expected that most programming will be at a level well above machine language, since the Systems Programming Language (SPL) is completely adequate for most of the systems development. This document represents an updated, edited and revised version of two working documents originally written by Butler Lampson and Charles Simonyi. This manual reflects the changes made to the CPU at the University of Hawaii.

## TABLE OF CONTENTS

|  | Page |
|--|------|
| 1. INTRODUCTION .....                      | 3    |
| 2. GENERAL CHARACTERISTICS AND STATE ..... | 4    |
| 3. ADDRESS SPACE AND MAP .....             | 7    |
| 4. ADDRESSING FROM INSTRUCTIONS .....      | 16   |
| 5. INDIRECT ADDRESSING .....               | 34   |
| 6. USE OF ADDRESSES BY INSTRUCTIONS .....  | 45   |
| 7. FUNCTION CALLS .....                    | 47   |
| 8. PROGRAMMED OPERATORS .....              | 62   |
| 9. SYSTEM CALLS .....                      | 63   |
| 10. TRAPS .....                            | 64   |
| 11. CPU INTERRUPTABILITY .....             | 65   |
| 12. ORDINARY INSTRUCTIONS .....            | 70   |
| 13. FLOATING POINT .....                   | 92   |
| 14. APPENDIX CONTENTS .....                | 102  |
| DEFINITION OF INSTRUCTION CODES .....      | 103  |
| DEFINITION OF OPR ADDRESSES .....          | 104  |
| SUMMARY OF ADDRESSING .....                | 105  |
| SUMMARY OF INSTRUCTION ADDRESSING .....    | 106  |
| FIXED TRAPS .....                          | 108  |
| RING-DEPENDENT TRAPS .....                 | 109  |
| RING-DEPENDENT TRAP IØ: BLLERR .....       | 110  |
| SUMMARY OF IMPORTANT CORE ADDRESS .....    | 111  |
| SPL PROGRAM TO DEFINE BLL .....            | 113  |
| WORD FORMATS .....                         | 118  |
| SOME FIELDS IN OCTAL FORM .....            | 127  |
| CHT HASHING ALGORITHM .....                | 128  |

## 1. Introduction

This is a reference manual for the BCC 500 central processor unit (CPU). It is intended to be a complete and self-contained description of the characteristics of the processor from the point of view of a machine language programmer (although it is hoped that few programmers will ever have occasion to descend to machine language).

Two considerations have dominated the design of the CPU. They are stated here in the hope that they will make clearer the rationale for some of the machine's characteristics.

- 1) The CPU will be implemented on a somewhat modified version of a BCC microprocessor. This implied that peculiar instruction and addressing sequencing can be used freely.
- 2) The CPU will be programmed almost entirely in SPL or FORTRAN. It is therefore essential that the common constructs of these languages have efficient hardware counterparts. Most notable among them are array referencing, function calls and returns, part word field accessing and string processing.

## 2. General Characteristics and State

The CPU is a 24-bit, word oriented, two's complement machine. It has 64 instructions and a variety of addressing modes. Bits are numbered 0 to 23 with bit 0 on the left (most significant) end of the word. Both single (48-bit) and double (96-bit) precision floating point arithmetic are implemented in hardware.

A process, which may be defined as a program in its execution environment, is called an active process if it is running on the CPU. All the information necessary to define a process is contained in a single page of its virtual memory and is called its context block. When an active process is blocked the following objects must be saved:

- All pages in the process that have been modified;
- The context block;
- The state of the CPU.

The state of the CPU is saved in a fixed location in the context block and consists of 12 registers as illustrated in figure 2.1

| WORD |   | NAME |
|------|---|------|
| 0    | <del>0</del> 5 6 Program Counter 23             | P    |
| 1    | <del>0</del> A - register 23                    | A    |
| 2    | <del>0</del> B - register 23                    | B    |
| 3    | <del>0</del> C - register 23                    | C    |
| 4    | <del>0</del> D - register 23                    | D    |
| 5    | <del>0</del> Floating point exponent 11 12 23   | E    |
| 6    | <del>0</del> Index register 23                  | X    |
| 7    | <del>0</del> Local environment register 23      | L    |
| 8    | <del>0</del> Global environment register 23     | G    |
| 9    | <del>0</del> 4 5 Status register (see below) 23 | SR   |
| 10   | <del>0</del> Compute Time Clock 23              | CTC  |
| 11   | <del>0</del> Interval Timer 23                  | IT   |

The status register contains the following fields:

| 0 | 4 | 5 | 6     | 8     | 9  | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|-------|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | F | D | TRMOD | PRMOD | CC | 9  | S  | U  | X  | T  | X  | D  | T  | P  | C  | T  | O  | I  |
|   | D |   |       |       |    | 4  | U  | T  | M  | F  | M  | F  | D  | A  | A  | O  | V  | N  |
|   | P |   |       |       |    | 0  | F  | I  | O  | L  | O  | F  | F  | R  | R  | V  |    | S  |
|   |   |   |       |       |    | M  |    | L  | M  | L  | T  | L  | L  | R  |    |    |    | T  |
|   |   |   |       |       |    |    |    | T  | O  | A  |    | A  | A  |    |    |    |    | D  |

| Bit   | Name   | Contents                        |
|-------|--------|---------------------------------|
| 5     | FDP    | Full double precision flag      |
| 6-8   | TRMOD  | Temporary rounding mode         |
| 9-11  | PRMOD  | Permanent rounding mode         |
| 12-13 | CC     | Condition Code                  |
| 14    | 940M   | 940 Mode                        |
| 15    | SUF    | Soft underflow flag             |
| 16    | XUTILT | Utility exit trap flag          |
| 17    | XMONT  | Monitor exit trap flag          |
| 18    | TDFLAG | Temporary double-precision flag |
| 19    | PDFLAG | Permanent double-precision flag |
| 20    | CARRY  | Carry bit                       |
| 21    | TOV    | Temporary overflow bit          |
| 22    | OV     | Overflow bit                    |
| 23    | INSTD  | Instruction terminated bit      |

FIGURE 2.1 CPU STATE

Note that it is convenient in the notation used in the manual to refer to some registers by affixing an "R" to their name. So you will see the following names referring to the same register.

|   |    |            |
|---|----|------------|
| A | AR | A-register |
| B | BR | B-register |
| C | CR | C-register |
| D | DR | D-register |
| E | ER | E-register |
| X | XR | X-register |

### 3. Address Space and Map

The CPU considers itself at any particular time to be running a process which is defined by its context block. Each process has a 256K address space. The CPU uses 18-bit addresses to specify memory locations. The address space has two significant characteristics:

- 1) it is divided into three rings as follows:

|                     |                        |
|---------------------|------------------------|
| addresses 0-377777B | user ring (lowest)     |
| 400000B-577777B     | utility ring           |
| 600000B-777777B     | monitor ring (highest) |

The rings are protected from each other according to certain rules. Every memory reference is said to have a source. The source for any references generated by an instruction up to and including a fetch of an indirect word is, for example, the program counter; the source for any reference generated after a fetch of an indirect word up to and including a fetch of the next indirect word is the address of the first indirect word. Every reference also has a target, which is the address being referenced. The following matrix defines those combinations of source and targets which are legal.

|        |         | Target |         |         |
|--------|---------|--------|---------|---------|
|        |         | User   | Utility | Monitor |
| Source | User    | Yes    | No      | No      |
|        | Utility | Yes    | Yes     | No      |
|        | Monitor | Yes    | Yes     | Yes     |



To summarize:

- a) References from one ring to a higher one are forbidden.
  - b) If indirection leads to a lower ring, it is forbidden to return to the same or higher ring during the same instruction.
- This fact makes it easy, for example, for monitor routines to enforce the user's protection rules when storing into a table provided by the user: they need only do their stores indirect through an address in the user ring, and the ring protection hardware will do the checking automatically.

A forbidden reference causes trap MACC (Memory ACCess error). The target is passed as a parameter to this trap. See figure 3.1 for the two representations of the virtual address space.

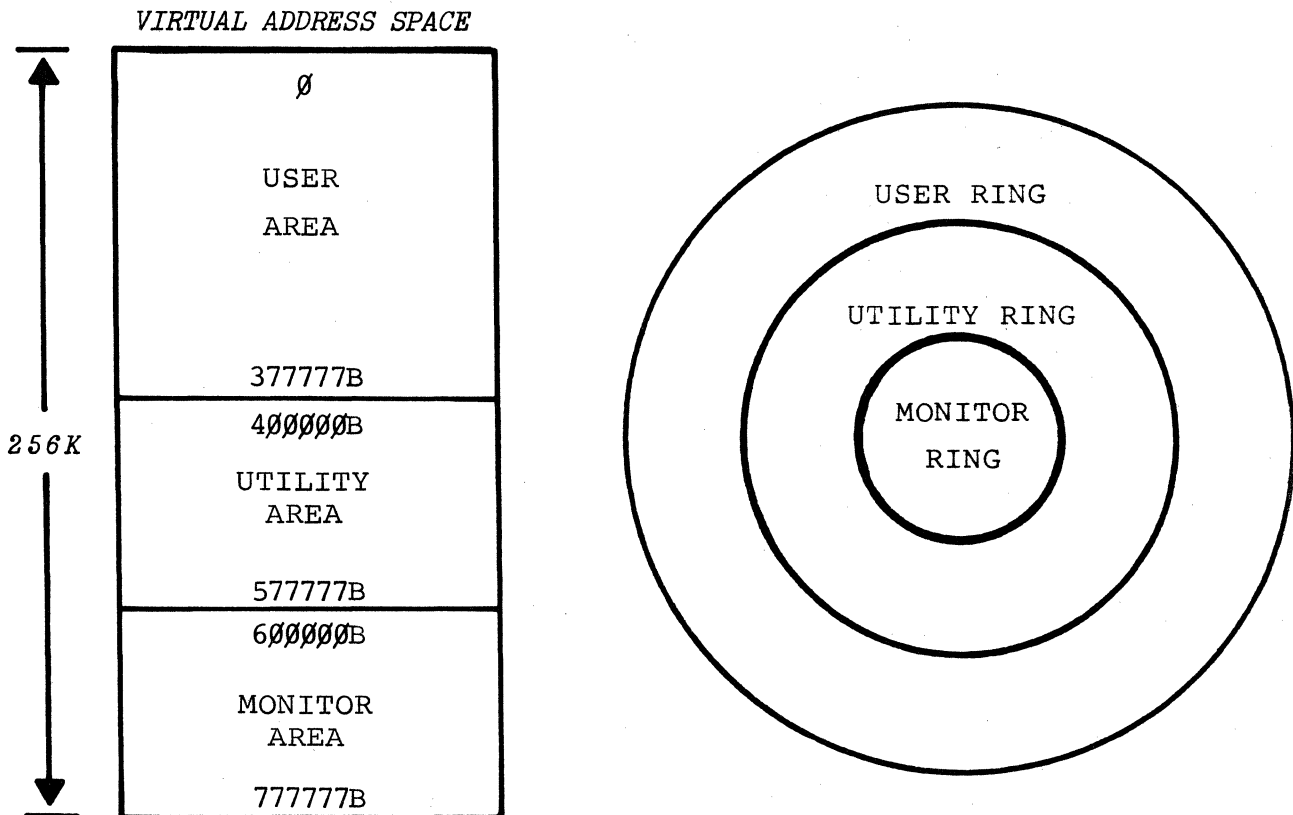


FIGURE 3.1 VIRTUAL ADDRESS SPACE AND PROTECTION RINGS

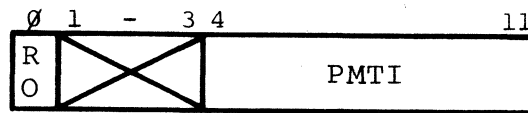
2) The address space is organized into 2048 (2K) word pages, and the precise collection of pages which make up the address space is specified by the map. Pages are named in a manner independent of their location in core, and the mapping hardware uses this location-independent name, together with a table called the core hash table (CHT), to determine the physical core location of a page. The page number (the top 7 bits) of every memory reference thus requires two levels of translation:

from page number to location-independent name

from location-independent name to physical page address

The various mechanisms for performing this translation will now be described.

Locations 200B-277B in the context block contain the map of the virtual address space for the process. These 128 half-word entries specify the contents of the corresponding 128 pages of the address space of the process. Each half-word is interpreted as follows:

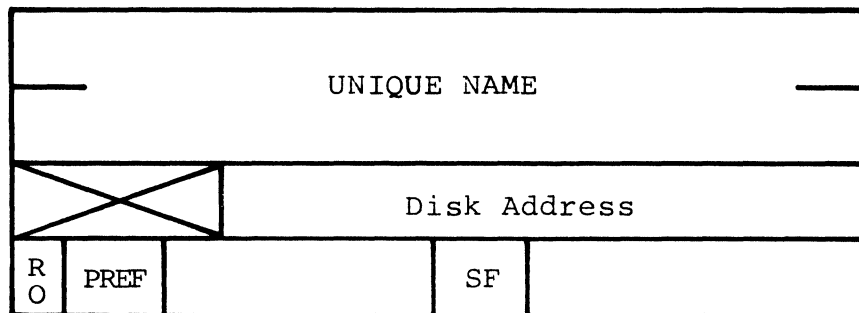


| <u>Bit</u> | <u>Name</u> | <u>Contents</u>  |
|------------|-------------|--|
| 0          | MAPRO       | Read-only bit. This bit is merged with the RO bit in PMT to make the read-only bit interpreted by the hardware |
| 1-3        | ----        | unused   |
| 4-11       | PMTI        | a PMT index  |

FIGURE 3.2 MAP ENTRY

The process memory table (PMT) provides enough information about each page accessible to the process to permit the hardware to access the page. The PMT starts at location 300B in the context block. Each entry is 4 words long; the address in the context block of PMT entry  $i$  is therefore  $4(i-1) + 300B$ .

A PMT entry has the form



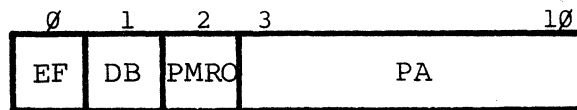
| <u>Word</u> | <u>Bits</u> | <u>Name</u> | <u>Contents</u>   |
|-------------|-------------|-------------|---|
| 0           | 0-23        | UN1         | First 24 bits of unique name for the page (Location-independent name)                   |
| 1           | 0-23        | UN2         | Second 24 bits of unique name for the page  |
| 2           | 2-23        | DA          | Disk address of the page  |
| 3           | 0           | PMTRO       | Read-only bit   |
| 3           | 1           | PREF        | Page has been referenced  |
| 3           | 12          | SF          | Page is scheduled for the process (i.e., in core working set and the process is active) |

The other bits are not used by hardware. The unique name is referred to as UN in the text.

FIGURE 3.3 PROCESS MEMORY TABLE ENTRY

Note that there is no provision for execute-only pages, since this device by itself is not sufficient to protect proprietary programs. The sub-process structure of the monitor is supposed to be used for this purpose.

The central processor contains a physical map (PM) which has 128 registers of 11 bits each. One of the registers has the form:



| <u>Bits</u> | <u>Name</u> | <u>Contents</u>   |
|-------------|-------------|---|
| 0           | EF          | Empty flag  |
| 1           | DB          | Dirty bit, set if the page has been stored into since it was read from the drum |
| 2           | PMRO        | Read-only bit   |
| 3-10        | PA          | Physical address of page in a real core of up to 512K.                          |

FIGURE 3.4 PHYSICAL MAP REGISTER

When a new process starts to run on the processor, the empty flag is set in each PM entry. Every address generated by the program must be mapped to convert it from virtual to real so that an access can be made to the real core. This is done by taking the top 7 bits of the 18-bit address and using them to select one of the 128 PM entries. If the empty flag is off, the re-

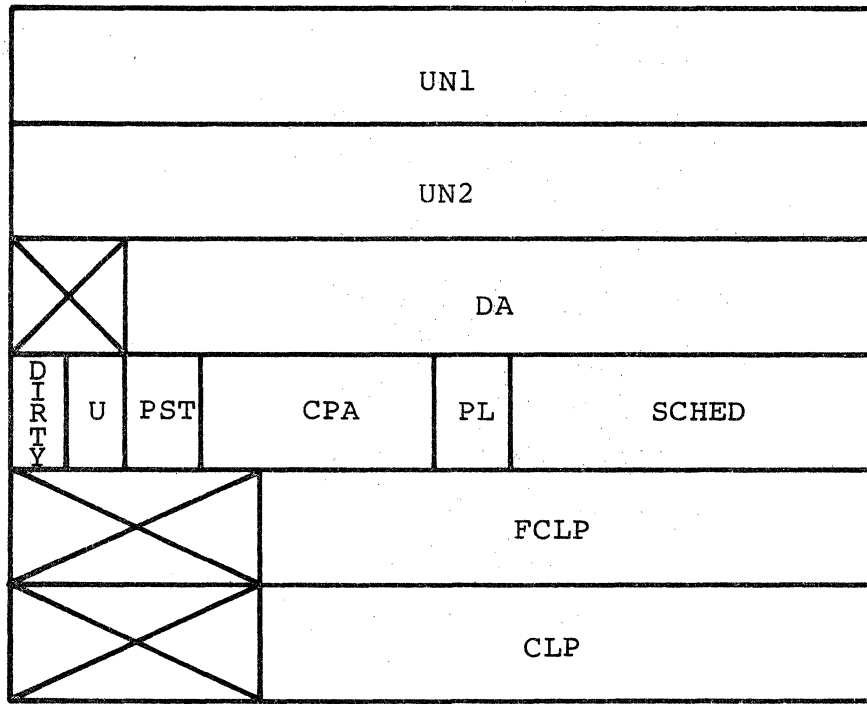
mainder of the entry is returned. The PA field is prefixed to the last 11 bits of the virtual address to make a read address. If the access is a store and PMRO = 1, the store is aborted and the PRO (Page Read Only) trap is caused. If the access is a store, PMRO = 0 and DB = 0, the dirty bit in the CHT entry for the page is set and DB is set to 1.

If the empty flag is on, the PM entry must be loaded. Let its index be  $i$ . First, entry  $i$  of the map (i.e. half-word  $400B + i$  in the context block) is fetched. If PMTI is 0, trap PNIM (Page Not In Map) occurs. If it is not 0 MAPRO [ $i$ ] is saved. Then the PMT entry specified by PMTI [ $i$ ] is fetched. Call it entry  $n$ . If SF[ $n$ ] = 0, trap PNIC (Page Not In Core) occurs. PMTRO is saved; if PREF [ $n$ ] = 0, it is set to 1; the UN found in PMT [ $n$ ] is then looked up in the core hash table.

The Core Hash Table contains information about the current contents of core memory. It starts at location  $400B$  in real core and is organized as a chained hash table. The table comes in two parts:

- 1) The index, called CHT1 which is an array of 256 pointers to lists of CHT entries. Each word of CHT1 is either END or the address of a CHT2 entry  $e$  with the property that HASH (UN( $e$ )) is the address of the CHT1 word. If there are several pages in CHT with the same value of HASH (UN), the CHT1 word points to one of them, which points to the next using the collision pointer field, and so on until all are chained into the list. The last element has END in its collision pointer. The hashing function HASH is to take the exclusive or of the six 8-bit bytes of the Unique Name (UN) and then the exclusive or of this result with  $264B$ .

2) The body, called CHT2, is an array containing a 6 word entry for each page of real core. Each entry has the form:



| <u>Word</u> | <u>Bits</u> | <u>Name</u> | <u>Contents</u>                                      |
|-------------|-------------|-------------|--|
| 0           | 0-23        | UN1         | First 24 bits of unique name                         |
| 1           | 0-23        | UN2         | Second 24 bits of unique name                        |
| 2           | 2-23        | DA          | Disk address of page                                 |
| 3           | 0           | DIRTY       | Dirty bit  |
| 3           | 1           | U           | Unavailable bit                                      |
| 3           | 2-4         | PST         | Page status  |
| 3           | 5-12        | CPA         | Core page address                                    |
| 3           | 13-15       | PL          | Page lock  |
| 3           | 16-23       | SCHED       | Number of occurrences of page in loaded working sets |
| 4           | 6-23        | FCLP        | Free core list pointer                               |
| 5           | 6-23        | CLP         | Collision PTR  |

FIGURE 3.5 CORE HASH TABLE ENTRY

If (U OR PST)  $\neq \emptyset$  or the page is not in CHT, trap PNIC occurs. If the page is found, CPA and DIRTY are copied into the PM and PMRO is set to MAPRO OR PMTRO [n].

All the traps (PRO, PNIM, PNIC) which can be generated by the mapping operation are given the virtual address being mapped as a parameter.

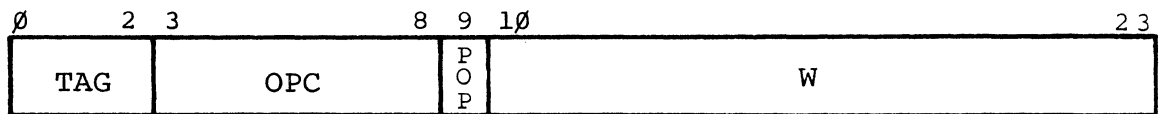
To make sure that a particular page is not being used by the CPU, an external processor may request a scan of the physical map. When such a request is received, the PA field of all non-empty registers in the physical map is matched against the contents of cell 2455B + CPU number \*4. If any of them matches, the MAB (Map ABort) trap occurs. The message cell is set to 4B7 upon completion of the scan, regardless of the outcome.



#### 4. Addressing from Instructions

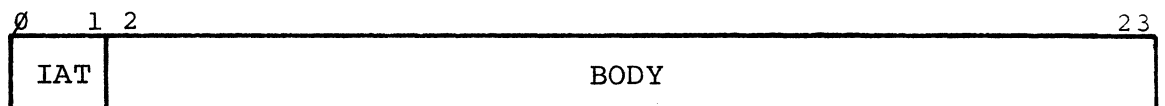
The machine has a rather complex addressing structure. The address calculation is performed in the same way for every instruction, and it may yield either an operand OP or an effective address Q. The format of an instruction and of an indirect address word (IAW) is as follows:

##### a) Instruction word format



| <u>Bit</u> | <u>Name</u> | <u>Contents</u>         |
|------------|-------------|-------------------------|
| 0-2        | TAG         | Address TAG field       |
| 3-8        | OPC         | Op code                 |
| 9          | POP         | Programmed operator bit |
| 10-23      | W           | Address field           |

##### b) Indirect address word format



| <u>Bit</u> | <u>Name</u> | <u>Contents</u>   |
|------------|-------------|---|
| 0-1        | IAT         | Tag field which defines the meaning of the rest of the word |
| 2-23       | Body        | The meaning depends on IAT                                  |

FIGURE 4.1 INSTRUCTION AND INDIRECT ADDRESS  
WORD FORMAT

Since the addressing is rather complex, it seems worthwhile to explain in some detail what the various features are for, before describing them precisely. There are a number of points which influenced the design:

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

2) Programs are normally written in relatively small units, each of which references some private storage of its own and some global storage.

3) Array references are very common. Since there is only one index register for holding subscripts, it would be very nice to have a convenient way of using core locations for indexing. Since the languages which are expected to account for a majority of the load on the machine require subscripts to be checked for size before being used, it would be nice to have a cheap and convenient way of doing this. Furthermore, we have to deal with arrays having elements which may occupy 1 (integer), 2 (real), or 4 (double) words. To have to multiply the index by the element size is a great annoyance.

4) References to fields which occupy whole words or parts of words relative to a pointer are also common, especially in system code.

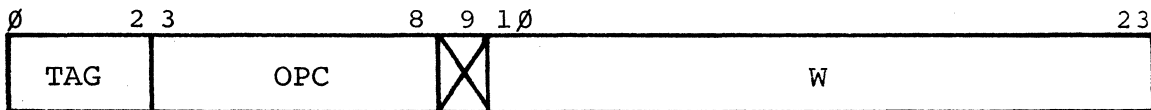
5) It is essential to have an effective mechanism for handling strings of 8-bit characters. If other byte sizes can also be accommodated, so much the better.

All of these goals are achieved in a fairly economical way by the addressing system. In particular, arrays, strings, and part-word fields are handled by indirect addressing, which allows an absolute 18-bit address to be supplied. The addressing modes available in an instruction allow for immediate operands, addressing relative to the instruction word for referencing the program, and addressing relative to two base registers which are intended to reference the local storage of the subroutine (called the local environment, L) and the global storage of the whole program (called the global environment, G). They also permit indexing to be specified from the X-register or from the first few cells of the local or global environment.

It should be obvious by now that the addressing system is designed to be used by programs which are organized in a very definite way, i.e., into a collection of subroutines or functions (of less than 4K words each), each with local storage (of less than 2K words for scalars), and all with access to a single global storage and communications area (of less than 16K words). The first 128 words of the local and global environments are special; this is because there are 8-bit fields in certain addresses in which the top bit specifies L or G and the remaining 7 bits address one of the first 128 words. The first 32 words are even more special, because there are 6-bit fields in which the top bit specifies L or G and the remaining 5 bits address one of the first 32 words. With this introduction, we proceed

to describe the addressing in detail, together with comments on the intended use of each feature. A reader unfamiliar with this material will find it helpful to read the text following the description of each mode first.

The 3-bit TAG field of an instruction determines one of 8 addressing modes.

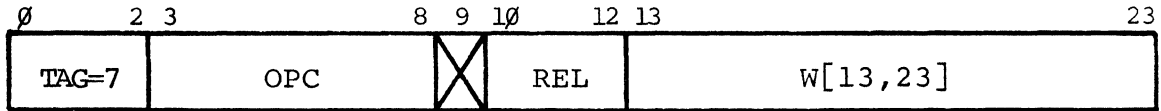


| <u>TAG</u> | <u>Name</u> | <u>Addressing Mode</u>              |
|------------|-------------|-------------------------------------|
| 0          | D           | Direct or G-relative                |
| 1          | I           | Indirect or G-Indirect              |
| 2 ,        | X           | Indexed                             |
| 3          | BX          | Base-Index                          |
| 4          | PD          | Pointer-Displacement                |
| 5          | PDI         | Pointer-Displacement-Indirect       |
| 6          | BXD         | Base-Index-Displacement             |
| 7          | REL         | Relative. This one has 6 sub-cases. |

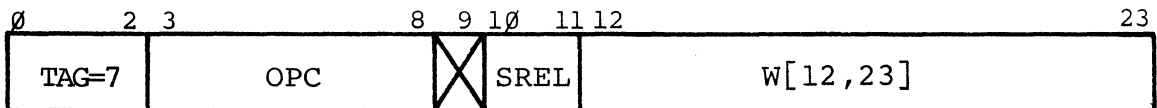
FIGURE 4.2 ADDRESSING MODES SPECIFIED  
BY TAG FIELD

The relative mode has 6 sub-cases, L-relative, source-relative, immediate, indirect L-relative, indirect source-relative and immediate indexed.

The relative words are formatted as follows:



| <u>REL</u> | <u>Name</u> | <u>Addressing Mode</u> |
|------------|-------------|------------------------|
| 0          | LR          | L-relative             |
| 1          | LRI         | L-relative Indirect    |
| 6          | IMX         | Immediate-Indexed      |
| 7          | IM          | Immediate              |



| <u>SREL</u> | <u>Name</u> | <u>Addressing Mode</u>   |
|-------------|-------------|--------------------------|
| 1           | SR          | Source-relative          |
| 2           | SRI         | Source-relative Indirect |

FIGURE 4.3 RELATIVE ADDRESSING MODES SPECIFIED BY REL AND SREL

Notice that we have represented the 6 sub-cases of the relative mode by introducing two fields called REL and SREL. This is because in the source relative modes of addressing bit 12 is used as part of the address field. Also we have introduced some new notation. The W[13,23] indicates that we are referring to bits 13 through 23 of the instruction word and that those bits are contained within the address field W of the instruction.

Most of the modes depend on the existence of an indexing register IR, and a source register R. The IR register is not to be confused with the index register X. In fact, it is not part of the state at all; i.e. its value does not have to be preserved from one instruction to the next. The IR is used to hold the 18-bit value which will be used when an indexing operation is called for by the addressing system. It is initialized from X at the beginning of each instruction. Thereafter, it may be loaded from a word specified by a BX or BXD mode or an array indirect word (see below). The source register is initialized to the address of the word from which the instruction has been fetched (normally P).

Some addressing modes compute Q directly from the information in the central registers, the instruction and possibly one memory word used for indexing. Others (the indirect modes) compute directly the location of an indirect address word, and the contents of this word then determines how the addressing computation is to proceed. If indirect addressing is specified, only the values of the IAW address and IR affect the subsequent address computation. We will therefore confine ourselves to specifying those values which describe instruction addressing, and leave the details of indirect addressing for later treatment.

Before we describe the various addressing modes in detail, we define some notation that will be used in defining the various modes.

CONTENTS(N) will be used to denote the contents of the memory location with address N. Ring checking is performed with R as source and N as target.

IA(N) implies 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 Q or OP.

Note that special cases of the IA(N) function may be specified for each of the 4 indirect addressing modes where we may want to indicate a particular mode. We may have any one of the following:

```

NORMAL'IA(N);
FIELD'IA(N);
STRING'IA(N);
ARRAY'IA(N);

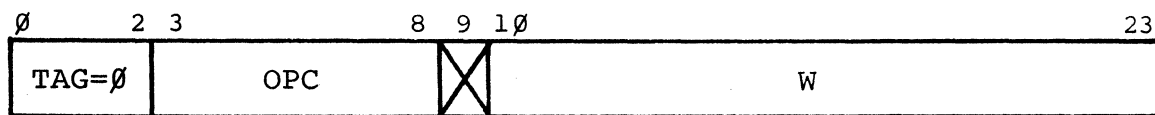
```

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].

SIGNED(W[i,j]) means W[i,j] interpreted as a two's complement number of (j - i + 1) bits.

All instructions start with  $IR \leftarrow XR \ \& \ R \leftarrow P$ ; in the notation used, the "\$" indicates indirection and "'" (e.g.  $G'[w]$ ) is used in the sense of a delimiter between symbols. We now define in detail all the addressing modes with indirect addressing discussed in a separate section which follows.

Direct (D) or G-relative:



$Q \leftarrow W + G;$

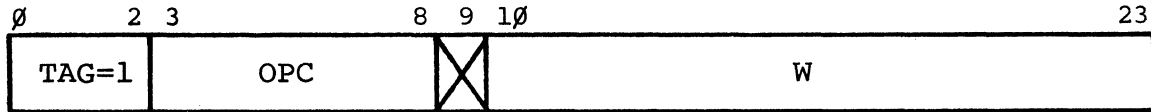
$OP \leftarrow \text{CONTENTS}(Q);$

In the direct or G-relative mode, the effective address is given by the 14-bit address field relative to G. This permits direct addressing of the first 16K of the global environment. The notation in SPL is

$OPC \ G'[W];$



Indirect (I) or G-relative Indirect:



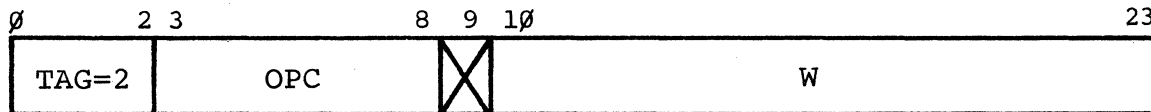
IA(W + G);

In the indirect mode, any of the first 16K words of the global environment can be used as an IAW (indirect address word) that may point anywhere in the virtual address space.

The notation is

OPC \$G'[W];

Indexed (X):



Q ← W + IR;

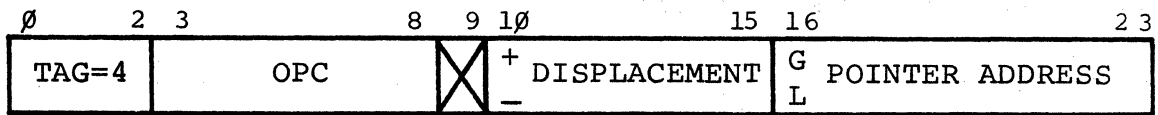
OP ← CONTENTS(Q);

Since IR is initialized by XR, the effective address is the (18-bit) sum of the indexing register and the address field.

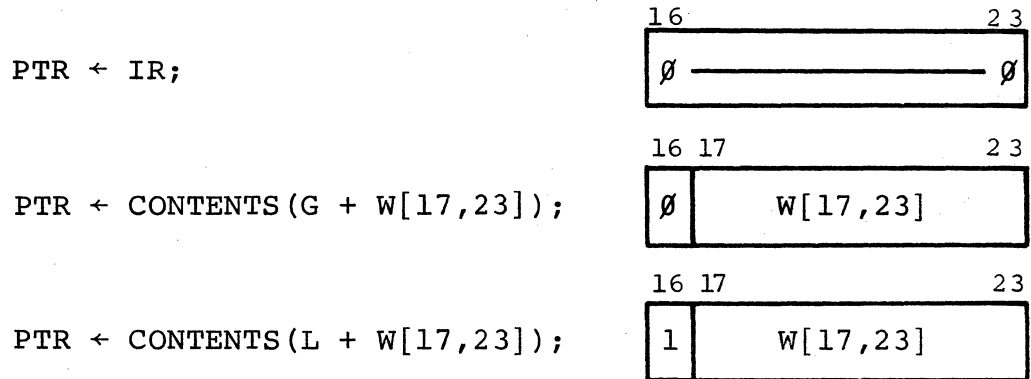
The notation is

OPC X'[W];

Pointer-Displacement (PD):



Where the pointer field is one of the following:



and the address calculation is:

```

PTR  $\leftarrow$  IR IF W[16,23] =  $\emptyset$  ELSE
PTR  $\leftarrow$  CONTENTS(G + W[17,23]) IF W[16] =  $\emptyset$  ELSE
PTR  $\leftarrow$  CONTENTS(L + W[17,23]);

DISP  $\leftarrow$  SIGNED(W[10,15]);

Q  $\leftarrow$  PTR + DISP;

OP  $\leftarrow$  CONTENTS(Q);

```

In this mode the address field is divided into an 8-bit pointer address field and a 6-bit signed displacement field. Similar arrangements are used in several other modes; they will be explained here in detail. The top bit of the 8-bit pointer address specifies the environment (1=local,  $\emptyset$ =global) and the remaining 7 bits address one of the first 128 words in the local or global environments. If pointer address is  $\emptyset$ , the contents of IR, rather than of word  $\emptyset$  in G, is specified. The calculation of DISP specified the conversion of a 6-bit number which is to be interpreted as two's complement into a 24-bit two's complement number.

Finally, the effective address is the sum of the pointer (PTR) specified by pointer address and the displacement (DISP). The typical use of this mode is in addressing the  $n$ th word of a table entry given a pointer to the start of the entry. If the pointer  $P$  is in the first 128 words of either environment, then the word is loaded into  $A$ , say by

LDA P[D]

which is the notation for PD addressing with pointer address  $P$  and displacement  $D$ .

The notation is

OPC P[D];

where  $P$  may be one of the following:

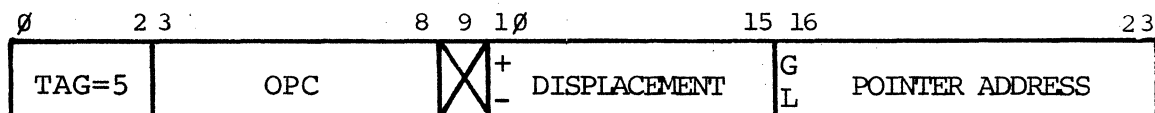
$\emptyset$  or  $G'[\emptyset]$  for  $PTR \leftarrow IR$ ;

$G'[N]$  where  $\emptyset \leq N \leq 127$  for  $PTR \leftarrow CONTENTS(G + N)$ ;

$L'[N]$  where  $\emptyset \leq N \leq 127$  for  $PTR \leftarrow CONTENTS(L + N)$ ;

and  $-32 \leq D \leq 31$

Pointer-Displacement-Indirect (PDI):



$Q \leftarrow PTR + DISP$  as for PD mode;

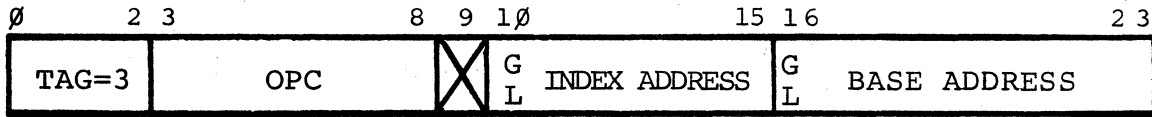
IA(Q);

This is just indirect addressing in PD mode.

The notation is

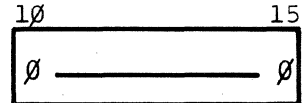
OPC \$P[D];

Base-Index:

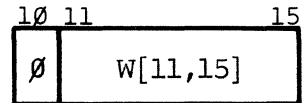


Where the index address field is one of the following:

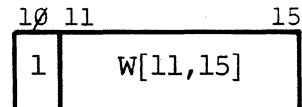
INDEX  $\leftarrow$  IR;



INDEX  $\leftarrow$  CONTENTS(G + W[11,15]);

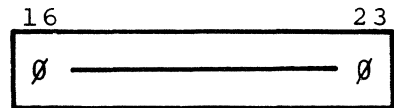


INDEX  $\leftarrow$  CONTENTS(L + W[11,15]);

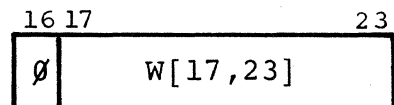


and the base address field is one of the following:

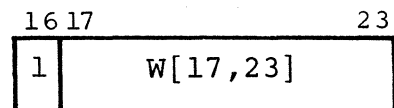
BASE  $\leftarrow$  IR;



BASE  $\leftarrow$  CONTENTS(G + W[17,23]);



BASE  $\leftarrow$  CONTENTS(L + W[17,23]);



and the address calculation is:

```

BASE  $\leftarrow$  IR IF W[16,23] =  $\emptyset$  ELSE
BASE  $\leftarrow$  G + W[17,23] IF W[16] =  $\emptyset$  ELSE
BASE  $\leftarrow$  L + W[17,23];

```

```

IR  $\leftarrow$  IR IF W[10,15] =  $\emptyset$  ELSE
IR  $\leftarrow$  CONTENTS(G + W[11,15]) IF W[10] =  $\emptyset$  ELSE
IR  $\leftarrow$  CONTENTS(L + W[11,15]);

```

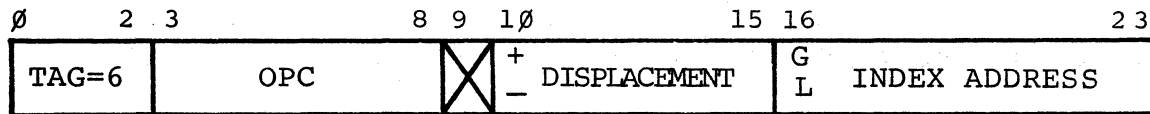
IA(BASE);

This is the array and part-word field mode and is written

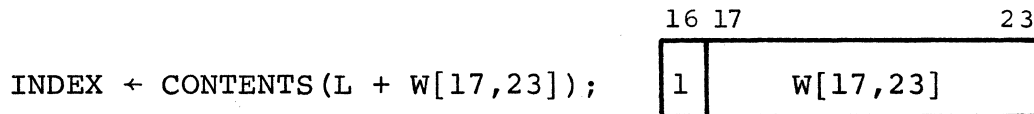
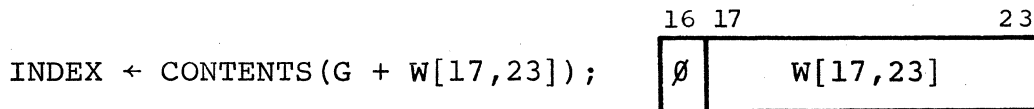
$$\text{OPC B}[I]$$

where B is the base and I the index. The 8-bit and 6-bit index are both treated as local or global environment addresses, exactly like the pointer address in PD mode. The index is put into IR and the base specifies an indirect word. If an array is being accessed, B will address an IAW which has the 18-bit base address of the array and specifies indexing. The contents of IR, which was loaded from I, will thus be added to the base address of the array to determine the final 18-bit address, which is just what we require for array referencing. This is not, however, the whole story; the rest will be told when we come to consider the indirect addressing type used for arrays.

Base-Index-Displacement (BXD):



Where the index address field is one of the following:



and the base address is in the indexing register.

The address calculation is:

BASE  $\leftarrow$  IR;

INDEX  $\leftarrow$   $\emptyset$  IF W[16,23] =  $\emptyset$  ELSE

INDEX  $\leftarrow$  CONTENTS(G + W[17,23]) IF W[16] =  $\emptyset$  ELSE

INDEX  $\leftarrow$  CONTENTS(L + W[17,23]);

DISP  $\leftarrow$  SIGNED(W[10,15]);

IR  $\leftarrow$  INDEX + DISP;

IA(BASE);

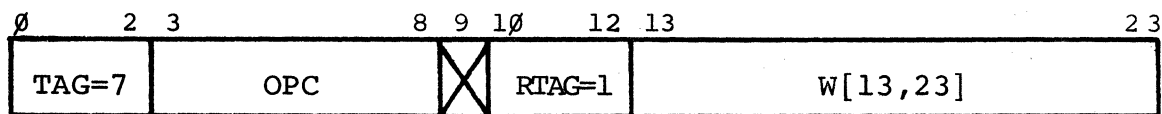
This mode is similar to BX. It assumes that the base address is in the IR. The field thus freed is used to provide a displacement (anything from -32 to +32) of the index. Thus to load B[I + 5] we would write

```
EAX    B
LDA    ($X')[I + 5];
```

where I is the index address, 5 the displacement. See the discussion of arrays in section (5) for more details on the BX and BXD addressing modes.



L-relative indirect (LRI):



DISP ← W[13,23];

Q ← L + DISP;

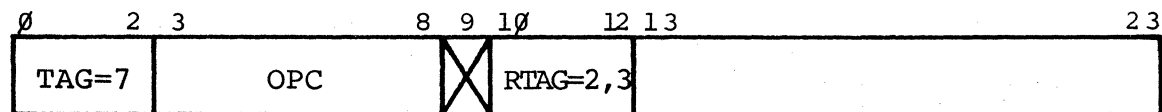
IA(Q);

This is simply the indirect counterpart for the L-relative mode.

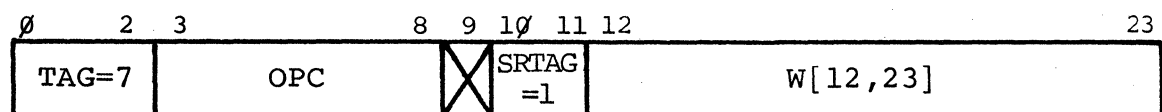
The notation is

OPC \$L'[D];

Source-relative (SR):



causes the instruction to be interpreted as



DISP ← SIGNED(W[12,23]) ;

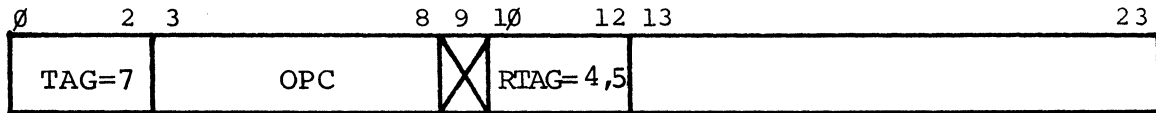
Q ← R + DISP;

OP ← CONTENTS(Q);

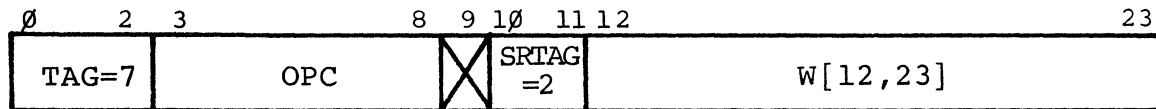
This mode allows location up to 4000B on either side of the instruction to be addressed. Remember R is initialized to the program counter at the start of the address calculation.



Source-relative-indirect (SRI):



causes the instruction to be interpreted as



DISP ← SIGNED(W[12,23]);

Q ← R + DISP;

IA(Q);

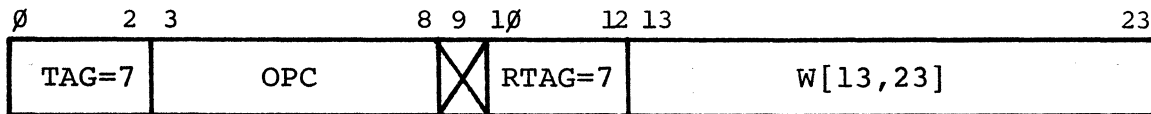
This is just the indirect counterpart for source relative.

The notation is

OPC \$R'[D];

All the relative modes allow routines to be placed anywhere in memory without modification and to address themselves without difficulty, as long as they are not more than 2048 words long.

Immediate (IM):



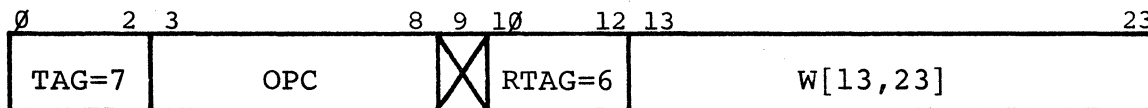
$OP \leftarrow \text{SIGNED}(W[13,23]);$

The immediate mode permits signed constants in the range  $-20000B$  to  $1777B$  to be provided as operands without an additional memory reference. Stores are not allowed and the operand must not be larger than 11-bits.

The notation is

OPC I;

Immediate-indexed (IMX):



$OP \leftarrow IR + \text{SIGNED}(W[13,23]);$

This adds the contents of IR to the immediate operand.

The notation is

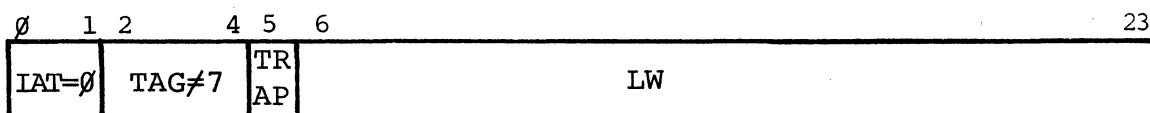
OPC X' + I;

## 5. Indirect Addressing

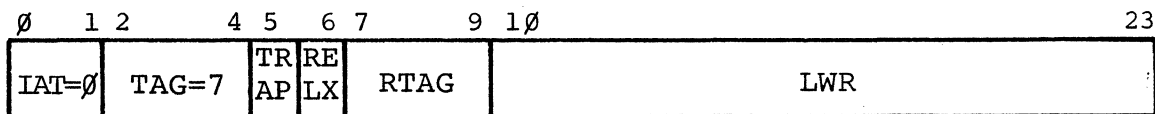
To prevent infinite loops of the indirect mechanism, a trap, ILIM (Indirect LIMit exceeded), will occur if indirection through more than 16 levels is attempted.

There are four types of indirect addressing: normal, field, string, and array. The type is selected by the first two bits of the word. The intended use of each type is suggested by its name and will now be explained in detail.

Normal: the IAW has the form



or



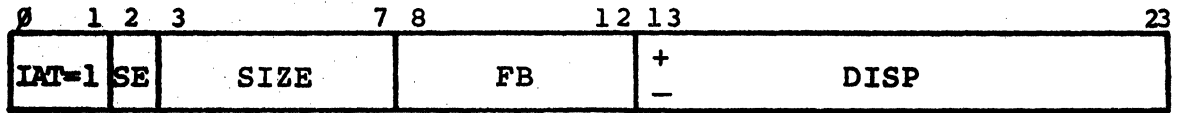
| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                             |
|-------------|-------------|---|
| 0-1         | IAT         | Indirect Address Type zero                  |
| 2-4         | TAG         | interpreted exactly like an instruction TAG |
| 5           | TRAP        | causes trap IATRP if set                    |
| 6           | RELX        | causes indexing for relative modes          |
| 10-23       | LWR         | long address for the relative modes         |
| 6-23        | LW          | long word address                           |

If TRAP is set, the IATRP trap is caused, and R is passed as its argument. Otherwise, TAG and W are interpreted as in an instruction word, with three exceptions:

- 1) if TAG = D, I, or X, LW is used in place of W, and G is not added. In other words, an 18-bit absolute address is supplied.
- 2) if TAG = REL, IR is added to the addresses computed by L and R-relative modes if RELX is set. I.e., indexing is possible with these modes. Also, the 3-bit subtag is found in bits 7-9, thus allowing the LR, LRI, SR, and SRI offsets to be 3 bits longer.
- 3) if TAG = PD or PDI, the mode is read-only direct (ROD) or read-only X-relative (ROX) respectively. These behave exactly like D and X modes except that an attempt to store will cause the ROIA trap with R as parameter.

Normal type permits any word in the address space to be addressed directly. It is generally used for pointers and for the addresses of arrays. Note that although the capabilities are almost identical to those provided by an instruction address, the format is quite different. It is not possible to use an instruction as an indirect word. It also permits indexing of a L-relative or source-relative address, so that arrays in the program of the local environment can be addressed conveniently.

Field: the IAW has the form



| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                           |
|-------------|-------------|---|
| 0-1         | IAT         | 1   |
| 3-7         | SIZE        | size of field in bits                     |
| 8-12        | FB          | address of first bit of the field         |
| 2           | SE          | causes sign extension of the field if set |
| 13-23       | DISP        | 2's complement signed displacement        |

FIELD:  $Q \leftarrow IR + DISP;$   
 $U \leftarrow CONTENTS(Q);$   
 $OP \leftarrow U [FB, FB + SIZE - 1];$   
 $OP \leftarrow OP - 2^{**}(24-FB) \text{ IF } SE = 1 \text{ AND } OP [FB, FB] = 1;$

The field which is SIZE bits in length and which starts at bit FB in word DISP + IR is referenced. Both FB and FB + SIZE - 1 must be  $\leq 23$ . If they are not a TI trap will occur. If SE is set, the leftmost bit of the field (bit FB at DISP + IR) will be extended into bits 0 through 23-SIZE of the resulting operand. DISP is taken as a 2's complement number, in the range -1024 to 1023.

The idea here is that IR contains a pointer to a table entry, and that the field descriptor (the IAW) specifies a group of bits at some definite location in the entry. Typically, the pointer might be in PTR within 32 words of L and the field descriptor in F within 128 words of G. Suppose the contents of F is

FIELD                    3: 6, 12

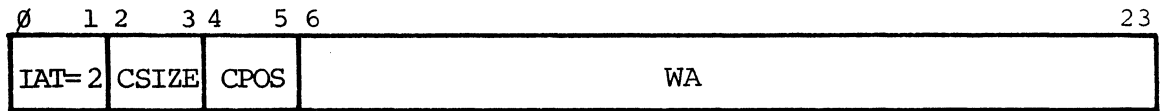
or in octal              DATA                    21640003B

then we might write

LDA                      F [PTR]

using base-index addressing. Since PTR appears in the index field, its contents is put into IR. Then F is taken as an IAW. Since it is of type field, it accesses the word at IR + 3, which is CONTENTS (PTR+3); i.e., the fourth word of the object pointed to by PTR. Bits 6 - 12 of this object will be loaded into A. If the word addressed was 01234567B, then A will contain 47B. The field can be used as an operand in any instruction which accesses a single-word operand, this includes both load and store types. Note that fields cannot cross word boundaries.

String: the IAW has the form



| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                                      |
|-------------|-------------|--|
| 0-1         | IAT         | 2  |
| 2-3         | CSIZE       | character size: 0 = 6 bits, 1 = 8,<br>2 = 12, 3 = 24 |
| 4-5         | CPOS        | character position in word                           |
| 6-23        | WA          | word address   |

The character at the indicated position in the word addressed by WA is referenced. The following table defines what bits are referenced by the 16 possible combinations of CSIZE and CPOS.

| <u>CSIZE/CPOS</u> | <u>0</u> | <u>1</u> | <u>2</u> | <u>3</u> |
|-------------------|----------|----------|----------|----------|
| 0                 | 0-5      | 6-11     | 12-17    | 18-23    |
| 1                 | 0-7      | 8-15     | 16-23    | X        |
| 2                 | 0-11     | 12-23    | X        | X        |
| 3                 | 0-23     | X        | X        | X        |

Combinations marked X in the table will cause a TI trap.

The bits referenced are treated exactly like the bits selected by a field IAW.

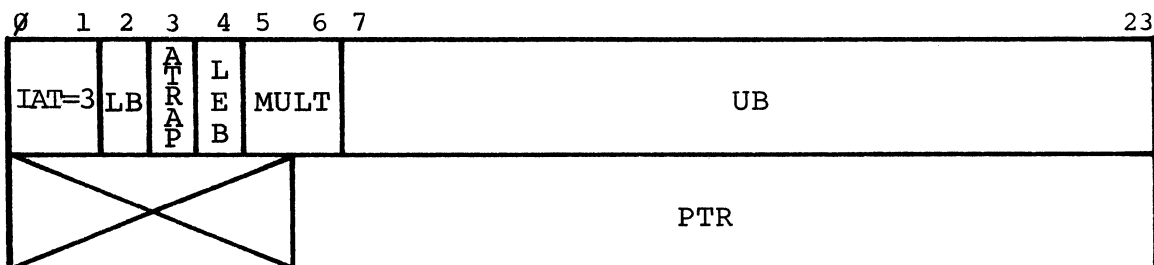
This type of indirection allows one byte in a string to be referenced. The instruction ISD increments the descriptor to point to the next byte, which may then be referenced. It has the

additional feature of setting the condition code depending on whether the descriptor is equal to the next word or not. The string type and this instruction are intended to be used with four-word string descriptors. The first word points just before the first byte allocated for the string. The second word (read pointer, RP) points to the first character of the string, the third word (write pointer, WP) to the last character. The fourth word points to the last byte allocated for the string. To read the first character, increment RP with ISD, then indirect through it. The case of no characters left can be detected by the abnormal CC setting. To write a character, increment WP with ISD and then store indirect through it. Overflow of available storage can be detected by the CC setting.



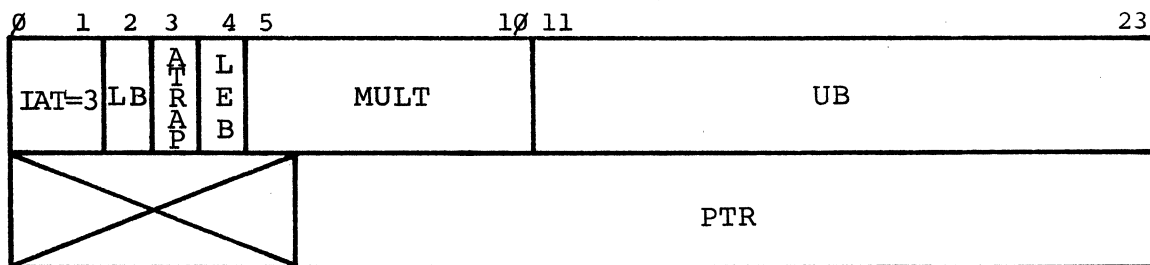
Array: an array descriptor is two words long. Its form is:

LEB =  $\emptyset$



or

LEB = 1



| <u>Word:Bit</u>              | <u>Name</u>                  | <u>Contents</u>                        |
|------------------------------|------------------------------|--|
| $\emptyset$ : $\emptyset$ -1 | IAT                          | 3                                      |
| $\emptyset$ :2               | LB                           | lower bound for IR ( $\emptyset$ or 1) |
| $\emptyset$ :3               | ATRAP                        | array trap bit                         |
| $\emptyset$ :4               | LEB                          | large element bit                      |
| $\emptyset$ :5-6             | MULT IF<br>LEB = $\emptyset$ | multiplier for IR                      |
| $\emptyset$ :5-10            | MULT IF<br>LEB = 1           |  |
| $\emptyset$ :7-23            | UB IF<br>LEB = $\emptyset$   | upper bound for IR                     |
| $\emptyset$ :11-23           | UB IF<br>LEB = 1             |  |
| 1:6-23                       | PTR P                        | pointer to array                       |

A multiplier of one is coded in the descriptor as zero, two as one, etc. If  $IR < LB$  or  $IR > UB$ , trap ABE occurs, with R as parameter. If  $ATRAP = 1$  in IAW and the instruction is not LAX, or  $ATRAP = \emptyset$  and the instruction is LAX, trap IATRP occurs with R as parameter.

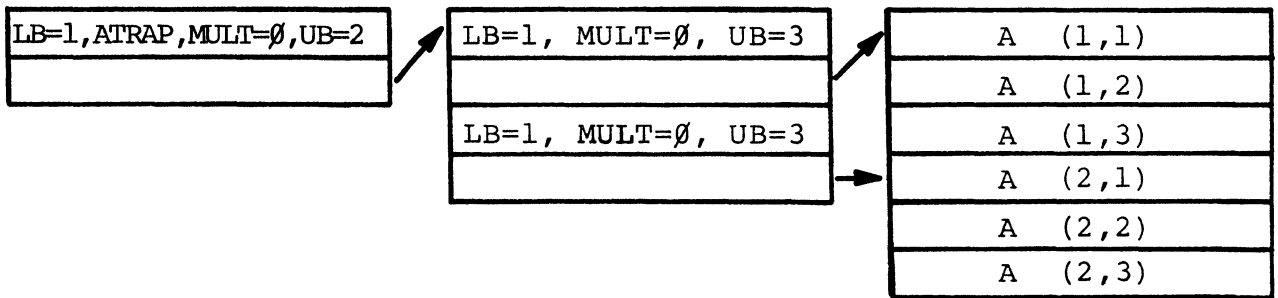
otherwise,  $IR \leftarrow (IR - LB) * (MULT + 1); T \leftarrow R + 1;$   
 NORMAL'IA(T);

Where NORMAL'IA indicates IA(T) of type normal. This is the most complicated of the IAW types. It is intended to accomplish the following functions connected with array accessing:

- 1) Allow  $\emptyset$  or 1 as lower bound
- 2) Perform a bounds check on the subscript
- 3) Multiply the subscript by the size of the array element, allowing for sizes up to 64
- 4) Check that the number of subscripts supplied is the number expected (see below)
- 5) Provide an 18-bit absolute base address for the array.

Arrays are intended to be stored with marginal indexing. Thus the 2 x 3 one-origin integer array A would appear as follows:

A =



(The three 2-word descriptors are array indirect words.)

The LAX instruction works just like EAX, except that it merges an X tag in XR[2,4] (leaving a normal IAW which specifies indirection) and treats the TRAP bit in an array descriptor as though it were complemented.

Then to do  $B \leftarrow A[K,L]$  we would write

```
LAX      A[K]      (BX addressing)
```

which leaves the address of the descriptor for the Kth row in X followed by

```
LDA      ($X')[L]  (BXD addressing)
STA      B
```

The second subscript can have a constant displacement without complicating things:

$B \leftarrow A[K, L-4]$  becomes

LAX      A[K]

LDA      (\$X')[L-4]

STA      B

If the first subscript has a displacement, there is a complication, since there is not enough room for three operands in one instruction.

$B \leftarrow A[K+1, L]$  becomes

EAX      A

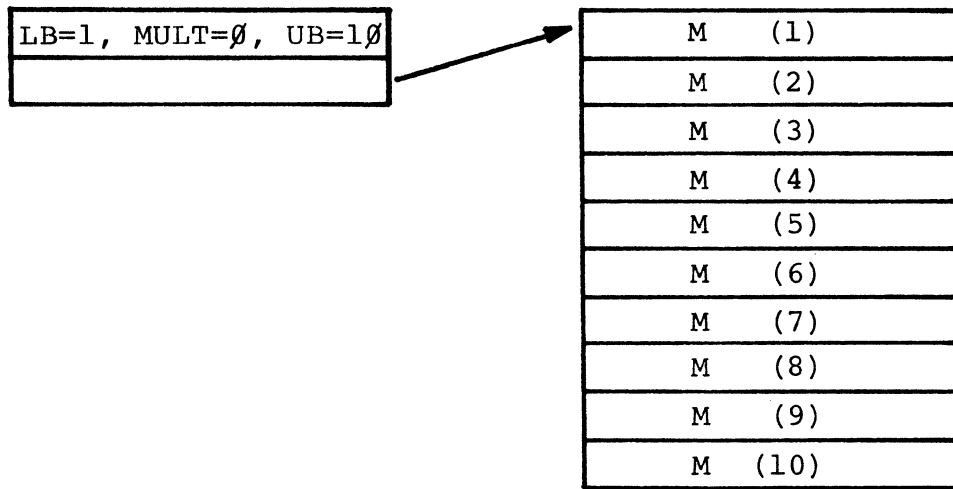
LAX      (\$X')[K+1]

LDA      (\$X')[L]

STA      B

A single subscripted array can be accessed without any extra instructions at all provided the subscript is a variable which can be accessed with an index field of the BX mode. If M is a 10-element integer array, it is allocated thus:

M =



and  $N \leftarrow M[J]$  becomes

```
LDA M[J]
```

```
STA N
```

If the array is integer (1 word items) and bounds checking is not required, the descriptors can be changed to normal indirect words which specify indexing, and no change is required in the instructions of the program.

The purpose of the peculiar behavior of LAX in the case of traps is to check that the proper number of subscripts is provided to an array. The trap bit should be set in the array descriptors except at the last level (the descriptors which point directly to the data) and clear there.

## 6. Use of Addresses by Instructions

All the instructions compute an effective address  $Q$  and/or an operand  $OP$  as described above. The use of these quantities once they have been computed, and in particular the error conditions which may arise, depend on the address type of the instruction.

There are four address types:

### 1) Fetch type (F)

These instructions will accept any kind of address. They make use of the 24-bit  $OP$  value and possibly  $Q$ .

### 2) Effective-address type (E)

These instructions make use only of the effective address  $Q$ , ignoring  $OP$ . Immediate addressing causes a TI trap if used with these instructions.  $Q$  is ring-checked with  $R$  as a source before use; if the check fails a trap  $MACC$  will occur.

### 3) Store type (S)

These instructions make use of the effective address  $Q$  and the operand  $OP$ . If the address calculation terminated with indirection through a field or string descriptor, the  $FB$  and  $SIZE$  (for a field) or  $CPOS$  and  $CSIZE$  (for a string) define a group of bits, say bits  $i$  to  $j$ . An  $S$  type instruction puts bits  $23-j+i$  to  $23$  of the word to be stored into bits  $i$  to  $j$  of the word addressed by  $Q$ , leaving the rest of this word untouched. Immediate addressing causes a TI trap and indirection through a read-only direct or read-only indexed word causes a  $RO$  trap.

### 4) Double-store type (D)

These instructions make use only of the effective address  $Q$ . They trap under the same conditions as  $S$ -type instructions.

Note that they are not affected by field or string indirection.

Legal combinations of instructions and addresses are summarized in the following table:

|                                | F  | E  | S  | D  |
|--------------------------------|----|----|----|----|
| Immediate                      | ok | TI | TI | TI |
| Indirection through ROD or ROX | ok | ok | RO | RO |
| Anything else                  | ok | ok | ok | ok |

Instructions of types S or D will give a PRO trap if Q (or Q+i) for instructions which reference double (i=1) or quadruple (i=1,2,3) words) addresses a read-only page.

## 7. Function Calls

A rather elaborate mechanism for calling functions and returning from them is provided in the hardware of the machine. The purpose is to include all the capabilities required by the FORTRAN and SPL languages directly in the hardware, so as to make software interpretation unnecessary. This is considered extremely important, since programs are expected to be written in small modules, and function calls and returns are consequently expected to be very frequent.

The basic features of the call instruction, BLL, are as follows:

- 1) The old P-counter and local environment are saved and new ones are picked up.
- 2) The new local environment may occupy a fixed area, or it may be allocated space at the end of a stack defined by two locations in the global environment. There is a check for stack overflow.
- 3) The caller provides a list of parameter addresses. The called function specifies for each parameter whether it wants the address or the value copied into its local environment. If he requests copying the value, he specifies whether it is 1, 2, or 4 words.
- 4) He also specifies whether or not a parameter is an array. The calling program tells whether it is passing a scalar variable, a scalar value (stores are not legal), an array or an array element (subscripted array). These distinctions permit all the checking for proper matches of arrays with scalars re-



quired by FORTRAN to be done automatically. The case of an actual parameter which is an array element corresponding to a formal parameter which is an array requires software handling and is trapped so that this may be accomplished.

5) Provision is made for an argument to be passed in the central registers.

A number of these points are somewhat subtle and cannot be properly understood unless explained in complete detail, which we now proceed to do.

The BLL instruction addresses a branch descriptor, which is a two-word object with the following form:

| <u>Word</u> | <u>Bit</u> | <u>Name</u>         | <u>Meaning</u>   |
|-------------|------------|---------------------|--|
| 0           | 0-23       | NEWPW               | This word looks like an IAW.<br>Its effective address is computed.                 |
| 0           | 4          | SREL                | c.f. REL + SR in Normal IAW  |
| 0           | 5          | TRAP                | Causes IATRP if set  |
| 0           | 9-23       | SRW                 | Signed displacement if SREL is set   |
| 0           | 6-23       | LW                  | Long word addresses  |
| 1           | 0          | CLL                 | Call bit. The old P and L are saved if the bit is set.                             |
| 1           | 1          | STK                 | The local environment is allocated from the stack if this bit is set.              |
| 1           | 2          | CPA                 | Arguments are copied if this bit is set  |
| 1           | 3          | CPR IF<br>CLL = 1   | The CPA bit in the return descriptor is turned on if this bit is set.              |
| 1           | 3          | UWSTK IF<br>CLL = 0 | Unwind stack on return.  |
| 1           | 4          | REL                 | Source relative label is supplied  |
| 1           | 5          | FTN                 | 1 FORTRAN type function  |
| 1           | 6-23       | E                   | This number determines the new L; precisely how it does so depends on STK and REL. |

When BLL is executed, the first step is to compute the effective address of NEWPW (which is LW if SREL is  $\emptyset$ , otherwise the sign-extended SRW + the address of the NEWPW). This 18-bit number is saved in a temporary register called NEWP; after undergoing further processing it will become the new P-counter. The following steps remain to be performed:

- 1) Obtain new local environment.
- 2) Copy arguments.
- 3) Compute return descriptor (for CALL) and save it in  
first two words of new local environment.
- 4) Transfer control.

We treat them in the order written, which is also the order in which they are performed. In describing what happens, we shall make use of a number of temporary registers or variables (such as NEWP, which was introduced above).

- 1) If  $STK=\emptyset$ , the E field of the descriptor is taken as the new value of L, which we call NEWL. In this case, the function being called is said to have a fixed local environment. Such a function cannot be recursive, and space must be allocated for its local environment at all times. On the other hand, the contents of such a fixed environment is normally preserved between function calls. A FORTRAN function has a fixed environment, for example. Since a call (CLL=1) saves the current L in the E field of the return descriptor, the return (CLL= $\emptyset$ ) handles E exactly as the call of a fixed function does.

If  $STK=1$ , space for the environment is allocated on a stack. Two words are required to describe the stack, which grows toward increasing memory addresses:

$SP$ , the address of the first unused word, kept in  $G'[2]$ ,  
the third word of the global environment  
 $SL$ , the address of the last word allocated for the stack,  
kept in  $G'[3]$ .

If the environment is stacked, different actions are required for calls and returns.

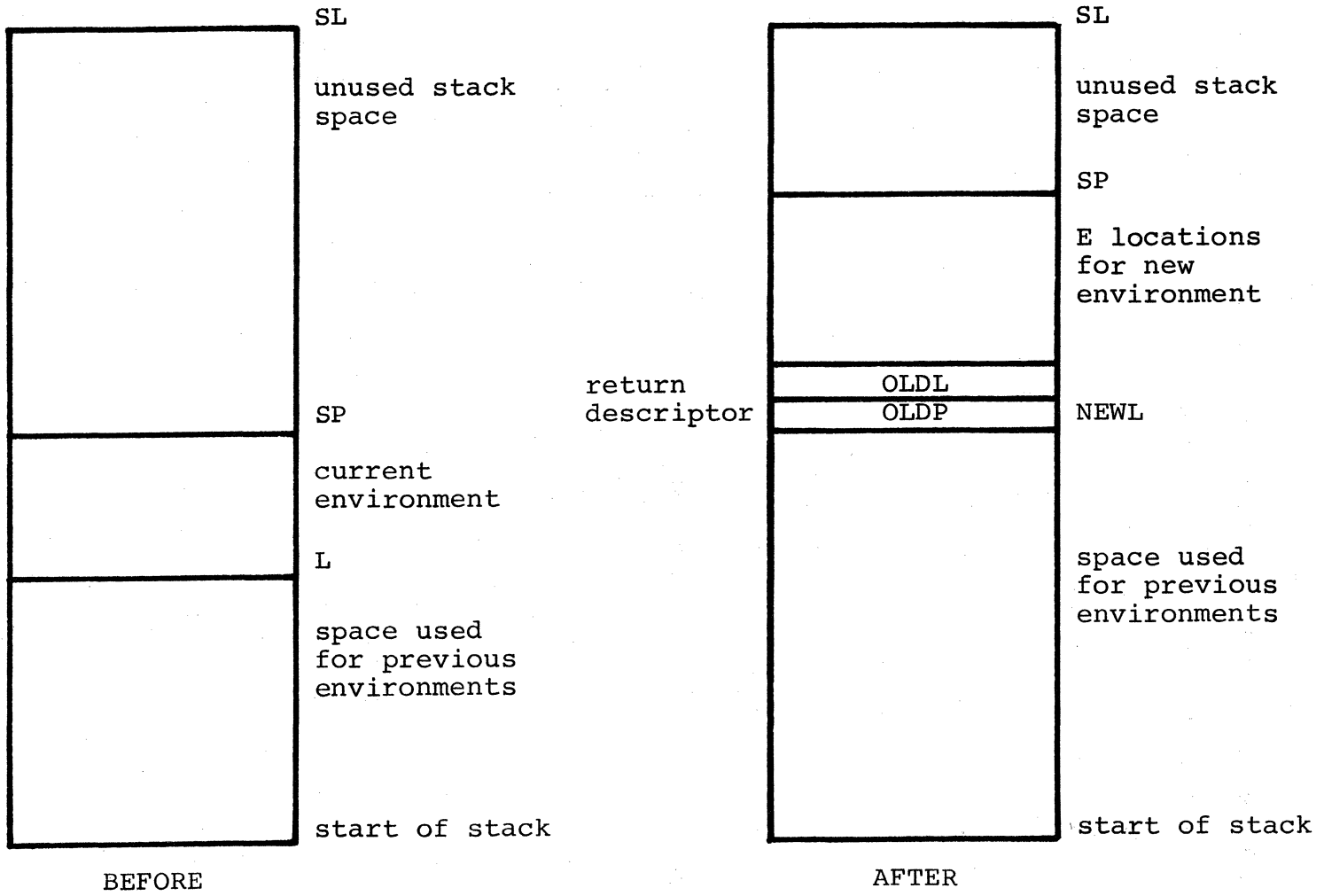
On a call ( $CLL=1$ ); we compute  $SP+E$ . If it is  $\geq SL$ , the  $STKOV$  trap occurs. Otherwise,  $NEWL \leftarrow SP$  and  $SP \leftarrow SP+E$ . In other words,  $E$  locations are taken from the top of the stack. The situation before and after is shown in figure 1.

On a return ( $CLL=\emptyset$ ) what ordinarily happens if  $STK$  is set is

$SP \leftarrow L$ ;  $NEWL \leftarrow E$ ;

in other words, the old  $L$  at the time of the call (which was saved in the  $E$  field of the return descriptor, as we will see) becomes the new  $L$ , and  $SP$  is reset to the value it had before the call, which is the current  $L$ . The before and after pictures of figure 1, looked at in the opposite order, should help to clarify this. With these rules, calls can be made freely from fixed environment functions to stacked environment ones and visa-versa. The industrious reader may check the four cases.

FIGURE 7.1 ALLOCATING A LOCAL ENVIRONMENT ON THE STACK DURING A CALL



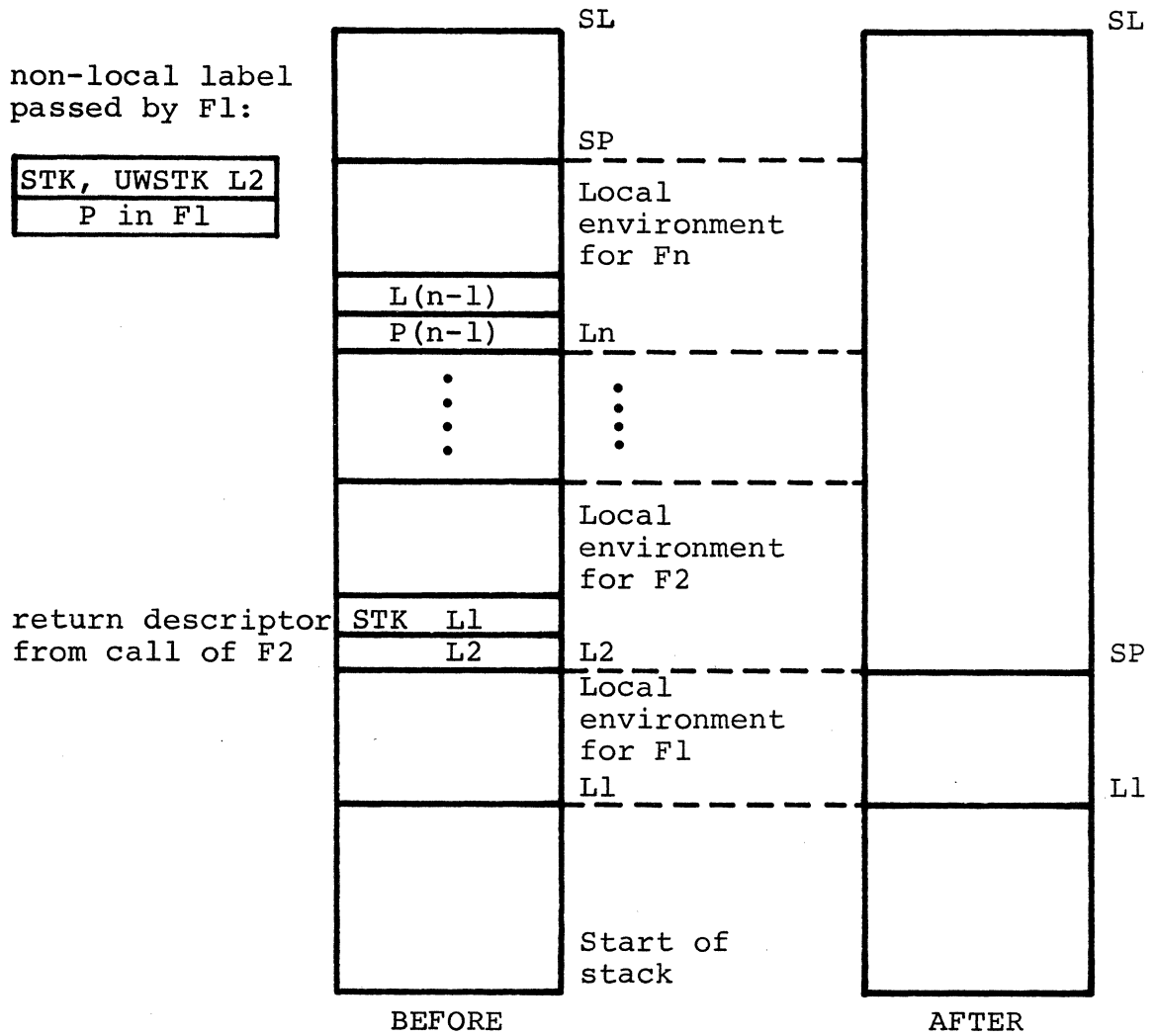


FIGURE 7.2 RETURN TO NON-LOCAL LABEL IN F1  
PASSED AS A PARAMETER

Unfortunately, if the return is to a function which is not the one which called the current one, SP is not reset correctly. This is expected to happen only as the result of a branch to a label which has been passed as a parameter (i.e. an error return). When such a parameter is passed (see below) from function F1 with L=L1 to F2 with L=L2, and the descriptor for the call has STK set, the parameter appears in F2 as a BLL descriptor with STK set, UWSTK set and L2 in E (see figure 2). The return (BLL) sees CLL= $\emptyset$ , STK=1, UWSTK=1 and does.

SP ← E; NEWL = the E field of the descriptor addressed by E. This trick allows both SP and L to be set correctly while carrying only one number in the descriptor.

It works regardless of whether F1 and Fn have fixed or stacked environments, but requires F2 to have a stacked environment. When a label is passed to a routine which has a fixed environment, therefore, E is set to L1 and STK, REL turned off. If additional space is allocated on the stack after the call, it will not be freed when a branch is made to this label. It is believed that this deficiency is not very serious.

2) If CPA=1, arguments are copied whenever a BLL is executed. If a function has multiple results, it will have CPR on in its descriptor. This will cause CPA to be turned on in the return descriptor, and the multiple results will be returned by the arguments - copying process when the return is executed. If CPA= $\emptyset$ , the BLLERR (2) trap occurs. A summary of all BLLERR traps and their parameters is given in the appendix. The BLLN

instruction should be used if no arguments are being passed; in this case the trap will occur if CPA=1.

The address of (actual) arguments to be copied are specified in the calling program in a list of actual argument words (AAWs) following the BLL instruction. These have a one-to-one correspondence with a list of formal argument words (FAWs) which starts at NEWP.

An argument word is formatted like an instruction. The addressing is interpreted exactly like the addressing for an instruction, but the 7-bit opcode field is treated differently, as follows.

| <u>Bits</u> | <u>Name</u> | <u>Contents</u>  |
|-------------|-------------|--|
| 3-4         | STR         | (actual argument only) structure<br>1 = variable<br>3 = computed scalar<br>2 = array element<br>Ø = array  |
| 3           | CADDR       | (formal argument only) copy value<br>1 = copy address of actual argument<br>Ø = copy value of actual argument  |
| 4           | FSTR        | (formal argument only)<br>1 = scalar<br>Ø = array  |
| 5-8         |             | type Ø = jump (actual argument only)<br>1 = integer (1 word)<br>2 = long (2 words)<br>3 = real (2 words)<br>4 = double (4 words)<br>5 = complex (4 words)<br>6 = longlong (4 words)<br>7 = string (4 words)<br>8 = label (2 words)<br>9 = pointer (1 word)<br>14 = unknown |
| 9           | ENDF        | end flag<br>Ø = not last argument word<br>1 = last argument word   |



Argument copying proceeds as follows: two pointers are initialized:

next formal argument word (NFW) initialized to NEWP

next actual argument word (NAW) initialized to P+1

Then  $FAW \leftarrow CONTENTS(NFW)$ , and FAW is treated as an instruction word for the purpose of computing its effective address, which is put into FQ. Only D or LR addressing is permitted; anything else will cause the BLLERR trap with class 4.

If  $ENDF(FAW) = \emptyset$ ,  $NFW \leftarrow NFW + 1$  and copying continues. Otherwise, copying stops. If the instruction is BLL, the BLLERR(2) occurs. If it is BLLN go to step (3).

We treat NAW as we treated NFW:  $AAW \leftarrow CONTENTS(NAW)$ ,  $R \leftarrow NAW$  and its effective address is computed. The address type is F if  $Type = 1$  (integer) otherwise E. BLLERR (5) will occur if the address type is not computable.

If  $type(AAW) = \emptyset$ , the AAW is a jump and its address specifies the next actual argument. Repeat from  $AAW \leftarrow CONTENTS(NAW \leftarrow Q)$ , etc.

If the AAW specifies G-relative addressing with an address of  $\emptyset$  it is taken to refer to the central registers. If  $CVAL \neq \emptyset$  then BLLERR(5) or if  $TYPE > 6$  or  $STR = \emptyset$  then BLLERR(4) will occur.

Next the types are checked. If  $\text{TYPE}(\text{FAW}) \neq \text{TYPE}(\text{AAW})$ , the  $\text{BLLERR}(3)$  trap occurs, unless one and only one of them is unknown.  $\text{FSTR}$  and  $\text{STR}$  are checked according to the following table:

| STR<br>FSTR |             | Array | Variable           | Array<br>Element | Computed<br>Scalar |
|-------------|-------------|-------|--------------------|------------------|--------------------|
|             | X           | 0     | 1                  | 2                | 3                  |
| Array       | $\emptyset$ | OK    | $\text{BLLERR}(3)$ | FTNAT            | $\text{BLLERR}(3)$ |
| Scalar      | 1           | FTNAT | OK                 | OK               | OK                 |

FTNAT means that if  $\text{FTN} = \emptyset$ ,  $\text{BLLERR}(3)$  occurs, otherwise the FTNAT bit is set, which will inhibit the skipping of one word in step (4).

The idea here is that if  $A[I]$  appears as an actual argument in FORTRAN and the corresponding formal  $B$  is dimensioned, an array descriptor for  $B$  must be computed, or if  $A$  appears as an actual argument and the formal is a scalar, the first element of the array must be found. A software routine is supposed to do this. It needs access to the descriptor for  $A$ ; the extra incrementing of  $\text{NAW}$  is to leave room for the address of the descriptor.

Now copying takes place. If  $\text{CADDR}(\text{FAW}) = 1$ ,  $Q$  is stored at  $\text{FQ}$  as an absolute IAW, except in the following two cases.

If the AAW supplied and immediate operand it is stored into  $\text{FQ}$  as an IM type Normal IAW,

If  $Q$  is the result of ROD or ROX addressing or  $\text{STR}(\text{AAW}) = 3$ ,  $Q$  is stored as a read-only absolute (ROD) IAW.

Otherwise, (CADDR(AAW) =  $\emptyset$ ) the value must be copied. The details of this depend on the type:

If TYPE = 1 and STR(AAW)  $\neq \emptyset$ , OP or the A register (in the special case) is copied to FQ.

For TYPE < 6 and STR(AAW)  $\neq \emptyset$ , the number of words specified above is copied from Q to FQ, or from the central registers (A, B, C, and D) to FQ if appropriate.

If TYPE = 3 or TYPE = 4, the floating point number addressed is examined. If it is undefined (see Floating Point) the trap UFN will occur. In case the central registers are used, storing is performed as in the floating point store (STF) instruction. (Refer to Floating Point)

For TYPE = 7 and STR(AAW)  $\neq \emptyset$ , the four-word string descriptor is copied. If the BLL being executed is a system call (as described later), four ring checks are done, with P as source and each of the four word addresses as target. Furthermore, the word addresses must be non-decreasing from one word to the next, and the CPOS and CSIZE fields of the first word are copied into the others. Finally, 2 is forced into the top two bits of each word to ensure that it is a string descriptor.

For TYPE = 8 and STR(AAW)  $\neq \emptyset$  a label is copied as follows:

The first word is made absolute, i.e. Q added to the sign-extended SRW becomes the new LW if SREL is set, then SREL is cleared.

In the second word, if bits 6-23 are  $\emptyset$ , the word is replaced by L if STK =  $\emptyset$ , else

NEWL + the STK and UWSTK bits, if STK = 1.

The basic idea is to supply the proper context, so that the current local environment will be restored if the label is branched to. Refer to the discussion of how to unwind the stack to see why NEWL is used when STK = 1.

If the label is passed by a system call, the absolute address in the first word is ring-checked. Before copying the second word CLL, STK and SREL are cleared and bits 6-23 are checked. If they are not  $\emptyset$ , BLLERR(6) occurs.

For STR(AAW) =  $\emptyset$  the type is ignored. An array descriptor is also copied like a two-word scalar, except that the second word W2 is replaced by an X or ROX IAW with address equal to the effective address which results from treating W2 as an IAW. This permits an array descriptor which uses relative addressing to be passed as a parameter, since the relative address is automatically converted to absolute. If BLL is a system call, in addition two ring checks are done with P as source and both the first and last words of the array as targets. This means that if an array descriptor is passed to a higher ring, the higher ring can use it without fear of accessing storage which the calling program could not have accessed.

When the address or value has been copied, ENDF(AAW) is compared with ENDF(NAW). If they differ, BLLERR(2) occurs. If both are  $\emptyset$ , copying continues with

NAW  $\leftarrow$  NAW + 1; NFW  $\leftarrow$  NFW + 1;

otherwise it stops. In the latter case  $NEWP \leftarrow NFW + 1$

3) If the CLL bit is on, a return descriptor is computed and stored at NEWL. It consists of 2 words:  $NAW + 1$

Note that this is the return address

$$(2B7 * STK) + (1B7 * CPR) + L$$

i.e., the old local environment, with STK bit on if it is on in the descriptor, and CPA bit on if CPR bit is on in the call descriptor. Note that if  $STK = 1$ , then  $2B7 * STK$  sets the STK bit (bit 1) of the return descriptor on.

4) Set L to NEWL, P to NEWP, and continue execution. If the FTN bit is set, skip one word unless the FTNAT flag is on. The instruction skipped presumably will contain a subroutine call to take care of the special cases in FORTRAN mentioned earlier.

In order to state precisely and concisely how this instruction works and to describe the details of ring-checking, an SPL program is presented in the Appendix which duplicates its functioning. This program uses some special functions. (Those not mentioned here refer to fields or functions defined elsewhere in this document.)

1) The construction  $\$X\leftarrow$  implies a ring check with R as source and X as target. As the access is a store, the trap PRO may also occur.

2) RINGCHECK(X) performs a ring check with R as source and X as target. If the check fails, trap MACC will occur.

3) RING(X) produces a number depending on the ring which contains X, say

1 if X is in the user ring

2 if X is in the utility ring

3 if X is in the monitor ring

4) MENTER(), MEXIT() and INTERRUPT() designate the places where the actions described under "CPU Interruptability" are taken.

5) EA(X) initiates the effective address calculation similar to IA(X), but the format of CONTENTS(X) is like an instruction (or an AAW) rather than an IAW.

## 8. Programmed Operators

If the POP bit of an instruction is 1, it is interpreted as a rather peculiar kind of subroutine call rather than an ordinary machine instruction. Execution proceeds as follows:

the OPC field of the instruction is put into the indexing register (IR) and the instruction:

```
BLL $G'[Ø];
```

is executed

Presumably word Ø of G will contain the address of a transfer vector. If desired, it may contain an array descriptor which limits the number of programmed operators and supplies a multiplier of 2.

There is one additional feature: BLL will initialize NAW to P, rather than to P + 1, so it will use the instruction word as the first AAW. STR, TYPE and ENDF will be taken from the corresponding bit positions of the first FAW.

## 9. System Calls

Two versions of the OPR instruction provide protected entry points into the system. The MCALL instruction works as follows:

8 bits provided by the OPR are put into IR

a BLL \$BA is executed, with BA = 604000B.

When the BLL is completed,

$G \leftarrow \text{NEWG}$ , where NEWG = 600000B.

The intention is that 604000B should contain an array descriptor with

LB =  $\emptyset$

UB = total number of defined system calls

MULT = 2

which points to an array of BLL descriptors for the various protected entry points. Note on any calls to the system from a lower ring, G is saved in NEWG[14]. G is restored from G'[14] by any BLL (BLLN, POP, etc.) which crosses the ring boundary into a lower ring.

For calls into the utility the UCALL version of OPR works the same way, except that BA = 400016B and NEWG = 400000B. Note that this is the beginning of the utility ring. Variants of these OPRs exist which execute a BLLN instead of a BLL. (MCALN, UCALN)

The PDFLAG, TDFLAG bits in the status register are cleared by both MCALLs and UCALLs.

MCALLs also set the locked bit of the CPU as described under "CPU interruptability".



## 10. Traps

A machine trap is a forced transfer of control which may occur as a result of a variety of untoward events which may arise during the execution of a program. It does not involve a switch to a new process.

A trap may be fixed or ring-dependent. All fixed traps save the first 10 words of the state in the 10 words starting at 602752B. They then set G to 6000000B and do  $X \leftarrow n$ ; BRU 604002B, where n is the trap number. They all have a one word parameter which is put into the A register after the state is stored. The value of the parameter depends on the trap. Like MCALL-s, fixed traps also clear PDFLAG, TDFLAG, 940M and set the LOCKED bit.

A table of all fixed traps is given in the Appendix. Each one is described more fully in its proper place in the manual.

The ring-dependent traps differ in that they send control to a location determined by the ring that P is in. They store P and the parameter at G'[4] and G'[5] respectively and then clear the 940M bit in the status register and do  $IR \leftarrow n$ ; BRU \$G'[6].

In 940 mode, if the S bit (bit 0) of an instruction and the P bit (bit 2) are set, the instruction is called a SYSPOP. The first 10 words of the state are stored starting at L[3], then A is set to the effective address of the instruction, clear 940M and do  $X \leftarrow OPC$ , BRU L'[2].

## 11. CPU Interruptability

The CPU described in this manual is expected to run as part of a system which includes, among other things,

1) Two physical CPUs, which are identical except for a number called the CPU number attached to each CPU. The CPUs are numbered from 0 to 1.

2) a separate processor called the μscheduler which is responsible for allocating CPUs to processes. The μscheduler also has facilities for causing the CPU to operate in a single-step mode, in which it stores the state, waits and then reloads it after each instruction execution, and for telling the CPU to stop execution at once (crash).

3) A protect mechanism which allows the various processors in the system to be interlocked or synchronized. There are eight protect lines, any of which may be seized by any processor. A line may be seized by only one processor at a time; anyone else attempting to seize the line is refused until the current owner lets it go.

This section describes the behavior of the CPU with respect to

- 1) A STROBE signal, which the μscheduler sends when the CPU is to switch processes
- 2) The single-step and crash signals
- 3) Protect 4, which is used to interlock the CPUs, keeping more than one from being in a locked state.
- 4) The timer trap, which occurs when the interval timer in the state becomes negative

- 5) The XMON and XUTIL traps
- 6) Initialization

The relevant information is:

- a) Some information in the state
  - 1) The ring in which the P-counter is contained
  - 2) The XMON trap bit in SR
  - 3) The XUTIL trap bit in SR
  - 4) The sign bit of the interval timer, which we call TO
- b) Some flip-flops in the microprocessor which are not part of the CPU state
  - 1) STROBE, which may be set by another microprocessor, normally the  $\mu$ scheduler
  - 2) STEP, which may be set by some external device to make the CPU operate in a single-step mode
  - 3) LOCKED, which is not accessible to external devices
  - 4) ALARM, which is set when a system crash is impending
- c) The state of Protect 4, which will be called CPUPRO
- d) A location in absolute core called CPUWAIT which is used to keep the CPU idle after the system has crashed or between STEPs.

### A. Idle State

When it is initialized (by setting the 0 register in the micro-processor to  $\emptyset$ ) the CPU goes into idle state.

IDLE: Clear map scan request;  
 GOTO IDLE IF NOT STROBE;  
 Clear STROBE.

PWAIT: T ← contents of absolute call (6 + CPU number)

(T is the process' PRT index)

Goto PWAIT if T =  $\emptyset$ ;

Clear absolute cell (6 + CPU number);

Clear LOCKED; Clear the map;

Find the page with the name in (T) and (T+1)

Take it as a context block and load the state from location 2764B-2777B in it (called the SAVE area).

If the page is not found in CHT, send a STROBE2 to the  $\mu$ scheduler with a message 4B7 in absolute cell 2454B + CPU number \*4 then do like ABORT.

Start executing instructions at the location given by the P-counter;

The CPU returns to the idle state whenever it dumps the state of a process.

### B. Interruption of Program Execution

At the start of every instruction, the truth of any of the following conditions will stop execution and cause the indicated

action to be taken. The conditions are treated in the order in which they are listed.

- 1) NOT LOCKED AND TO: cause timer overflow trap.
- 2) NOT LOCKED AND STROBE: dump the state into the SAVE area, send a RETURN message to the  $\mu$ scheduler and go into idle state.
- 3) STEP OR ALARM: dump the state into the SAVE area, clear STEP. Clear the wait location (23B + CPU number) and wait until it becomes 12343210B, then reload the state from the SAVE area and proceed.

At every step of indirection, every start of an instruction which is the target of EXU, every parameter of a BLL and in all other places where the CPU might be held up for more than a few microseconds, (MVB, MVS, CPS), conditions 1 and 2 are tested and their indicated actions taken.

### C. Setting the Bits

XMONT and XUTILT are part of SR and may be set or cleared with SRS, LOADS or XSA.

LOCKED is set by MCALL or fixed trap. It can also be set by SLOK. It is cleared by any BLL or LOADS which leaves the monitor ring (BLL, here, includes all variants: UCALL, MCALL, POP), and can also be cleared by RLOK.

TO can be changed by loading a state from the SAVE area or by the OPR to set the interval timer.

#### D. The X Traps

At every BLL or LOADS a check is made for transition into a lower ring. If there is a transition from monitor to utility or user rings, the XMON trap is caused if the XMONT bit is set. Then if there is a transition from utility to user ring, the XUTIL trap is caused if the XUTILT bit is set.

#### E. The CPUPRO Signal

This protect is seized automatically at each point where LOCKED is set and cleared at each point where LOCKED is cleared. The programmer can set it himself with the PRO operate, but this is probably unwise.

## 12. Ordinary Instructions

This section contains a complete description of the behavior of the machine when interpreting an instruction word, with the following exceptions:

instructions with POP = 1 are described under "Programmed Operators"

the BLL instruction is described under "Function Calls"

the floating point instructions are treated in a separate section

effective address computation for all instructions is described under "Addressing"

Each instruction is specified in terms of its operands, its effect on the state of memory of the running process, and any unusual traps it may cause. Traps which are caused by the addressing system are the same for all instructions and are not considered. Traps caused by the map are the same except for the read-only trap. Its occurrence depends on whether the instruction attempts to modify memory; this should be obvious from the instruction description and will not be further mentioned. The address type is S or D for instructions which modify memory.

Part of the state is a 2-bit condition-code. This code is set by the RESULT of most instructions as follows:

0 if RESULT < 0

1 if RESULT = 0

2 if RESULT > 0

The RESULT is indicated in the description of each instruction. Unless some other change in P is indicated, all instructions end with

$$P \leftarrow P + 1$$

The INSTD bit in the status register is set to  $\emptyset$  at the end of every instruction, except for LOADS.

The address type of the instruction is indicated for every instruction, e.g.,

LDA (F)

In the description some special notation is used: STORE(X,Y) stores X in the memory location addressed by Y. The storing includes some special logic for (S) type instructions if a field or character is specified as operand (refer to Use of Addresses by Instructions); ABS(T) is the absolute value of T.

ABS(4B7) = 4B7.



Summary of Abbreviations

|                 |  |  |
|-----------------|--|--|
| AR              | A register   |  |
| BR              | B register   |  |
| CR              | C register   | } (used only for double-precision floating-point and quadruple loads and stores) |
| DR              | D register   |  |
| XR              | X register   |  |
| P               | Program counter  |  |
| L               | Local environment register   |  |
| G               | Global environment register  |  |
| CC              | Condition Code, equivalent to RESULT:  |  |
|                 | CC = 0   | RESULT < 0   |
|                 | CC = 1   | RESULT = 0   |
|                 | CC = 2   | RESULT > 0   |
| SR              | Status register  |  |
| OV = SR[22]     | Overflow bit   |  |
| TOV = SR[21]    | Temporary Overflow bit   |  |
| CARRY = SR[20]  | Carry bit  |  |
| PDFLAG = SR[19] | Permanent double-precision flag. Used to set TDFLAG after STF, STD or FCP                |  |
| TDFLAG = SR[18] | Temporary double-precision flag. Makes all floating-point instructions double-precision. |  |

## A. Data Transfer Instructions (12)

LDA (F) Load A register

AR  $\leftarrow$  OP;RESULT  $\leftarrow$  AR;

LDB (F) Load B-register

BR  $\leftarrow$  OP;RESULT  $\leftarrow$  BR;

LDX (F) Load X-register

XR  $\leftarrow$  OP;

CC is unchanged

LDD (F) Load double

AR  $\leftarrow$  CONTENTS(Q); BR  $\leftarrow$  CONTENTS(Q+1);CR  $\leftarrow$  CONTENTS(Q+2) & DR  $\leftarrow$  CONTENTS(Q+3) IF TDFLAG=1;RESULT  $\leftarrow$  AR;

EAX (E) Effective address to X

XR  $\leftarrow$  Q;

CC is unchanged

LAX (F) Load array index

XR  $\leftarrow$  Q OR 4B6; (sets TAG to 2 for indirection)

CC is unchanged

Treats bit ATRAP in an array descriptor opposite to all other instructions

LN<sub>X</sub> (F) Load negative to X

$XR \leftarrow -OP$ ; two's complement negation

CC is unchanged

ST<sub>A</sub> (S) Store A register

STORE(AR,Q);

CC is unchanged

ST<sub>B</sub> (S) Store B register

STORE(BR,Q);

CC is unchanged

ST<sub>X</sub> (S) Store X register

STORE(XR,Q);

CC is unchanged

ST<sub>D</sub> (D) Store double

STORE(AR,Q); STORE(BR,Q+1);

STORE(CR,Q+2) & STORE(DR,Q+3) & TDFLAG  $\leftarrow$  PDFLAG

IF TDFLAG = 1;

CC is unchanged

XMA (S) Exchange memory and A

TEMP  $\leftarrow$  AR; AR  $\leftarrow$  OP; STORE(TEMP,Q);

RESULT  $\leftarrow$  AR;

## B. Integer Arithmetic Instructions (10)

ADD (F) Add memory to A

AR  $\leftarrow$  AR + OP; (two's complement)

CARRY  $\leftarrow$  carry from bit 0 of adder, i.e., set if the sum of AR and OP taken as unsigned 24-bit integers, is  $\geq 2^{24}$ , and cleared otherwise;

TOV  $\leftarrow$  1 if the add causes overflow, i.e., if AR and OP have the same sign but the sum has a different sign, else 0;

OV  $\leftarrow$  OV or TOV;

RESULT  $\leftarrow$  AR;

SUB (F) Subtract memory from A

Proceed exactly like ADD except that (-OP) replaces OP.

This is a two's complement negate, i.e., (NOT OP + 1)

ADC (F) Add memory and CARRY to A

OV  $\leftarrow$  0

AR  $\leftarrow$  AR + OP + CARRY;

Then proceed exactly like ADD

SUC (F) Subtract memory from A + CARRY

OV  $\leftarrow$  0;

AR  $\leftarrow$  AR + CARRY + (NOT OP);

Then proceed exactly like ADD

MIN (S) Memory increment

RESULT  $\leftarrow$  AR  $\leftarrow$  OP + 1;

STORE (RESULT, Q);

MDC (S) Memory decrement

RESULT  $\leftarrow$  AR  $\leftarrow$  OP - 1;

STORE (RESULT, Q);

ADM (S) Add to memory

RESULT  $\leftarrow$  AR  $\leftarrow$  OP + AR;

STORE (RESULT, Q);

ADX (F) Add to X

XR  $\leftarrow$  XR + OP;

CC is unchanged

MUL (F) Multiply memory and A

TOV  $\leftarrow$  0;

TOV  $\leftarrow$  OV  $\leftarrow$  1 IF OP = AR = 40000000B;

PROD  $\leftarrow$  AR \* OP; as two's complement numbers, yielding a 47-bit two's complement result

AR[0,23]  $\leftarrow$  PROD[0,23];

BR[0,22]  $\leftarrow$  PROD[24,46];

BR[23]  $\leftarrow$  0;

RESULT  $\leftarrow$  (AR OR (BR RSH 1));

The product, consisting of a sign bit and 46 magnitude bits, is left-justified in the AB registers. If integers are being multiplied, an ASHD -1 is required to obtain the integer product in B.

DIV (F) Divide memory into AB

TEMP ← OP; TOV ← ∅;

DIVIDEND ← AB[∅, 46];

QUOTIENT ← DIVIDEND/TEMP;      a 47 bit two's complement  
integer treating both o-  
perands as fractions in  
the range  $-1 \leq f < 1$ , and  
obtaining a quotient with  
23 fraction bits

TOV ← OV ← 1 and proceed to next instruction  
unless  $-1 \leq \text{QUOTIENT} < 1$

AR ← QUOTIENT;

TEMP ← QUOTIENT \* TEMP;      yielding a 47-bit product  
as for multiply

BR ← (DIVIDEND - TEMP);      this is the remainder

RESULT ← AR;

The quotient of the 47-bit dividend and the 24-bit divisor, both taken as signed two's complement fractions, is put into A and the remainder into B. Overflow occurs if the dividend is larger than the divisor, since the quotient cannot be represented as a fraction; in this case, the central registers are unaltered.

To divide an integer in A by one in memory, do ASHD -23 first.

## C. Test Instructions (5)

ICP (F) Integer compare

RESULT ← AR - OP;

CPZ (F) Compare with zero

RESULT ← OP;

CMZ (F) Compare A and memory with zero

RESULT ← AR AND OP;

The following two instructions operate on string descriptors, which are pairs of indirect address words of type string. The intended interpretation is that the first points to the first character of the string, the second to the last character.

ISD (S) Increment string descriptor

TEMP ← CONTENTS(Q);

CSIZE ← TEMP[2,3]; CPOS ← TEMP[4,5];

RESULT ← TEMP - CONTENTS(Q + 1);

Proceed to next instruction if RESULT = 0;

IF CPOS + CSIZE &lt; 3 DO;

CPOS ← CPOS+1;

ELSE DO;

CPOS ← 0; TEMP ← TEMP + 1;

ENDIF;

TEMP[2,3] ← CSIZE; TEMP[4,5] ← CPOS;

STORE(TEMP,Q);

If the string is empty (the two IAWs are equal) the instruction sets CC to 1 and exits. Otherwise it sets CC to  $\emptyset$  or 2, and increments the first IAW by one character position in the string.

DSD (S) Decrement string descriptor

```

TEMP ← CONTENTS(Q+1);
CSIZE ← TEMP[ 2,3] ; CPOS ← TEMP[ 4,5] ;
RESULT ← TEMP - CONTENTS(Q);
Proceed to next instruction if RESULT =  $\emptyset$ ;
IF CPOS >  $\emptyset$  DO;
    CPOS ← CPOS -1;
ELSE DO;
    CPOS ← 3-CSIZE; TEMP ← TEMP -1;
ENDIF;
TEMP[2,3] ← CSIZE; TEMP[4,5] ← CPOS;
STORE(TEMP,Q+1);

```

The idea is the same for ISD, but the second IAW is decremented by one character position.

D. Logical Instructions (3)

ETR (F) And A and memory

```
AR ← AR AND OP;
```

```
RESULT ← AR;
```

IOR (F) Or A and memory

```
AR ← AR OR OP;
```

```
RESULT ← AR;
```



EOR (F) Exclusive or A and memory

AR  $\leftarrow$  AR EOR OP;

RESULT  $\leftarrow$  AR;

#### E. Shift Instructions (6)

All shift instructions interpret the absolute value of OP MOD 64 as the number of shifts to be done. The sign of OP specifies the direction: positive for left shifts, negative for right.

SHIFTC  $\leftarrow$  ABS(OP MOD 64);

right shift as specified IF OP  $<$  0 ELSE

left shift as specified;

RESULT  $\leftarrow$  AR;

ASHD (F) Arithmetic shift double (A and B registers)

A and B taken as a single 48-bit register are shifted. On a right shift, the original sign bit is copied into vacated bit positions. On a left shift, OV  $\leftarrow$  1 if any of the bits shifted out differ from the final sign of A. TOV is set to 1 when OV is set, otherwise it is set to 0.

ASHA (F) Arithmetic shift A

Identical to ASHD except that only AR is shifted

LSHD (F) Logical shift double

A and B taken as a single 48-bit register are shifted. Vacated bit positions are filled with zeros.

LSHA (F) Logical shift A

Identical to LSHD except that only AR is shifted

## CYD (F) Cycle double

A and B taken as a single 48-bit register are cycled. i.e., they are shifted, but bits which are shifted out one end fill the vacated positions at the other end.

## CYA (F) Cycle A

Identical to CYD except that only AR is cycled.

## F. Branch Instructions

## BRU (E) Branch unconditionally

$P \leftarrow Q;$

CC is unchanged

Six instructions test the condition code

## BLT (E) Branch on less than

$P \leftarrow Q$  IF  $CC = 0;$  (RESULT  $< 0$ )

CC is unchanged

## BLE (E) Branch on less than or equal

$P \leftarrow Q$  IF  $CC = 0$  OR  $CC = 1;$  (RESULT  $\leq 0$ )

CC is unchanged

## BEQ (E) Branch on equal

$P \leftarrow Q$  IF  $CC = 1;$  (RESULT = 0)

CC is unchanged

## BNE (E) Branch on not equal

$P \leftarrow Q$  IF  $CC \neq 1;$  (RESULT  $\neq 0$ )

CC is unchanged

BGE (E) Branch on greater than or equal

$P \leftarrow Q$  IF  $CC = 1$  or  $CC = 2$ ; ( $RESULT \geq \emptyset$ )

CC is unchanged

BGT (E) Branch on greater than

$P \leftarrow Q$  IF  $CC = 2$ ; ( $RESULT > \emptyset$ )

CC is unchanged

Two branch instructions affect the X register

BRX (E) Branch on index

$XR \leftarrow XR + 1$ ;

$P \leftarrow Q$  IF  $XR < \emptyset$ ;

CC is unchanged

BSX (E) Branch and set X

$XR \leftarrow P + 1$ ;

$P \leftarrow Q$ ;

CC is unchanged

BLL (S) Branch and load L

is described elsewhere

## G. Miscellaneous Instructions (5)

HLT (F) Halt

Always causes the TI trap

EXU (F) Execute

Initializes  $IR \leftarrow XR$  &  $R \leftarrow Q$ , then interprets  $CONTENTS(Q)$  as an instruction and executes it.

## EAC (E) Effective address computation

This instruction computes the effective address of CONTENTS(Q) interpreted as an instruction word. Similar to EXU, IR and R are initialized to XR and Q respectively. The results of the computation are given in registers as follows:

$XR[\emptyset,5] \leftarrow RESULT \leftarrow 1 \ \& \ AR \leftarrow OP$

if the address is Immediate

$XR[\emptyset,5] \leftarrow RESULT \leftarrow 2 \ \& \ XR[6,23] \leftarrow Q$

if the address is ROD or ROX read only

$XR[\emptyset,5] \leftarrow RESULT \leftarrow 3 \ \& \ XR[6,23] \leftarrow Q \ \& \ AR \leftarrow MASK \ \& \ BR \leftarrow SHIFT$

if the address refers to a field or character

MASK has bits (24-SIZE), 23 on, the rest off.

SHIFT equals to 24-(FB + SIZE)

$XR[\emptyset,5] \leftarrow RESULT \leftarrow \emptyset \ \& \ XR[6,23] \leftarrow Q$

in all other cases

Note that Q - whenever given - is ring checked against R in the final phase of the address calculation. (Refer to "Addressing from Instructions")

## SRS (F) set or reset status bits

The operand is used to set or reset the status register in the state in the following way:

$SR \leftarrow (SR \ OR \ OP \ IF \ (OP \ AND \ 1) = 1 \ ELSE$

$SR \ AND \ NOT \ OP);$

TSB (F) Test status bits

RESULT ← SR AND OP;

i.e., 1 bits in the operand select bits of SR. The condition code is set depending on whether all the selected bits are 0 or not.

H. OPR (F) Operate (1)

If the operand is negative, the instruction is a system call. Bits 14-15 in the absolute value of the operand select one of four alternatives:

0 UCALL

1 UCALN

2 MCALL

3 MCALN

Bits 16-23 in the absolute value is the address for the system call. (as described in a separate section)

If the operand is positive, it is decoded to determine what is to be done:

|     |                              |                                   |
|-----|------------------------------|-----------------------------------|
| CAB | Copy A to B                  | BR ← AR;                          |
| XAB | Exchange A and B             | T ← AR; AR ← RESULT ← BR; BR ← T; |
| CBA | Copy B to A                  | AR ← RESULT ← BR;                 |
| CBX | Copy B to X                  | XR ← BR;                          |
| XXB | Exchange B and X             | T ← BR; BR ← XR; XR ← T;          |
| CXB | Copy X to B                  | BR ← XR;                          |
| CAX | Copy A to X                  | XR ← AR;                          |
| XXA | Exchange X and A             | T ← AR; AR ← RESULT ← XR; XR ← T; |
| CXA | Copy X to A                  | RESULT ← AR ← XR;                 |
| CNA | Negate A                     | AR ← RESULT ← -AR;                |
| CNX | Negate X                     | XR ← -XR;                         |
| ZOA | Clear A                      | AR ← RESULT ← ∅;                  |
| ZAB | Clear AB                     | AR ← BR ← ER ← ∅;                 |
| ZOB | Clear B                      | BR ← ∅;                           |
| CGA | Copy G to A                  | AR ← RESULT ← G;                  |
| XGA | Exchange G and A             | T ← AR; AR ← RESULT ← G; G ← T;   |
| CLA | Copy L to A                  | AR ← RESULT ← L;                  |
| XLA | Exchange L and A             | T ← AR; AR ← RESULT ← L; L ← T;   |
| CSA | Copy SR to A                 | AR ← RESULT ← SR;                 |
| XSA | Exchange SR and A            | T ← AR; AR ← RESULT ← SR; SR ← T; |
| CTA | Copy interval timer to A     | A ← RESULT ← IT;                  |
| CCA | Copy compute time clock to A | A ← RESULT ← CTC;                 |
| NOP | No operation                 |                                   |

**MVB** Move block

The block of AR words starting at XR is moved to the AR words starting at BR. The words are moved one at a time, and the registers are updated after each word is moved to reflect the number of words remaining to be moved. This instruction is interruptable. The move is done in such a way that no word is overwritten until it has been moved.

**MVC** Move constant

XR is stored into the AR words starting at BR. This instruction is interruptable.

**MVS** Move string - (Not presently implemented)

The string of AR bytes starting at the byte specified by BR taken as a string IAW is moved to the AR bytes starting at the byte specified by XR taken as a string IAW. The bytes are moved one at a time, and the registers are updated after each byte is moved to reflect the number of bytes remaining to be moved. If the source and target strings overlap, the move is done in such a way that no character is overwritten until it has been moved. If the strings do not overlap, after execution BR and XR will always point to the first characters after the source and target strings respectively. This instruction is interruptable.

**CPS** Compare string - (Not presently implemented)

The string of AR bytes starting at the byte specified by BR taken as a string IAW is compared with the AR bytes starting at

the byte specified by XR. RESULT is set to indicate whether the first string is smaller, equal to, or greater than the second. The registers are updated every time a byte is compared. This instruction is interruptable.

CLS    Compute length of string

AR and BR are taken as string IAWs. The number of bytes in the string starting at the byte specified by AR and ending at the byte specified by BR, -1 is put into AR. The CSIZE field of BR is used to determine the byte size.

RESULT ← AR;

ASP    Add to string pointer

AR is taken as a string IAW. Into XR is put a string IAW which points to the XRth byte beyond the one pointed to by AR.

LLT    Locate leading transition

The bit number (counting from 0 on the left) of the left-most bit in AB which differs from the sign bit of A is put into XR. If no bits differ, 0 is put into XR.

RESULT ← XR;

COB    Count one bits

The number of one bits in the A and B registers is put into XR.

RESULT ← XR;



## LOADS Load state

Loads the first 10 words of the state (not including the compute time clock or the interval timer) from the 10 words addressed by XR. An MACC trap will occur if the new P is in a higher ring than the current P. This instruction does not clear the INSTD bit. An XMON or XUTIL trap may occur if the new P is in a lower ring than the current P and the XMONT or XUTILT bits are set in the current SR as described under "CPU Interruptability".

## STORS Store state

Stores the first 10 words of the state into the 10 words addressed by XR, but does not store P and XR; the corresponding locations are left unchanged.

## LSC Load string constant

The word addressed by XR is fetched and used to form a 4-word string constant in A, B, C, and D as follows:

```
TEMP ← CONTENTS(XR);
CSIZE ← TEMP[2,3]; CPOS ← TEMP[4,5];
AR ← BR ← 4B7 + CSIZE * 4B6 +
      (3 - CSIZE) * 1B6 + XR;
CR ← DR ← 4B7 + CSIZE * 4B6 +
      CPOS * 1B6 + XR + TEMP[6,23];
```

This means that the XR points to a word used to generate a string descriptor and that this word is immediately followed by the string constant specified.

The following OPRs are privileged. If  $P < 6000000$ , the TI trap will occur.

SLOK Set CPU lock  
 RLOK Reset CPU lock  
 ALD Absolute load A

Loads AR with the contents of the core location whose absolute address (i.e., unmapped address) is contained in XR.

AST Absolute store A

Stores AR into the core location whose absolute address is contained in XR.

AAX Absolute address to X

Loads XR with the absolute address corresponding to the virtual address in XR. Bit 0 is set if the physical map entry was empty. Bit 3 is set if PMRO was on in the physical map entry, bit 2 is set if bit 3 is set or the dirty bit was clear.

PRO Protect

Attempts to set  $PRO_i$  if  $AR[24-i]$  is on. If all the selected PROs are set successfully  $CC \leftarrow 0$ ; else  $CC \leftarrow 1$ .

UNPRO Unprotect

Clears  $PRO_i$  if  $AR[24-i]$  is set.

ATTN Attention

Sends a STROBE signal to microprocessor  $i$ , if  $AR[24-i]$  is set.

USCL     $\mu$ scheduler call

This OPR initiates a switch-processes sequence. The state of the machine is dumped at the SAVE area (602764B). The interval timer, shifted 7 to the right so that the least significant bit counts milliseconds, is stored into the MCT field (8:0,7) of the process' PRT entry.

The  $\mu$ scheduler is called with AR[0,5] as an opcode, the CPU is put into the IDLE state.

## CMAP    Clear map

Sets all EF empty flags in the map to 1.

## CMAPS    Clear maps

Clears the maps of both CPUs in the system.

## CAT    Copy A to timer

Copy A to interval timer IT  $\leftarrow$  AR;

## CAC    Copy A to clock

Copy A to compute time clock CTC  $\leftarrow$  AR;

## RUN    Read unique name

A unique name is read from the unique name generator and put into AB.

BR  $\leftarrow$  low order bits of unique name;

AR  $\leftarrow$  high order bits of unique name;

**LDMP** Load physical map

An entry is loaded into the physical map by placing a 7-bit page number into bits 6-12 of the XR and the entry in bits 13-23 as follows:

```
TEMP ← CONTENTS(XR);
```

```
MAP'REG ← TEMP[6,12];
```

```
MAP'REG[0,0] ← EF ← TEMP[13,13];
```

```
MAP'REG[1,1] ← DB ← TEMP[14,14];
```

```
MAP'REG[2,2] ← PMRO ← TEMP[15,15];
```

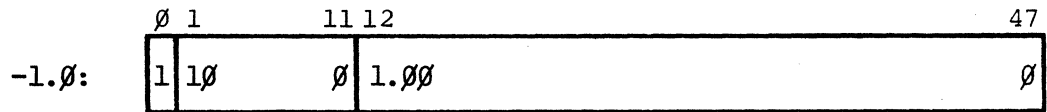
```
MAP'REG[3,10] ← PHYSICAL'PAGE'NO ← TEMP[16,23];
```



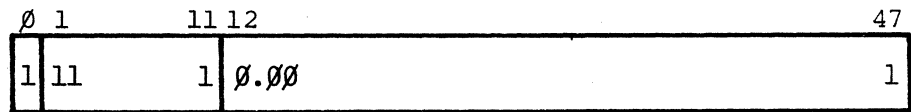
2) Negative numbers

The sign bit (bit 0) indicates that the number is negative. N is given in two's complement form:

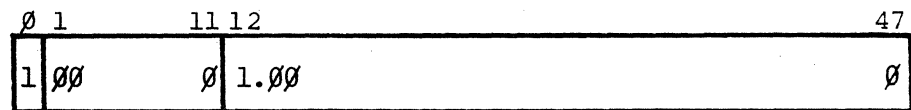
$$\text{negative number } X = (N - 2^{36}) * 2^{(E-35)}, \quad 1 \leq N \leq 2^{35}$$



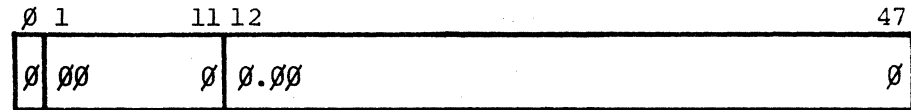
Lowest negative number is  $-2^{2^{1\emptyset}} * (1 - 2^{-36})$



Maximum negative number is  $-2^{-2^{1\emptyset}}$

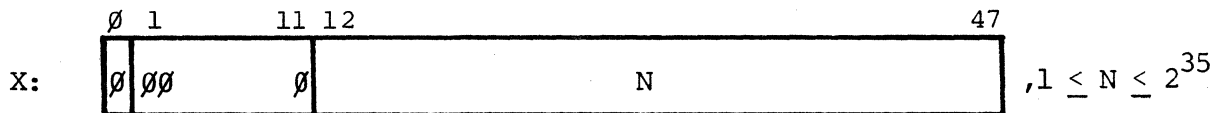


3) Zero



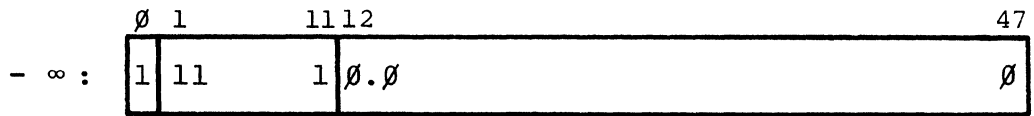
4) Un-normalized numbers

The only un-normalized numbers allowed are these:



and their negatives, i.e.,  $|X| \leq 2^{-2^{1\emptyset}}$ . Note that  $\pm 2^{-2^{1\emptyset}}$  are both normalized and un-normalized.

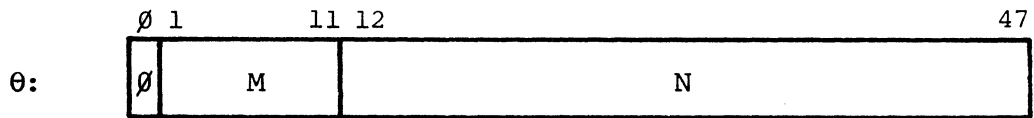
5) Infinity



The symbol  $-\infty$  is treated as the single point at infinity in the one-point (projective) closure of the reals. Operations on  $-\infty$  are summarized in the Appendix.

6) Undefined floating point numbers

Data of the form



with  $\emptyset < M$  &  $\emptyset \leq N \leq 2^{35} - 1$ ,

and their negatives are not floating point numbers. If such a number appears as an operand for any floating point operation, the trap UFN will occur.

B. Algebraic Closure Properties of Normalized Numbers

Numbers of the form A.1, A.2 and A.3 are normalized numbers.

(n.n's)

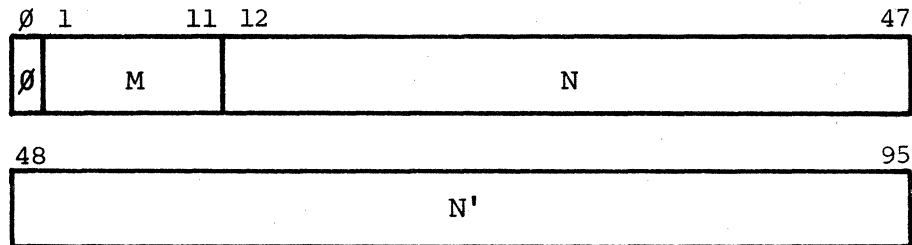
1) If X is an n.n, so is -X.

2) If X is an n.n not zero nor  $\pm 2^{-2^{1\emptyset}}$ , so is  $1.\emptyset/X$ .

The smallest possible n.n whose reciprocal is an n.n is  $2^{-2^{1\emptyset}} (1 + 2^{-35})$ .

### C. Double Precision

The 96-bit double precision data have an additional 48 fraction bits. For example a DP positive number:



represents  $X = (N + N' * 2^{-48}) * 2^{(E-35)}$ ,  $0 \leq N' \leq 2^{48} - 1$ .

### D. Floating Point Instructions (8) and OPRs

All floating point operations have single (SP) and double (DP) precision variants, bit TDFLAG in SR selecting the one to be used. Bit PDFLAG is used to set TDFLAG after a compare (FCP) or store (STF).

Floating operations set CC to indicate if the result is less or greater than or equal to 0. (STF and FIX leave CC unchanged.)

#### FLD (E) Floating load

An SP or DP floating point number starting at Q is copied into the floating point accumulator. (The A, B, C, D, and E central register)

#### STF (D) Floating store

SP: The floating point accumulator is rounded at bit 35 of the fraction and copied to (Q) and (Q+1).



DP: Four words are copied from FA to the locations starting at Q. A double floating store causes no rounding if the FDP bit in SR is set. Otherwise it rounds at bit 71 of the fraction and zeros the last 12 bits. The FDP bit thus determines whether DP numbers are stored with 72 or 84 bits of fraction. Overflow may occur because of the rounding. In all cases TDFLAG ← PDFLAG after the store.

#### FAD (E) Floating add

SP: The operand is extended with 48 zeros on the right. A DP is then done.

DP: Let the operands be  $a * 2^b$ ,  $c * 2^d$ . The two exponents are compared. Suppose  $b \geq d$ . Then  $c$  is shifted right by  $b - d$ . An 87 bit register is provided to hold  $c$ , which is loaded (sign + 84-bit fraction) into the 85 most significant bits. The two least significant bits are cleared. The 86 most significant bits participate in the right shift in the usual way. The least significant bit is 'sticky': if a 1 is ever shifted into it, it remains 1 from then on.

After  $c$  has been shifted, it is added to  $a$  in an 85-bit adder, yielding a result  $r$  of 87 bits. Bits 85:86 of  $c$  do not participate in addition.

Now, if an overflow has occurred ( $a[\emptyset] = c[\emptyset] \neq r[\emptyset]$ ),  $r$  is shifted right by 1.  $r[86]$  is treated as a sticky bit in this shift just as it was in the shift of  $c$ .  $b$  is incremented by 1 if this shift occurs and  $r[\emptyset] \leftarrow \text{NOT } r[\emptyset]$ ;

The result is normalized by left shifting until either:

- 1) the sign bit differs from the next bit or
- 2) the fraction is  $1100 \dots 0$

The exponent  $b$  is decremented by 1 for each left shift.

Lastly the result, rounded at bit 83 of the fraction (i.e.,  $r[84]$ , since when we say 'bit 83 of the fraction' we don't count the sign bit) in accordance with the rounding mode in force, is assigned to the floating point accumulator. See the discussion of rounding below for details. Both overflow and underflow may occur.

#### FSB (E) Floating subtract

Identical to addition except that the negative of the second operand is taken first. This cannot cause any abnormal conditions.

#### FMP (E) Floating multiply

SP: The accumulator is rounded to single precision, then the two 36-bit fractions are multiplied to yield a 72-bit result. The exponent which goes with the result is the sum of the exponents of the operands plus one, to correct for the placement of the binary point in the product. The 72-bit fraction is shifted left if required for normalization. No rounding is required since the accumulator can hold this entire product. Overflow or underflow may occur.

DP: The two 84-bit fractions and the two signs are multiplied to yield an 86-bit result (sign plus 85 magnitude bits) and an

87th bit which is the union of the 82 least significant bits of the full 168-bit product. The resulting 87-bit number and the exponent obtained by the procedure described for single precision are normalized and rounded like the result of an add.

#### FDV (E) Floating Divide

SP: The 36-bit divisor fraction is divided into 38 bits of the accumulator fraction to produce a 37-bit quotient. To this is appended a 38th bit which is set if the division is not exact or if the other 46 bits of the accumulator fraction are non-zero. The resulting 38-bit number is put into the accumulator and filled out with 46 zeros on the right. The exponent of the result is computed by subtracting the divisor exponent from the dividend exponent.

DP: The 84-bit divisor fraction is divided into the 84-bit accumulator fraction to produce an 85-bit quotient. The exponent is computed as for SP and the result is rounded in the usual way.

Overflow or underflow may occur. Division by  $\emptyset$  produces its own trap. (DIZ)

If the divisor is an un-normalized number it is normalized prior to division. It may or may not cause overflow as explained above.

#### FCP (E) Floating compare

Identical to floating subtract, but the result is not assigned to the floating accumulator. CC will be set as usual to indicate the sign of the result. TDFLAG + PDFLAG

**FLX (E) Fix and load X**

XR is assigned a 24-bit integer which is the floor of the floating operand. If the floor is  $> 2^{23} - 1$  in magnitude, the trap FLXO occurs. The result does not depend on SP or DP mode.

**FNA (OPR) Floating negative**

The number in the floating point accumulator is replaced by its negative.

**FIX (OPR)**

Similar to FLX, but the operand is taken from the floating point accumulator and the result is put into RESULT and AR.

**FLOAT (OPR)**

A FLOAT operation produces a (normalized) floating point number in the floating point accumulator which when FIXed will restore the integer operand in AR. (unless it is 4B7) Nothing can go wrong with FLOAT.

**E. Rounding**

There is a three-bit field (TRMOD) in SR which specifies how rounding is to be done (the field PRMOD is used to set TRMOD after every FAD, FSB, FMP, FDV, STF or FCP). The descriptions of instructions above state explicitly each point where rounding is done. The phrase 'round at bit n of the fraction' means that bit n of the fraction (numbering the magnitude bits from 0 and not counting the sign) is the least significant bit retained.

The rounding modes are:

| <u>TRMOD</u> | <u>Name</u> | <u>Rounding</u>       |
|--------------|-------------|-----------------------|
| Ø            | N           | nearest number        |
| 2            | F           | floor (toward Ø)      |
| 3            | C           | ceiling (away from Ø) |
| 4            | P           | away from - ∞         |
| 5            | M           | toward - ∞            |

Rounding involves three bits. The first is the least significant bit to be retained and is called Q. The one following Q is called R. The third is the union of all the bits following R (sometimes only 1, none for double divide) and is called T.

The rounding rules are as follows (call the sign S):

- N: +1 (add 1 to least significant retained bit)  
if R = 1 unless Q = Ø and T = Ø
- F: +1 if S = 1 and R or T = 1
- C: +1 if S = Ø and R or T = 1
- P: + 1 if R or T = 1
- M: no action

#### F. Overflow and Underflow

Overflow and underflow occurs if at the end of a floating point instruction, the exponent is outside the permitted range.

Overflow always causes a trap (FLO). It leaves a correct result except for the exponent, which must be read as a 12-bit two's

complement number with sign bit the complement of the high-order bit preserved.

Underflow action depends on the SUF bit in SR. If it is set, no trap occurs and a suitable un-normalized number of zero results. Otherwise, trap FLU occurs and the result is correct (and normalized) with the same rule for the exponent as was stated for overflow.

## APPENDIX CONTENTS

|   | Page |
|---|------|
| DEFINITION OF INSTRUCTION CODES . . . . .   | 103  |
| DEFINITION OF OPR ADDRESSES . . . . .       | 104  |
| SUMMARY OF ADDRESSING . . . . .             | 105  |
| SUMMARY OF INSTRUCTION ADDRESSING . . . . . | 106  |
| FIXED TRAPS . . . . .                       | 108  |
| RING-DEPENDENT TRAPS . . . . .              | 109  |
| RING-DEPENDENT TRAP IØ: BLLERR . . . . .    | 110  |
| SUMMARY OF IMPORTANT CORE ADDRESS . . . . . | 111  |
| SPL PROGRAM TO DEFINE BLL . . . . .         | 113  |
| WORD FORMATS . . . . .                      | 118  |
| SOME FIELDS IN OCTAL FORM . . . . .         | 127  |
| CHT HASHING ALGORITHM . . . . .             | 128  |

## DEFINITION OF INSTRUCTION CODES

| code | mnemonic | a.type | code | mnemonic | a.type |
|------|----------|--------|------|----------|--------|
| 0    | HLT      | F      | 40   | ASHD     | F*     |
| 1    | LDA      | F*     | 41   | ASHA     | F*     |
| 2    | LDB      | F*     | 42   | LSHD     | F*     |
| 3    | LDX      | F      | 43   | LSHA     | F*     |
| 4    | LDD      | F*     | 44   | CYD      | F*     |
| 5    | EAX      | E      | 45   | CYA      | F*     |
| 6    | LNK      | F      | 46   | TSB      | F*     |
| 7    | XMA      | S*     | 47   | LAX      | F      |
| 10   | ETR      | F*     | 50   | BRU      | E      |
| 11   | IOR      | F*     | 51   | BLT      | E      |
| 12   | EOR      | F*     | 52   | BEQ      | E      |
| 13   | STD      | D      | 53   | BLE      | E      |
| 14   | STF      | D      | 54   | BGT      | E      |
| 15   | STA      | S      | 55   | BNE      | E      |
| 16   | STB      | S      | 56   | BGE      | E      |
| 17   | STX      | S      | 57   | BLL      | S      |
| 20   | ADD      | F*     | 60   | BLLN     | S      |
| 21   | SUB      | F*     | 61   | BRX      | E      |
| 22   | ADC      | F*     | 62   | BSX      | E      |
| 23   | SUC      | F*     | 63   | SRS      | F      |
| 24   | ADM      | S*     | 64   | EAC      | E*     |
| 25   | ADX      | F      | 65   |          |        |
| 26   | MIN      | S*     | 66   |          |        |
| 27   | MDC      | S*     | 67   |          |        |
| 30   | MUL      | F*     | 70   | FLX      | F      |
| 31   | DIV      | F*     | 71   | FLD      | F*     |
| 32   | ICP      | F*     | 72   | FCP      | F*     |
| 33   | CPZ      | F*     | 73   | FAD      | F*     |
| 34   | CMZ      | F*     | 74   | FSB      | F*     |
| 35   | ISD      | S*     | 75   | FMP      | F*     |
| 36   | DSD      | S*     | 76   | FDV      | F*     |
| 37   | EXU      | F?     | 77   | OPR      | F?     |

\* indicates that CC is set by the instruction



## DEFINITION OF OPR ADDRESSES

| OPR address | mnemonic |   | OPR address | mnemonic |   |
|-------------|----------|---|-------------|----------|---|
| 0           | CAB      |   | 40          |          |   |
| 1           | XAB      | * | 41          | LOADS    | * |
| 2           | CBA      | * | 42          | STORS    |   |
| 3           | CBX      |   | 43          | LSC      |   |
| 4           | XXB      |   | 44          | FIX      | * |
| 5           | CXB      |   | 45          | FLOAT    | * |
| 6           | CAX      |   | 46          | FNA      | * |
| 7           | XXA      | * | 47          |          |   |
| 10          | CXA      | * | 50          |          |   |
| 11          | CNA      | * | 51          |          |   |
| 12          | CNX      |   | 52          |          |   |
| 13          | ZOA      | * | 53          |          |   |
| 14          | ZAB      |   | 54          |          |   |
| 15          | ZOB      |   | 55          | SLOK     |   |
| 16          | CGA      | * | 56          | RLOK     |   |
| 17          | XGA      | * | 57          | ALD      | * |
| 20          | CLA      | * | 60          | AST      |   |
| 21          | XLA      | * | 61          | AAX      |   |
| 22          | CSA      | * | 62          | PRO      | * |
| 23          | XSA      | * | 63          |          |   |
| 24          | CTA      | * | 64          | UNPRO    |   |
| 25          | CCA      | * | 65          | ATTN     |   |
| 26          | NOP      |   | 66          | USCL     |   |
| 27          | MVB      |   | 67          | CMAP     |   |
| 30          | MVC      |   | 70          | CMAPS    |   |
| 31          | MVS      |   | 71          | CAT      |   |
| 32          | CPS      | * | 72          | CAC      |   |
| 33          | CLS      | * | 73          | RUN      | * |
| 34          | ASP      |   | 74          |          |   |
| 35          | LLT      | * | 75          | LDMAP    |   |
| 36          | COB      | * | 76          |          |   |
| 37          |          |   | 77          |          |   |

\* indicates that CC is set by the OPR

## SUMMARY OF ADDRESSING

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  $IR ← XR$  &  $R ← P$ ;

## SUMMARY OF INSTRUCTION ADDRESSING

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

## SUMMARY OF INSTRUCTION ADDRESSING (continued)

| <u>Abbr</u> | <u>Name</u>         | <u>Notation</u>   | <u>Address Computation</u>   |
|-------------|---------------------|---|--|
| LR          | L-Relative          | OPC L'[D];  | DISP ← W[13,23];<br>Q ← L + DISP;<br>OP ← CONTENTS(Q);   |
| LRI         | L-Relative-Indirect | OPC \$L'[D];  | DISP ← W[13,23];<br>Q ← L + DISP;<br>IA(Q);  |
|             | Field               | SE(2), SIZE(3,7),<br>FB(8,12), DISP(13,23)                                    | Q ← IR + DISP;<br>U ← CONTENTS(Q);<br>OP ← U[FB,FB+SIZE-1];<br>OP ← OP - 2**(24-FB) IF SE = 1 AND OP[FB,FB] = 1;   |
|             | String              | Csize(2,3), CPOS(4,5),<br>WA(6,23)  | Select byte CPOS of Csize from word WA of<br>string.   |
|             | Array               | LB(2), ATRAP(3), LEB(4),<br>MULTS(5,6), MULTL(5,10),<br>UBS(7,23), UBL(11,23) | TRAP'ABE(R) IF IR < LB;<br>IATRP(R) IF (ATRAP=1) AND (INSTR≠LAX);<br>IATRP(R) IF (ATRAP=0) AND (INSTR=LAX);<br>IF LEB = 0 DO;<br>TRAP'ABE(R) IF IR > UBS;<br>IR ← (IR-LB) * (MULTS+1);<br>ELSE DO;<br>TRAP'ABE(R) IF IR > UBL;<br>IR ← (IR-LB) * (MULTL+1);<br>ENDIF;<br>T ← R + 1;<br>NORMAL'IA(T); |

## FIXED TRAPS

| <u>Number</u> | <u>Name</u> | <u>Caused by</u>   | <u>Parameter</u>                 |
|---------------|-------------|--|----------------------------------|
| 1             | MACC        | Memory access error - attempted access to monitor from below M or utility from below U | $Q + (\text{RING}(R) - 1) * 1B6$ |
| 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  | ---                              |

## RING-DEPENDENT TRAPS

| <u>Number</u> | <u>Name</u> | <u>Caused by</u>   | <u>Parameter</u>             |
|---------------|-------------|--|------------------------------|
| 1             | ABE         | array bound exceeded   | address of IAW               |
| 2             | FLO         | floating overflow  | ---                          |
| 3             | FLU         | floating underflow   | ---                          |
| 4             | RO          | read only trap   | address of ROD<br>or ROX IAW |
| 5             | IATRP       | indirect address trap bit  | address of IAW               |
| 6             | UFN         | undefined floating number  | ---                          |
| 7             | FLXO        | overflow on FIX or FLX<br>instruction                                  | ---                          |
| 8             | DIZ         | floating divide by zero  | ---                          |
| 9             | STKOV       | stack overflow   | ---                          |
| 10            | BLLERR      | function call error<br>described in separate table<br>on the next page | NAW+CLASS*1B6                |

## RING-DEPENDENT TRAP IØ: BLLERR

| <u>Class</u> |   | <u>Parameter</u> |
|--------------|---|------------------|
| 1            | address type error in A                           | 1B6              |
| 2            | wrong number of arguments                         | 2B6 + NAW        |
| 3            | argument type mismatch                            | 3B6 + NAW        |
| 4            | inadmissible argument                             | 4B6 + NAW        |
| 5            | address type error                                | 5B6 + NAW        |
| 6            | array, label or string<br>descriptor format error | 6B6 + NAW        |

## SUMMARY OF IMPORTANT CORE ADDRESS

|            |  |
|------------|--|
| ∅          | Start at the user ring   |
| G'[∅]      | POP entry IAW  |
| G'[1]      | 2nd word of POP entry IAW  |
| G'[2]      | SP - Stack Pointer   |
| G'[3]      | SL - Stack Limit   |
| G'[4]      | Ring dependent trap - P is stored here                                   |
| G'[5]      | Ring dependent trap - parameter is stored here                           |
| G'[6]      | Ring dependent trap service entry IAW                                    |
| G'[7]      | (may be used as 2nd word of IAW)   |
| G'[31]     | Last word which can be used as an index in BX                            |
| G'[127]    | Last word which can be used as a pointer in PD or IPD or as a base in BX |
| G'[37777B] | Last word which can be accessed by D, I addressing                       |
| L'[∅]      | 1st word of the return descriptor - P                                    |
| L'[1]      | 2nd word of the return descriptor - L, STK, CPA                          |
| L'[2]      | SYSPOP transfer address  |
| L'[31]     | Similar to G'[31]  |
| L'[127]    | Similar to G'[127]   |
| L'[2∅47]   | Last word which can be addressed by L, LI addressing                     |



## SUMMARY OF IMPORTANT CORE ADDRESS (continued)

|          |   |
|----------|---|
| 4000000B | Start of utility ring, G for utility                |
| 400014B  | G may be stored here                                |
| 400016B  | UCALL entry IAW                                     |
| 6000000B | Start of monitor ring, G for monitor, context block |
| 600014B  | G may be stored here                                |
| 602752B  | State is stored here if a fixed trap occurs         |
| 602764B  | Start of the SAVE area                              |
| 6040000B | MCALL entry IAW                                     |
| 604002B  | Fixed trap entry                                    |
| 777777B  | Maximum virtual address                             |

## SPL PROGRAM TO DEFINE BLL

\* SPL PROGRAM TO DEFINE BLL

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

\* OPR WITH NEGATIVE OPERAND:

OPR: OP← -OP;  
 N←OP \$ BIT15; SPEC←∅;  
 MCAL←OP \$ BIT14+1;  
 (NEWG←4∅∅∅∅∅B & R←4∅∅∅14B) IF MCAL=1 ELSE  
 (NEWG←6∅∅∅∅∅B & R←6∅4∅∅∅B);  
 IR←OP \$ BIT16THRU23; IA(R); GOTO BLL1;

\*  
 POP: POPW←CONTENTS(P); IR←POPW \$ FOPC; N←∅  
 SPEC←1; MCAL←∅; 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=∅  
 ELSE Q+NEWPW \$ FSRW);  
 BRD←CONTENTS(Q+1) FTNATF←∅;  
 CLL←BRD \$ BIT∅; 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=∅ DO;  
 IF UWSTK=∅; 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=∅;  
 ENDIF;

\*  
 RINGCHECK(NEWP);



```

* 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;
      NAW←NAW+1;

```

```

L3:          ENDIF;
            INTERRUPT'CHECK();
            GOTO L4 IF FP $ ENDF=1;
            ENDFOR;
L4:          NEWP←NEW+1;
            ENDIF;
*
* COMPUTE RETURN DESCRIPTOR
            IF CLL=1 DO;
                R←NEWP;
                NEWL[0]←NAW;
                NEWL[1]←L+2B7*STK+1B7*CPR;
                NEWG[14B]←G IF MCAL>0 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←0 IF MCAL>0;
            L←NEWL; G←NEWG; OLDP←P; P←NEWP;
            IF RET=1 DO;
MEXIT:      IF OLDP>=6B5 DO;
                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=0;
*
* EXIT FROM BLL
            GOTO NEXT'INSTRUCTION;
*
STRING:     COUNT←4; GOTO L2 IF MCAL=0
            FORM←TEMP AND 14B6 OR 4B7; OLDT←0;
            FOR I←0 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;

```

```

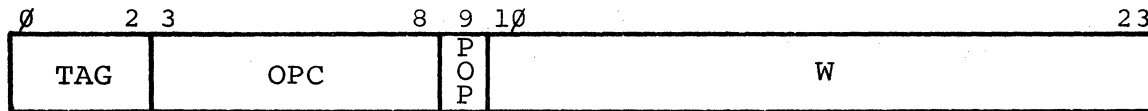
*
LABEL: Q←(TEMP $ FLW IF TEMP $ BIT4=Ø
      ELSE ARGADR+TEMP $ FSRW);
RINGCHECK(Q) IF MCAL>Ø;
R←NEWP;
STORE(CPYADR, Q AND NOT 75B6 OR TEMP AND 75B6);
R←OLDR; BRD←CONTENTS(ARGADR+1);
IF BRD $ FE=Ø AND BRD $ FSTK=Ø DO;
  BRD←BRD AND NOT 4B7 IF MCAL>Ø;
  BRD←BRD OR (L IF STK=Ø ELSE NEWL+2B7+4B6);
ELSE DO;
  BLLERR(6) IF MCAL>Ø;
ENDIF;
R←NEWP;
STORE(COPYADR+1,BRD); GOTO L2;

*
ARRAY: R←NEWP; $CPYADR←TEMP;
BLLERR(6) IF TEMP $ IAT#3;
IF MCAL>Ø DO;
  IF←(TEMP $ UB1 IF TEMP $ LEB=Ø ELSE TEMP $ UB2);
  IA(ARGADR+1); RINGCHECK(Q);
ENDIF;
IR←Ø; R←ARGADR; IA(ARGADR+1);
BLLERR(6) IF IMMEDIATE=1;
RINGCHECK(Q) IF MCAL>Ø;
R←NEWP;
$(CPYADR+1)←(Q+(4B6 IF READONLY=Ø ELSE 12B6));
GOTO L3;

```

## WORD FORMATS

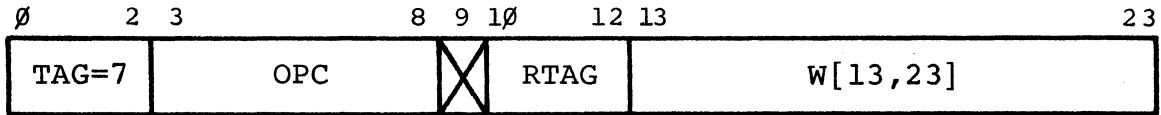
## A. Instruction Word



| <u>Bit</u> | <u>Name</u> | <u>Normal Mode</u>            |
|------------|-------------|-------------------------------|
| 0-2        | TAG         | Addressing mode for TAG field |
| 3-8        | OPC         | Opcode                        |
| 9          | POP         | Pop bit                       |
| 10-23      | W           | Address field                 |

| <u>TAG</u> | <u>Name</u> | <u>Addressing Mode</u>             |
|------------|-------------|------------------------------------|
| 0          | D           | Direct                             |
| 1          | I           | Indirect                           |
| 2          | X           | Indexed                            |
| 3          | BX          | Base-index                         |
| 4          | PD          | Pointer-displacement               |
| 5          | PDI         | Indirect-pointer-displacement      |
| 6          | BXD         | Base-index-displacement            |
| 7          | REL         | Relative. This one has 6 sub-cases |

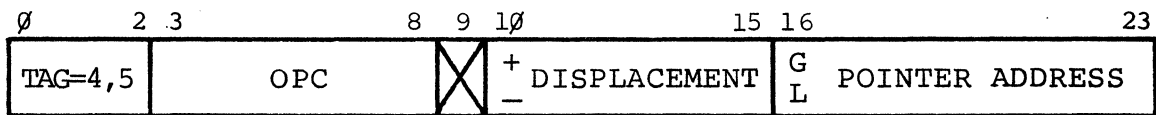
## B. Relative Addressing



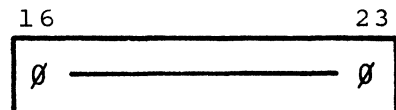
| <u>RTAG</u> | <u>Name</u> | <u>Addressing Mode</u>    |
|-------------|-------------|---------------------------|
| $\emptyset$ | LR          | L-relative                |
| 1           | LRI         | L-relative indirect       |
| 2           | SR          | Source relative*          |
| 4           | SRI         | Source relative indirect* |
| 6           | IMX         | Immediate indexed         |
| 7           | IM          | Immediate                 |

\* these modes use bit 12 in the address field (e.g. W[12,23])

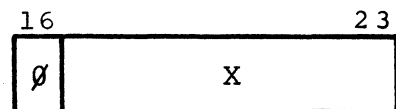
## C. PD, PDI Addressing



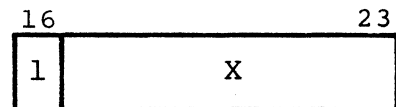
POINTER = IR



POINTER = CONTENTS (G + X)



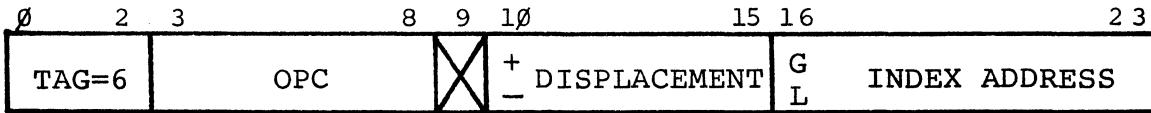
POINTER = CONTENTS (L + X)



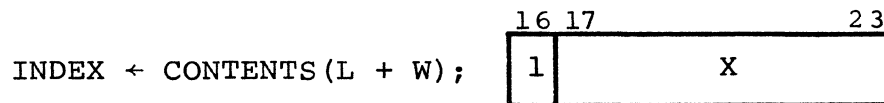
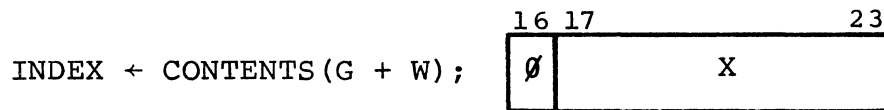




E. BXD Addressing (Base in XR)

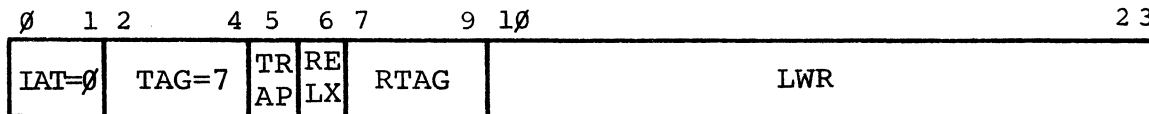


Where the index address field is one of the following:



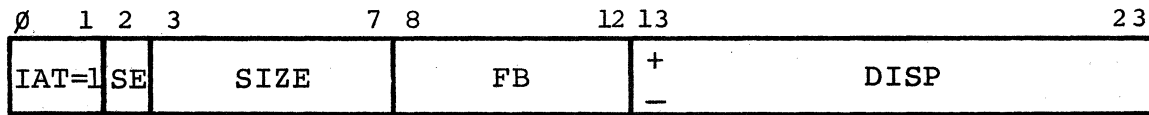
and the base address is in the indexing register.

F. Normal Indirect Address Word (IAW)



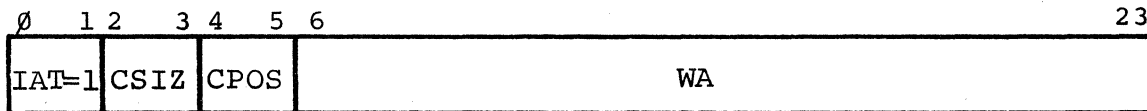
| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                             |
|-------------|-------------|---|
| 0-1         | IAT         | 0   |
| 2-4         | TAG         | interpreted exactly like an instruction TAG |
| 5           | TRAP        | causes trap IATR if set                     |
| 6           | RELX        | causes indexing for the relative modes      |
| 7-23        | LWR         | long address for the relative modes         |
| 6-23        | LW          | long word address                           |
| 10-23       | W           | word address                                |

## G. Field IAW



| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                           |
|-------------|-------------|---|
| 0-1         | IAT         | 1   |
| 3-7         | SIZE        | size of field in bits                     |
| 8-12        | FB          | address of first bit of the field         |
| 2           | SE          | causes sign extension of the field if set |
| 13-23       | DISP        | 2's complement signed displacement        |

## H. String IAW

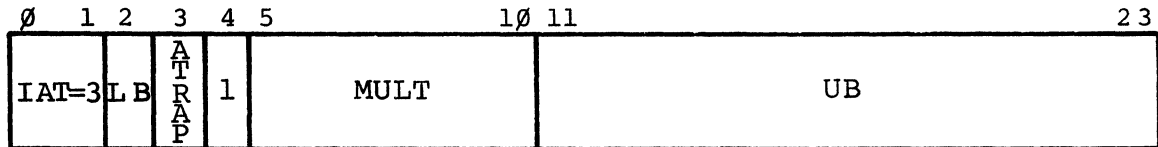
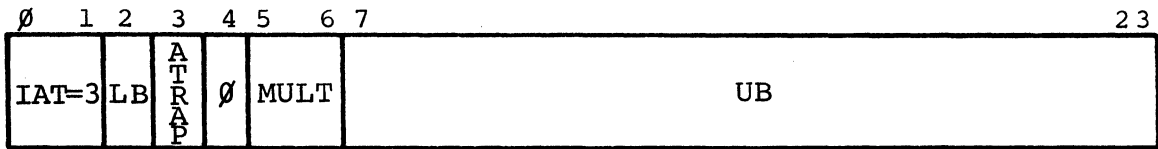


| <u>Bits</u> | <u>Name</u> | <u>Contents</u>                                   |
|-------------|-------------|---|
| 0-1         | IAT         | 2   |
| 2-3         | CSIZE       | character sign: 0 = 6 bits, 1 = 8, 2 = 12; 3 = 24 |
| 4-5         | CPOS        | character position in word                        |
| 6-23        | WA          | word address                                      |

Bits assigned by CSIZE and CPOS:

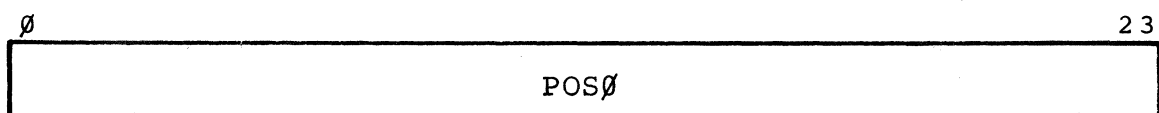
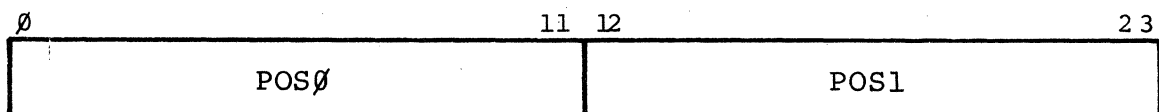
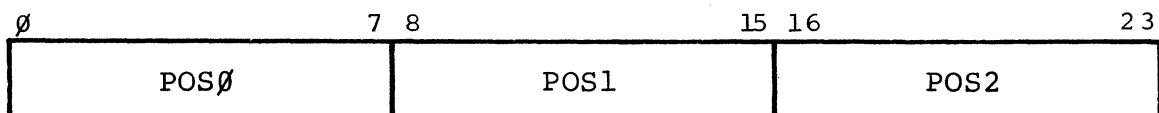
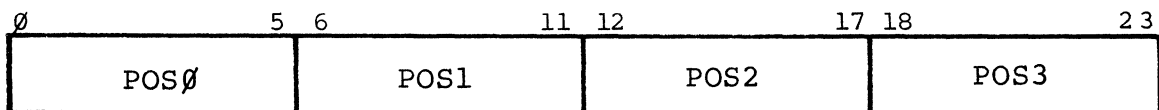
| <u>CSIZE/CPOS</u> | <u>0</u> | <u>1</u> | <u>2</u> | <u>3</u> |
|-------------------|----------|----------|----------|----------|
| 0                 | 0-5      | 6-11     | 12-17    | 18-23    |
| 1                 | 0-7      | 8-15     | 16-23    | X        |
| 2                 | 0-11     | 12-23    | X        | X        |
| 3                 | 0-23     | X        | X        | X        |

I. Array IAW

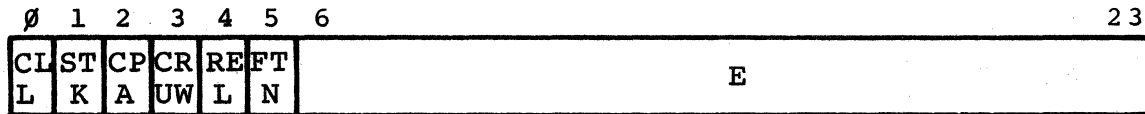
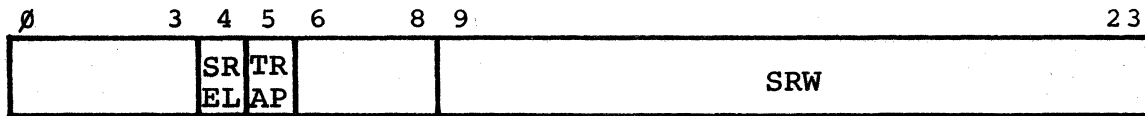


| Bits    | Name            | Contents                    |
|---------|-----------------|-----------------------------|
| 0:0-1   | IAT             | 3                           |
| 0:2     | LB              | lower bound for IR (0 or 1) |
| 0:3     | ATRAPP          | array trap bit              |
| 0:4     | LEB             | large element bit           |
| 0:5-6   | MULT IF LEB = 0 | multiplier for IR           |
| 0:5-10  | MULT IF LEB = 1 |                             |
| 0:7-23  | UB IF LEB = 0   | upper bound for IR          |
| 0:11-23 | UB IF LEB = 1   |                             |

J. String Words

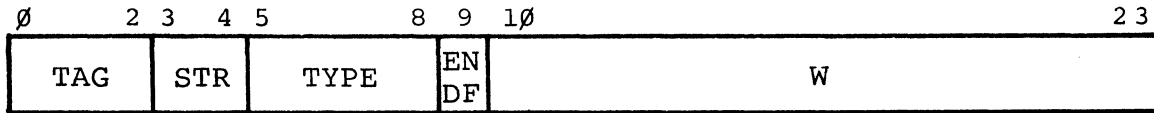


## K. BLL Branch Descriptor

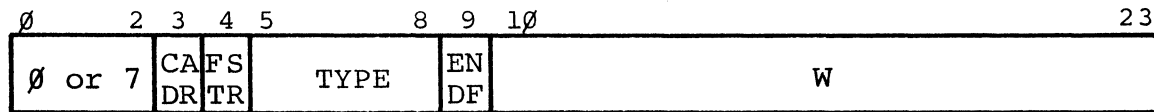


| <u>Word</u> | <u>Bit</u> | <u>Name</u>    | <u>Meaning</u>  |
|-------------|------------|----------------|---|
| 0           | 4          | SREL           | c.f. REL + SR in Normal IAW   |
| 0           | 5          | TRAP           | Causes TRP if set   |
| 0           | 9-23       | SRW            | Signed displacement if SREL is set  |
| 0           | 6-23       | LW             | Long word addresses   |
| 1           | 0          | CLL            | Call bit. The old P and L are saved if the bit is set.                    |
| 1           | 1          | STK            | The local environment is allocated from the stack if this bit is set.     |
| 1           | 2          | CPA            | Arguments are copied if this bit is set                                   |
| 1           | 3          | CPR IF CLL=1   | The CPA bit in the return descriptor is turned on if this bit is set.     |
| 1           | 3          | UWSTK IF CLL=0 | Unwind stack on return  |
| 1           | 4          | REL            | Source-relative label supplied  |
| 1           | 5          | FTN            | 1 FORTRAN type function   |
| 1           | 6-23       | E              | This number determines the new L; precisely how it does so depends on STK |

## L. Actual Argument Word (AAW)

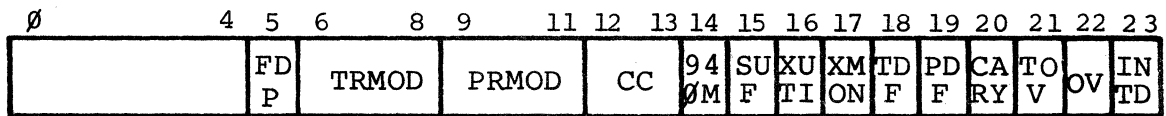


## M. Formal Argument Word (FAW)



| <u>Bits</u> | <u>Name</u> | <u>Contents</u>  |
|-------------|-------------|--|
| 3-4         | STR         | (actual argument only) structure<br>1 = variable<br>3 = computed scalar<br>2 = array element<br>0 = array  |
| 3           | CADDR       | (formal argument only) copy value<br>1 = copy address of actual argument<br>0 = copy value of actual argument  |
| 4           | FSTR        | (formal argument only)<br>1 = scalar<br>0 = array  |
| 5-8         | TYPE        | type<br>0 = jump<br>1 = integer (1 word)<br>2 = long (2 words)<br>3 = real (2 words)<br>4 = double (4 words)<br>5 = complex (4 words)<br>6 = longlong (4 words)<br>7 = string (4 words)<br>8 = label (2 words)<br>9 = pointer (1 word)<br>14 = unknown |
| 9           | ENDF        | end flag<br>0 = not last argument word<br>1 = last argument word   |

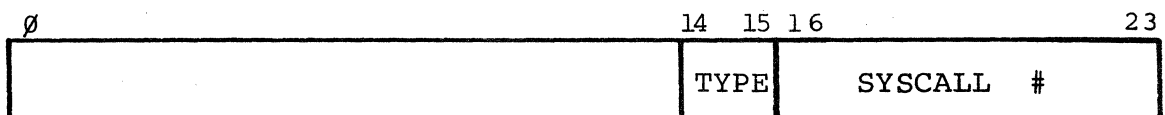
## N. State and Status Register



CC = 0      A < 0  
           1      A = 0  
           2      A > 0

| Trap State addresses | Save addresses | Relative addresses | Name |
|----------------------|----------------|--------------------|------|
| 602752B              | 602764B        | 0B                 | P    |
| 602753B              | 602765B        | 1B                 | A    |
| 602754B              | 602766B        | 2B                 | B    |
| 602755B              | 602767B        | 3B                 | C    |
| 602756B              | 602770B        | 4B                 | D    |
| 602757B              | 602771B        | 5B                 | E    |
| 602760B              | 602772B        | 6B                 | X    |
| 602761B              | 602773B        | 7B                 | L    |
| 602762B              | 602774B        | 10B                | G    |
| 602763B              | 602775B        | 11B                | SR   |
|                      | 602776B        | 12B                | CTC  |
|                      | 602777B        | 13B                | IT   |

## O. Absolute Value of Negative OPR Operand



| Type | Meaning |
|------|---------|
| 0    | UCALL   |
| 1    | UCALN   |
| 2    | MCALL   |
| 3    | MCALN   |

## SOME FIELDS IN OCTAL FORM

|            |                         |
|------------|-------------------------|
| 300000000B | X                       |
| 700000000B | LR                      |
| 700100000B | SR                      |
| 700200000B | SRI                     |
| 700340000B | IM                      |
| 020000000B | I in IAW                |
| 040000000B | X                       |
| 161000000B | SR                      |
| 166000000B | SRI + RELX              |
| 164000000B | LR + RELX               |
| 010000000B | TRAP                    |
| 200000000B | FIELD IAW               |
| 100000000B | SE                      |
| 076000000B | SIZE                    |
| 001740000B | FB                      |
| 400000000B | STRING IAW              |
| 140000000B | Csize                   |
| 030000000B | CPOS                    |
| 600000000B | ARRAY IAW               |
| 100000000B | LB                      |
| 040000000B | ATRAP                   |
| 014000000B | MULT if LEB = 0         |
| 020000000B | LEB                     |
| 017600000B | MULT if LEB = 1         |
| 400000000B | CLL bit in BRD          |
| 200000000B | STK                     |
| 100000000B | CPA                     |
| 040000000B | CPR / UWSTK             |
| 020000000B | variable AAW            |
| 060000000B | scalar                  |
| 040000000B | array element           |
| 040000000B | copy address, array FAW |
| 020000000B | copy value, scalar      |
| 060000000B | copy address, scalar    |
| 000400000B | ENDF                    |



## CHT HASHING ALGORITHM

In this hashing algorithm consider a byte to be 8-bits. Also consider the unique name as being composed of six 8-bit bytes (BYTE1 - BYTE6).

The algorithm is:

```
HASH'UN1 ← BYTE1 E' BYTE2 E' BYTE3;  
HASH'UN2 ← BYTE4 E' BYTE5 E' BYTE6;  
HASH'UN ← HASH'UN1 E' HASH'UN2;  
HASH'UN ← HASH'UN E' 264B;  
HASH'UN ← HASH'UN A' 377B;
```