

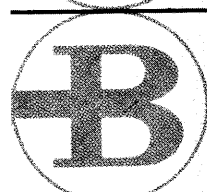
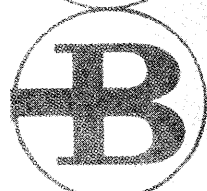
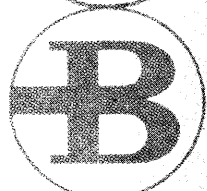
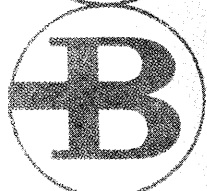
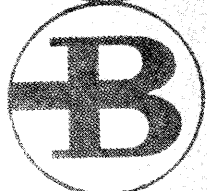
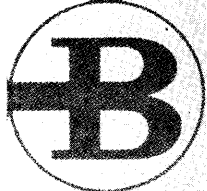
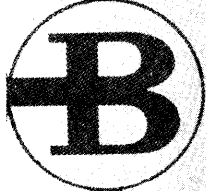


**PROGRAMMING
FOR THE
D851 MODULAR DATA PROCESSING SYSTEM**

PROGRAMMING FOR THE D851
MODULAR DATA PROCESSING
SYSTEM

BC

Burroughs Corporation



**PROGRAMMING
FOR THE
D851 MODULAR DATA PROCESSING SYSTEM**

MARCH 1964

Burroughs Corporation

CONTENTS

SECTION 1.	INTRODUCTION	1-1
SECTION 2.	PROGRAMMING TECHNIQUES	2-1
2.1	ASSEMBLY LANGUAGE NOTATION	2-3
2.2	INDEXING	2-6
2.3	ADDRESSING MODES	2-11
2.4	LITERALS	2-31
2.5	D851 ARITHMETIC OPERATIONS	2-34
2.6	FIELD OPERATIONS	2-42
2.7	JUMP OPERATIONS.	2-45
2.8	OPERATIONS INVOLVING LOCAL DATA BUFFER	2-50
2.9	STACK MANIPULATION	2-53
2.10	TAG BITS	2-58
2.11	ASSEMBLY ERROR INDICATIONS	2-67
2.12	PSEUDO-OPERATIONS.	2-68
2.13	MEMORY-BOUNDS REGISTER	2-69
2.14	NON-COMMUTATIVE OPERATIONS	2-70
2.15	STORAGE QUEUE	2-71
2.16	TIMING	2-72

CONTENTS (Cont'd)

SECTION 3.	INSTRUCTION REPERTOIRE	3-1
3.1	ARITHMETIC INSTRUCTIONS	3-4
3.2	SHIFT INSTRUCTIONS	3-9
3.3	LOGICAL INSTRUCTIONS	3-13
3.4	BIT-MANIPULATING INSTRUCTIONS	3-21
3.5	LITERAL INSTRUCTIONS	3-25
3.6	FETCH AND STORE INSTRUCTIONS	3-26
3.7	INDIRECT ADDRESSING INSTRUCTIONS	3-32
3.8	STACK-MANIPULATING INSTRUCTIONS	3-35
3.9	INDEX INSTRUCTIONS	3-38
3.10	JUMP INSTRUCTIONS	3-40
3.11	CONTROL INSTRUCTIONS	3-53
3.12	PSEUDO-OPERATIONS	3-60
APPENDIX A	THE CASE FOR POLISH NOTATION	A-1
APPENDIX B	TABLE OF POWERS OF 2	B-1
APPENDIX C	NUMERICAL LISTING OF INSTRUCTIONS	C-1
APPENDIX D	ALPHABETICAL LISTING OF INSTRUCTIONS	D-1

PREFACE

This publication represents the first formal documentation of the programming techniques, operations, and instructions which have been developed as part of the software program for the D851 Modular Data Processing System. This information is presently being augmented and incorporated into a comprehensive and detailed D851 programming manual.

This document consists of three sections and four appendices. Section 1 contains an introduction which states the D851 approach to programming, and summarizes some of the features that aid programmers. Section 2 is a general discussion of programming techniques that can be used by experienced programmers to better exploit the full possibilities of the D851 system. Some background material is included on the implementation of arithmetic modes and the manipulation of the stack. Section 3 lists the individual assembly language instructions and their descriptions. Appendix A discusses the virtues of Polish notation. A table of the powers of 2 comprises Appendix B. Appendix C lists instructions by operation code, and Appendix D lists them alphabetically.

Descriptors, interrupt capabilities, and other similar software topics are not discussed in this document, but appear in the proposal to which this document is appended.

The forthcoming programming manual will not only include the above data, but will present new and important material made available through continuing exploitation of programming possibilities.

SECTION 1

INTRODUCTION

The D851 Modular Data Processing System is designed to incorporate all reasonable features which simplify programming, yet permit full realization of inherent system flexibility, expansibility, and high throughput. The design philosophy may be simply stated: the more sophisticated the system becomes, the less painstaking the programming should become. Since programming facility, perhaps more than any other single system aspect, is a prime consideration in determining a system's value to the user, programming has received paramount attention in the design of the D851. And in areas where hardware and software trade-offs have been involved in regard to methodology, decision has invariably favored software where high usage can be expected. To a large degree, programming needs have dictated the hardware design, and the D851 programmer finds system programming straightforward.

Burroughs experience on the D825 and D830 modular systems has been used as a basis for the advanced design of the D851. Experience gained in building and programming other machines of a classified nature has also been applied to the D851. Some of the principal features contributing to the flexibility, ease of coding, ease of debugging, and simulation of other machines are the D851 arithmetic computational capabilities, the addressing modes and indexing possibilities, and the special provisions made for minimizing and detecting program errors.

A great deal of attention has been given to the selection and implementation of the D851 arithmetic operations. As a result, normalized floating point, fractional fixed point, and integer arithmetic are provided, both because of their wide usage and for compatibility purposes. A form of uniform representation of numbers is built into the D851 Computer Module to facilitate

automatic conversion between integer and floating-point modes. Specified point and significant digit are also provided as computational modes.

In addition to these five arithmetic modes of computation, the Computer Module incorporates the alphameric and logical modes of computation. Double precision, extended double precision, and field-defined operations are included to augment the D851 data processing and character manipulating capabilities.

Four addressing modes are available to the D851 programmer. Self-relative addressing and addressing relative to either of two base registers provide for relocation of programs, ease of subroutine coding and loading, and handling arrays. The modular programming permitted by the D851 design allows several programmers to work simultaneously on portions or modules of a program system.

Absolute addressing is provided principally to simplify the inter-program modular communications monitoring functions of the Executive and Scheduling Program (ESP).

In the D851 system, any word in a Memory Module can be used as an index register. This unusual flexibility provides, for all practical purposes, an unlimited number of index registers, and facilitates multi-indexing. Numerous indexing instructions also contribute to the indexing and multi-processing capabilities of the D851. To obtain the most efficient use of the high speed processing built into the D851 system, each Computer Module contains its own associative memory. The eight most recently used index words are always available through the use of a part of the associative memory.

The D851 system's machine instructions and assembly language are designed to make programming considerably less tedious than on the older generations of computers. The D851 features the use of a push-down stack and the associated Polish notation to further simplify programming, increase operation speeds, and reduce the possibility of undetected programming errors.

A number of provisions have been made to assist a programmer in debugging a new program. For example, the operation codes 00 and 77 (octal) automatically interrupt the execution of a program in normal mode. Thus, a program is likely to be interrupted if it attempts to execute data words as instructions, and the clearing of unused memory areas to ZERO's causes an immediate interrupt if an undebugged program executes a jump to an area outside of itself. One tag bit is set aside just for the purpose of assisting the programmer in determining which words in a Memory Module have been used or have not been used by the program. Other tag bit arrangements allow tagging instruction or data words to provide interrupts and/or snapshot dumps at points in the program selected by the programmer. Jumps can also be initiated by the tag bits, both for debugging and for terminating loops.

SECTION 2

PROGRAMMING TECHNIQUES

This compilation of programming techniques is presented as a guide for experienced programmers in understanding those areas of the discipline which are both essential and unique to the D851. It is assumed that the users of this manual will have a knowledgeable understanding of such elementary areas as binary notation, computer usage, organization and nomenclature, assembly language coding, indexing, executive scheduling routines, and debugging.

The discussions in this section utilize language accepted by the D851 assembly program; actual machine language is discussed only when it aids descriptions of programming options and techniques or when it facilitates explanations of results obtained when assembly-language instructions are executed in object programs.

Most coding for the D851 will probably be in the language of a compiler such as FORTRAN, ALGOL or JOVIAL. However, information on hand coding the Burroughs D851 Modular Computer System is included because a discussion of hand coding describes operations and programming capabilities of the D851 which would be obscured by a discussion of compiler language programming. In addition, information on hand coding may be useful for special problems. It will be used for programming the Executive and Scheduling Program (ESP), as well as the FORTRAN, ALGOL, and JOVIAL compilers.

2.1 ASSEMBLY LANGUAGE NOTATION

Assembly inputs are normally punched on standard 80-column cards, although other input media are accepted by the program. A specimen assembly-language coding sheet is shown in Figure 2-1.

2.1.1 LOCATION FIELD

The location field is used for notation of location symbols and program points.

2.1.1.1 Location Symbols

A location symbol must not be coded so that it is equivalent to any of the reserved mnemonics. Such a symbol will be refused by the assembly program, and an error will be indicated.

The location-symbol field may be blank, in which case the instruction or data defined by the card may not be specifically referenced elsewhere in the program. Asterisks in columns 1 and 2 denote a remarks or comments card; the card is printed in the assembly listing but is ignored in the assembly process.

Columns 1 through 8 of the coding sheets define a location symbol, the name by which the instruction or data defined by the remainder of the card may be referenced elsewhere in the program. A location symbol may consist of any mixture of from one to eight characters, as long as at least one character is non-numeric, and none of the following is included: asterisk (*), slash (/), plus (+) sign, minus (-) sign. Parentheses may not be included as characters within location symbols.

2.1.1.2 Program Points

A single asterisk in column 1 defines the one to seven characters which may follow it as a "program point", or a reference location symbol. When a program point is referenced elsewhere in the program, the entire location symbol except the asterisk is written, followed by either a plus (+) to indicate a forward reference to a program location not yet defined, or a minus (-) to indicate a backward reference to a program location already defined by the assembler. The placing of a + or - sign at the end of a program point denotes it as a program point, rather than a location symbol.

Parentheses may not be included as characters within program points. However, a program point may be enclosed in parenthesis to distinguish the terminating + or - from an operator in address arithmetic.

A D851 program consists of a string of six-bit syllables, with eight syllables to an instruction word. Instructions may start any place in a word, and a single instruction may overlap two words. Jump addresses include syllable numbers so jumps may occur to instruction starts within words. A location symbol or program point does not cause the assembly program to start a new word. Instead, the syllable number is recorded by the assembly program when a location symbol or program point is defined, and all references to the defined symbol or point automatically generate the correct syllable number for a jump. When desirable, instructions may be started at the beginning of a word by the use of special pseudo-operations. Syllable numbers have no meaning in data fetches and stores; data-generating pseudo-operations generate discrete data words.

All words in the D851 Memory Modules are 52 bits in length. Only 48 bits of the word are available for storage of instructions or data. Three of the remaining bits are used as tag bits, and the fourth bit is used as a parity-checking bit.

Although the assembly program allows address arithmetic to be performed, the syllable structure of D851 instructions makes it preferable to specify jump addresses via location symbols and program points, rather than via a specification such as "present contents of location counter plus five". Programmer-specified relative addressing is ambiguous. In the example given it is not clear whether "plus five" means five instruction words, five instructions, or five syllables.

2.1.2 OPERATIONS (OP) FIELD

Columns 10 through 15 constitute the operations field. This field may include any of the instructions, system macros, or pseudo-operation symbols defined for the D851 assembly program; this full list is given in Appendix C. A programmer-defined macro-instruction may also appear in this field, if defined previously by a "MACRO" pseudo-operation.

2.1.3 ADDRESS FIELD

The address field starting in column 17 is interpreted in a manner depending on the contents of the operations field. The address field may contain information other than actual addresses, such as variants or specifications. Except for the HOL (Hollerith) and LHOL (literal Hollerith) pseudo-operations, the address field is terminated by the first blank column within it; anything following is interpreted as a remark.

2.1.4 IDENTIFICATION FIELD

Columns 71 through 80 of the coding sheets are reserved for card-numbering and program-identification codes. These columns are always ignored in assembly, but the contents are printed in the assembly listing.

2.1.5 FIELD SEPARATION

Columns 9 and 16 on the Burroughs D851 Symbolic Assembly Form must be blank to indicate field termination, or the assembly program will indicate a possible error. However, these columns may be used on remark cards.

2.1.6 EXAMPLES OF PROGRAM POINTS

Program points eliminate the need for a programmer to generate numerous symbols to reference short portions of coding. They may also be used as an internal reference to short independently-coded subroutines which are to be assembled with larger programs. The use of program points in this manner reduces the coordination required among several programmers writing several sections of a large program.

Program points may be coded without the restrictions placed on location symbols; the use of a program point clearly designates what it is. The following is acceptable coding:

```
*1   OP   - - -
*2   OP   - - -
      OP   1
      OP   . 2
```

Program points may be combined with location symbols and constants in address arithmetic. There are two acceptable formats for assuring that a plus or minus following a string of characters is interpreted as designating a program point rather than being taken as an operator in address arithmetic. The program-point designation, including the plus or minus, may be enclosed in parentheses, or the program-point designation may terminate the expression. As an example, program point "LOCX+" minus five may be indicated in either of the following ways:

```
(LOCX+)-5
-5+LOCX+
```

The following would be flagged as an error:

```
LOCX+ -5
```

2.2 INDEXING

The index registers used by a D851 Computer-Module program are 52-bit words in main memory. The number of index registers which may be used by a program is limited only by the available storage in memory. However, the Computer Module automatically retains the eight most frequently used index-register words in local storage; a program is almost never delayed for a fetch of an index word from a Memory Module.

2.2.1 INDEX WORD FORMAT

The format of an index word is as follows.

BITS	CONTENTS
30-47	Index contents
12-29	Limit or Refill Value
1-11	Increment Magnitude
0	Increment Sign

The index-contents field contains the value which may be used to modify an address; this field corresponds to index registers in most computers, and may be compared or tested to control conditional jumps.

The limit-field is normally used to provide a value against which the contents are compared. This comparison may be for less-than, greater-than, equality, less-than or equal-to, greater-than or equal-to, or inequality. Both the limit and contents fields are unsigned; negative quantities are represented in 2's complement form. In this format, two negative quantities or two positive quantities may be compared without difficulty, but a negative quantity appears larger than a positive quantity. However, in 2's complement there is only one representation of zero; there is no minus zero.

A secondary use of the limit field is the refill operation (XRL or JX--RL), in which the contents field is replaced by the limit field of the same index word.

The increment, being signed, may also be used as a decrement. It may be added to the index contents as part of the same instruction which uses the contents to modify an address, as part of the same instruction which tests the contents field, or in a separate instruction.

2.2.2 COMPARISON INVOLVING THE STACK

In addition to comparing an index contents with its own limit, the contents may be compared with zero, or with a limit supplied from the stack. A one may be subtracted from or added to the contents, the contents may be replaced from the stack, or an unsigned 18-bit increment may be added or subtracted from the stack.

In general, the index operations involving the stack are slower than those which utilize the self-contained limit, the implied limit of zero, the self-contained increment, or the implied increments of plus or minus one. When the stack is not involved, the advanced station (ADVAST) executes the instruction fully and continues without waiting. When the stack is involved, however, the ADVAST stops until the final station (FINST) has emptied the final processing queue (FINQ) and has provided the required value from the stack.

2.2.3 DESIGNATION OF INDEX REGISTERS

When a program designates an index register, it does so by number. However, the "EQU" pseudo-operation may be used to define a symbol equal to an index number. The absolute memory address of the designated index word is the sum of the designated number and the contents of the base index register (BXR). In other words, BXR contains the absolute memory address of the word which is currently index register zero. Whenever any address is specified relative to BXR, either by the use of "X" in an address field or by designation as an index, the absolute address generated is automatically compared with the absolute addresses of words currently held in the associative memory of the Computer Module. If the address corresponds to one already in the Computer Module, no access to a Memory Module is made, and the word is retrieved from the local memory. If access to a Memory Module is required, the least-recently used word is dropped from the associative memory to make room for the new word. This newly-specified index word is retained in the associative memory until room is needed for another word. When a word is removed from the associative memory, it is returned to its original address in a Memory Module so it is never lost to the program.

2.2.4 ASSOCIATIVE MEMORY

The Computer Module always holds eight index words and their associated Memory addresses. The following sequence of operations illustrates how the associative memory keeps the most recently used index words in the Computer Module. For this explanation, the eight words initially in the Computer Module are designated by the letters "A" through "H" in that order, and index words "I" and "J" are in Memory Modules.

<u>Index Used</u>	<u>Resulting Order in Computer Module</u>	<u>Index Fetched</u>	<u>Index Stored</u>
	A B C D E F G H		
D	D A B C E F G H		
H	H D A B C E F G		
I	I H D A B C E F	I	G
F	F I H D A B C E		
J	J F I H D A B C	J	E

The Computer Module always keeps the last eight words used relative to BXR in the order of their last use. Most index-words in most programs are used repeatedly for a while, then a new set is used, and the originally used ones may then be used again. Thus, the associative memory and the ordered list of eight index words allow most programs to define as many different index words as there are different functions for indexes (or subscripts) in the program, with the assurance that the hardware will automatically keep the most likely required index words in fast-access storage.

The interrogation of the associative memory requires less than 0.1 micro-second; the interrogation time is included in the instruction times listed in Table 2-8. Whether relative to BXR or not, all fetches and stores have their addresses compared with associative memory. If a word is stored relative to BXR and a later fetch not relative to BXR addresses the same absolute address in a Memory Module, the program fetches the most recent version of the word, which is the version retained in the Computer Module. If a word is stored relative to BXR and a later store (not relative to BXR) addresses the same absolute address in a Memory Module, the version of the word held in the Computer Module is updated. However, if a store to a Memory Module is neither relative to BXR nor involves the same absolute address as a word already in the associative-memory storage within the Computer Module, no entry is made in associative memory; the word is just transmitted to a Memory Module. Conversely, when a word is changed in associative memory, the Memory Module version of the word is not updated until room is required in associative memory for later insertion.

2.2.5 SUBROUTINES

All entries to subroutines increment BXR, and the return information is stored in the new index register zero (the word pointed at by the new contents of BXR). The programmer can control the amount of increment to be added to BXR on subroutine entry as long as the increment is not zero. Thus, a subroutine may use index registers freely since the parent program has stepped BXR past the set of index registers which the parent program wished to save. The BXR-stepping may also be used to save local variables via fetching and storing relative to BXR.

The old setting of BXR is retained in the least significant 18 bits of index register zero while a subroutine is being executed. This setting corresponds to the contents field of index register zero. If a subroutine wants to access an index register or local variable which is retained in the stepped-past portion of memory by the parent program or the next higher level of nested subroutines, it may do so by absolute addressing modified by index register zero. As an example, the following coding would fetch the contents of the memory word which was index register five before the current subroutine was entered.

```
FMS      5/0,A
```

If subroutines are nested, the successive index-register-zero words form a linked list back to the BXR setting for the parent program. That is, the current index register zero links to the previous word which was index register zero; this word links to the previous one, and so on.

2.2.6 ADDRESS INDEXING

The operation of indexing an address is actually accomplished by a separate set of index instructions, although assembly-language coding will generally imply these instructions by the use of slash (/) in the address field of another instruction. Indexing is executed by the addition (2's complement if subtraction is required) of the contents field of the designated index word to the previous contents of the address register. Multiple indexing, indicated as in the following examples, is accomplished by repeated indexing instructions; each indexing instruction adds the contents of an index word to the previous sum in the address register.

```
FMS      JOE/1/2/3
SSM      LØCX/1/1/2
SJF      INPUT/O
```

As shown by the above examples, multiple indexing by a single index register is permissible; this method gives indexing by the product of an integer and the index contents.

It should be noted that the D851 index register zero is a true index register; the use of zero in an indexing field does not denote lack of indexing.

2.2.7 USE OF SELF-CONTAINED INCREMENT

When the self-contained increment is to be added to the index contents after an address has been modified by addition of the old contents to the address.

register, this requirement may be denoted by an asterisk immediately following the slash.

```
FMS      JOE/*1/2/*3
```

In the above example, index registers 1 and 3 are modified by addition of their self-contained increments after they are used. Index register 2 is not modified after its use.

An alternate coding which is closer to the actual machine language uses "X" as a mnemonic for indexing without modification, and "XM" as a mnemonic for indexing with modification. The following coding is an alternate representation of the example given above of the use of asterisks in index fields.

```
XM      1
X       2
XM      3
FMS     JOE
```

2.2.8 INDEXING WITH MODIFICATION OF INDEX WORD

The mnemonics "X" and "XM" each may assemble as either of two actual octal instructions. The indexing steps shown above would result in the following string of octal syllables.

```
05 01 04 02 05 03
```

The octal op-code syllables 04 and 05 instruct the processor module to use the next syllable as the index designation; 04 results in simple indexing (X) and 05 in indexing with modification (XM). Since a single syllable may reference only 64 words as index registers, there are alternate instructions provided that permit the use of any word in memory as in index. However, the assembly program chooses the proper machine instruction as a function of the number of bits required to designate the index specified. If "X" is used in coding, or if the asterisk is omitted when indexing is specified in an address field, octal code 04 results for index registers 0 to 63, and octal code 06 for index registers 64 to 262143. Similarly, the assembly program chooses between octal codes 05 and 07 for indexing with modification.

The octal coding shown below the following example of assembly-language coding is the result of the indexing operations.

```
FMS      JOE/*64/65/*66
07 00 01 00 06 00 01 01 07 00 01 02
```

For convenience in discussing machine-language operations, the 06 op-code is sometimes designated as "XA" (index augmented) and 07 as "XAM" (index augmented and modify index-word). It is recommended that these mnemonics be avoided in assembly-language coding. If they are used, the 18-bit index-designation fields are generated even when they are not required.

An example is given below.

XA	1
XAM	2
XM	3

06 00 00 01 07 00 00 02 05 03

Although the above coding would have the desired result if the object program were executed, storage would be wasted.

2.3 ADDRESSING MODES

The Burroughs D851 Modular Computer System may have up to sixteen Memory Modules containing 16,384 words each. An 18-bit address field is used to permit direct access to 262,144 memory words by any Computer Module. Address arithmetic is performed by having the advanced station of the instruction processor place results in an 18-bit address register (AR). This address register may be considered as the accumulator of the ADVAST.

There are four modes of memory addressing provided: self-relative, absolute, relative to base data register (BDR), and relative to base index register (BXR). All four modes may be used for transmission of data. Each mode may be used with indirect addressing, and with single or multiple indexing. Jump addresses are specified in self-relative and absolute modes only.

2.3.1 SELF-RELATIVE ADDRESSING

Self-relative addressing is defined as addressing relative to the position in memory of the instruction which is doing the addressing. If a program is entirely self-contained, assembled, or compiled as a single entity with no communication outside of itself until completed, self-relative addressing allows relocation of the program without modification of addresses by a loading routine or modification of base registers.

2.3.1.1 Normal Mode of Addressing

Self-relative addressing is the normal mode of addressing memory in a D851 program; the assembly program considers all addresses to be self-relative

except where the coding specifically designates another mode. However, self-relative assembly does not require programmer specification of self-relative increment or decrement.

2.3.1.2 Assembly Program Calculation of Self-Relative Address

With self-relative addressing, all programs are assumed to start at a nominal origin of zero; origin cards are not needed for assembly. When the assembly program encounters a location symbol, it records the word address relative to the nominal origin of zero and the syllable number.

Instructions may consist of one or more syllables, and multisyllabic instructions may overlap adjacent memory words. Both for assembly purposes and actual machine operation, the location of a multi-syllabic instruction is defined as the location of the first syllable of the instruction. When a location symbol is used in the address field to refer to another section of the program, the assembly program subtracts the address of the instruction which makes the reference from the address recorded for the referred-to symbol. The resulting difference is inserted in the binary object program string as the self-relative address.

2.3.2 ABSOLUTE ADDRESSING

The retention of absolute addressing in the D851 concurs with the general design objective of retaining features available on past-generation computers while offering new techniques, and of expanding rather than replacing capabilities of older equipment.

2.3.2.1 Uses

Absolute addressing may be used for simplified inter-program communication such as the specification by ESP of a subroutine location to be used by several other programs. Absolute addressing is actually used by the D851 in all inter-module transmissions of data, regardless of the mode specified by the program. For example, if a program fetches a data word from memory by self-relative addressing, the instruction processor computes the absolute address before requesting the word from the Memory Module containing the data word. However, this use of absolute addressing occurs without program intervention and, in general, "behind the programmer's back".

2.3.2.2 Method of Specifications

Absolute addressing is specified by the use of the letter "A" in the address field of an assembly-language subject-program, preceded by a comma and

following the address specification. Some examples of assembly-language specification of absolute addressing follow:

```
OP          5, A
OP          0/1, A
OP          LØCX - LØCY + 62000, A
```

Specification of an addressing mode other than self-relative is always by a single letter separated from the address symbol by a comma. Indexing is denoted by a slash (/) following address specification. Both indexing and addressing mode must be specified after the address. Although it is recommended that the indexing be specified before addressing mode to make coding uniform, the assembly program will accept and properly handle coding specifying the address mode before indexing. Thus, both of the following assembly inputs would result in the same binary coding of the object program.

```
OP          LØCX/1, A
OP          LØCX, A/1
```

Although the following coding is not recommended, the assembly program would accept it to mean "modify the defined value of LØCX by index registers 1 and 2 and treat the result as an absolute address":

```
OP          LØCX/1, A/2
```

Since absolute addressing is expected to be used primarily when a location symbol is defined by means other than its position within a program, the pseudo-operations which define symbols may be used to define the addressing mode to be used when the symbol is referred to.

```
JØE EQU 5, A
```

The above coding defines location symbol "JØE" as absolute value 5. When "JØE" is referenced elsewhere, it is treated as though the letter "A" were appended. The two following examples would now yield the same object program coding:

```
OP JØE
OP JØE, A
```

The second example above, although redundant, is not treated as an error. If the definition of a symbol specifies an addressing mode and the reference to the same symbol specifies another mode, the mode specified in reference is accepted but a possible error is indicated. The following coding would

result in addressing mode "R" being used in the second line of coding but with an error indication.

```
JØE    EQU    5, A
        OP    JØE, R
```

2.3.3 INDIRECT ADDRESSING

All arithmetic performed in indirect addressing is performed by the control arithmetic unit (CAR) in conjunction with the address register in the advanced station of the instruction processor. Indirect addressing can be programmed by using an asterisk in the operation field, or by using the three instructions provided specifically for indirect addressing: Fetch from Memory to Address Register (FMA), Add from Memory to Address Register (AMA) and Fetch from Local Data Buffer to Address Register (FLA).

2.3.3.1 Use of Address Register for Indirect Addressing

The use of the address register in the D851 Computer Module may be easily remembered by considering it as the equivalent of an accumulator for ADVAST of the instruction processor. ADVAST and CAU perform all address computations in the Computer Module, and its effective accumulator may be programmed in a simple manner to allow various mixes of indirect addressing and indexing.

Since indirect addressing is performed by ADVAST which normally runs a few microseconds ahead of the final station (FINST), most uses of indirect addressing will not impede execution of the over-all program. Although ADVAST is delayed while waiting for a pointer to come from a Memory Module, in most cases ADVAST will not be delayed sufficiently to affect the speed of the final station.

2.3.3.2 Use of an Asterisk

In the following example, indirect addressing is designated by having an asterisk follow the instruction mnemonic.

```
ADD*    JØE
```

This example utilizes self-relative addressing for both the indirect-address pointer (the word at location JOE) and the pointed-at word to be added (given by the contents of location JOE). The self-relative indirect addressing is achieved by the AMA instruction which adds the contents of the addressed

word to the contents of the address register. At the time the AMA is executed, the address register contains the address of the word which is to be added to it. Therefore, the contents of the addressed word (JOE in the above example) give the position of the pointed-at word relative to the position of the pointer.

However, instructions AMA and FMA may utilize any of the addressing modes permitted elsewhere: self-relative, absolute, relative to BDR, or relative to BXR. The instruction which utilizes the address generated by AMA or FMA may also be in any of the modes. The convention adopted by the assembly program is that if indirect addressing is indicated by an asterisk in the operations field, the mode of addressing indicated by the coding in the address field applies to both the fetching of the indirect-addressing pointer and the use of the fetched address. The address generated by the FMA or AMA is used unchanged; the instruction which follows and uses the indirect address has an included address of zero.

2.3.3.3 Use of Indirect-Addressing Instructions

While the use of an asterisk in the operations field is the easiest way to program a simple case of indirect addressing, it does not allow the programmer to use the flexibility inherent in the D851. In particular, it is often desirable to index the access to a table of pointers and to index, by a different index register, the access to a pointed-at table, or to access a word displaced by a known amount from the pointer value. It also may be desirable to use different addressing modes to access the pointer and the pointed-at word. These and other combinations are programmable by using the actual indirect-addressing instructions rather than by implying them with asterisks. For example, the following coding accesses an absolutely addressed table of pointers, modifying the access by index register 1, and fetches the fifth word of the pointed-at table relative to the base data register and modified by index register 2.

FMA	PØINTERS/1, A
FMS	5/2, R

2.3.3.4 Addressing the Local Data Buffer Indirectly

The instruction which allows an indirect address to be fetched from the local data buffer (FLA) is included for subroutines. A parent program can put the address of a data area in the local data buffer, and a subroutine can use this to indirectly address the data area.

If an indirectly addressed instruction denoted by an asterisk uses the letter "L" in its address field, both the indirect-addressed pointer and pointed-at operand are fetched from the local data buffer.

2.3.3.5 Standard Methods of Programming Indirect Addressing

Obviously, the method of indirect addressing discussed so far require programmer specification of the depth of nested indirect addressing. The index by top of stack instruction, however, does permit any specified bit or group of bits to be tested to determine if indirect addressing is to be repeated. Thus, this instruction allows one of the standard methods of indirect addressing to be programmed. There are two other methods available, both of which allow the processor program to continue with some other operations. Hence they both tend to be faster than the index by top of stack method.

If a word fetched to the local data buffer was tagged empty in memory (STB EMPTY or STB 11X), the communication unit treats this word as a request for the fetch of the word indicated by the least significant 18 bits of the word (30-47). This pointed-at word which is fetched to the local buffer may also be "empty," causing repetition. Care must be taken to avoid initiating a block transfer to local data buffer.

Another method of indirect addressing uses the linked-list searching ability of a Memory Module. Since the test for search completion may be the test of a single bit, this method also provides what has generally been called indirect addressing.

The machine-language versions of the instructions which test and/or modify an index register do not contain the designation of the index register involved. The Computer Module always operates on the word in associative memory which was most recently used as an index. When a blank address field is used in assembly-language coding of an index test and/or modify instruction, the index operated upon is the one most recently used; it is assembled as a test and modify instruction without designation of an index. When a non-blank address field is used, the contents of the address field designate the index register which is to be the operand. This assembles as an index instruction preceding the index test and modify. Although the indexing operation always adds to the address register, the index test and modify clears the address register; the only net effect of the indexing operation is to move the designated index word to the most recently used position in associative memory.

In a simple iterative loop in which only one index is designated, a programmer may test this index for completion of iteration without specifying its number. Also, some routines may be simplified by utilizing the most recently-used index, rather than by having the program determine which index was most recently used.

Since any word in memory may be used as an index, indexing may be considered a form of indirect addressing. However, to take advantage of the speed gained by the use of associative memory, indexing as such must address relative to BXR. Self-relative, absolute, and relative to base data register addressing may not be used. Furthermore, indexing of an index designation may not be performed since repeated use of an index instruction results in multiple indexing of whatever address follows. Therefore, other means of indirect addressing are provided which may prove more convenient. However, the distinction on the D851 between indexing and indirect addressing is somewhat arbitrary. Any address which may be indexed may be indirectly addressed, and vice versa.

The instruction called index by top of stack (XS) allows the addition of the top of the stack directly to the address register. This instruction may be used for a form of indirect addressing; fetching the pointer to the stack and then using it as an address. The instruction allows variable-depth indirect addressing since any test may be performed in the stack to determine if further levels of indirect addressing are required. Furthermore, the XS instruction permits a computed address to be utilized directly without being returned to main memory. However, for most uses of indirect addressing there are faster methods available, as the XS instruction forces the advanced station to stop until the final station is ready to provide the computed address at the top of the stack.

2.3.4 ADDRESSING RELATIVE TO BASE DATA REGISTER

The use of the letter "R" following address specification denotes addressing relative to the base data register (BDR), mnemonically "relative addressing". Addressing relative to the BDR is not permitted for jump-addressing. The BDR allows ESP to point to a discontinuous data area which may arise when ESP loads a data block and determines from its contents or header which program is required to process it. The required program may already be in memory and would generally not be contiguous to the independently-loaded data block. The BDR is used in the D851 FORTRAN compiler to point to "COMMON". In assembly-language coding, BDR would be similarly used for data areas common to several programs.

2.3.5 ADDRESSING RELATIVE TO BASE INDEX REGISTER

Addressing relative to the base index register (BXR) is denoted by the letter "X" following address specification, and is not permitted for jump-addressing. The BXR contains the absolute address, in main memory, of the word currently designated as index register zero. The following coding means "store or set the value of index register 5".

SMS 5, X

2.3.5.1 Uses

The primary use of BXR-relative addressing is for storing index words, as in the example above. However, the special features of the D851 which provide a virtually unlimited set of fast-access index words make BXR-relative addressing useful for quantities other than indexes.

2.3.5.2 Communications-Limited Programs

The normal functioning of the advanced station (ADVAST) fetches data words without delaying a program in all cases but those that are "communications-limited". For a D851 program to be delayed by the functioning of the communication unit (COMM), the average time required by the final station must be less than 0.5 microsecond for each data word that is fetched. This delay can occur only for non-arithmetic programs such as sorts or searches. However, in such programs the same small block of words is not usually required repeatedly, and the addresses are usually determinable in advance. Thus, other means of speeding communications (see discussion of local data buffer below) are available when BXR-relative addressing is not effective in speeding program operation.

2.3.5.3 Associative Memory

When BXR-relative addressing is specified, the addressed memory word is treated as though it were an index word. If the word is not already in the computer's associative memory, it is fetched from a Memory Module and put into the associative memory. The word is then retained in the associative memory for repeated fast-access use while as many as seven "newer" words are specified relative to the BXR. When the eighth newer word is put into the associative memory, the word under discussion is returned to the Memory Module. Thus the associative memory, which was designed to keep the eight most recently-used index words in fast-access local memory, may be used whenever a programmer expects the same memory words to be accessed repeatedly as data, but the addresses of the repeatedly-accessed words are not determinable when the program is being written.

2.3.5.4 Subroutines

When a subroutine is entered, the BXR is stepped by an amount specified by the programmer. BXR stepping is used to permit a subroutine to use index registers while preserving the index registers being used by the parent program. This is further explained in the sections dealing with indexing and subroutines. However, the stepping of BXR, coupled with BXR-relative

addressing of data, permits a subroutine to have a "personal" storage area in Memory Modules. Data stored and fetched relative to BXR are in absolute memory locations which are functions of the processor exercising the subroutine and the depth of subroutine nesting. This is particularly useful for recursive subroutines; BXR-relative stored data may be used for local variables and other modes of addressing for universal variables. The same binary coding of BXR-relative addressing accesses different memory words as a function of depth of recursion.

2.3.6 ADDRESSING THE LOCAL DATA BUFFER STORAGE

The letter "L" following an address designation denotes storage in the 64-word local "scratch-pad" or local data buffer storage. All addressing of the local data buffer storage is in absolute; indirect addressing and indexing are permitted. Any assembly-language instruction in which addressing of memory is permitted may be designated as operating on the local data buffer storage within the Computer Module by the letter "L". Local storage also has parallel data-transfer capabilities, described elsewhere.

Access to the local data buffer is accomplished by machine-language instructions different from those accessing the Memory Modules. The assembly program automatically generates these special instructions in response to the letter "L". However, the instructions accessing the local data buffer may be alternately specified in the op-code field of the assembly-language form. When this is done, an "L" in the address field is accepted as a non-erroneous redundancy. The specification of absolute, relative-to-BDR, or relative-to-BXR addressing is flagged as an error, and is ignored. A location symbol may be defined as a location in local data buffer in a manner identical with that used for defining symbols in terms of their addressing modes.

2.3.7 ADDRESSING OF REGISTERS

All registers in a D851 Computer Module can be designated by number or mnemonic. The mnemonics, corresponding numbers, and descriptions of the registers are given in Table . Whenever one of these mnemonics is used in the address field of an instruction it is interpreted as the designated register. Even if the instruction is one which does not normally address a register it is interpreted as addressing the designated register. For example, the field register is register number 00 and mnemonic "FR". The following coding is interpreted as "fetch the contents of the field register to the stack, and add to the word previously at the top of the stack".

ADD FR

The register mnemonics are reserved for this use; the assembly program will refuse to define one of the reserved mnemonics for any other use.

A programmer-designated symbol may be defined as a register number number:

```
FIELD EQU FR
```

If the above example is followed by any reference to "FIELD", it is interpreted as a reference to the field register.

2.3.8 OPTIONAL ADDRESSING

2.3.8.1 Compiler-Generated Coding

The following coding shows the machine operations actually performed in adding the words "A", "B", and "C" from memory when the memory addresses are modified by index registers 1, 2, and 3 respectively.

```
X      1      Index by Index Register 1
FMS    A      Fetch Word "A" from Memory to Stack
X      2      Index by Index Register 2
FMS    B      Fetch Word "B" from Memory to Stack
ADD                    Add Words "A" and "B"
X      3      Index by Index Register 3
FMS    C      Fetch Word "C" from Memory to Stack
ADD                    Add Word "C" to Previous Sum
```

2.3.8.2 Assembly-Language Coding

Although the above coding may be convenient in many cases and is actually the Polish notation that would be generated by a compiler, most programmers would prefer to write the above program in the following assembly language.

```
FMS    A/1     Fetch Word "A", Modified by Index Register 1
ADD    B/2     Add Word "B", Modified by Index Register 2
ADD    C/3     Add Word "C", Modified by Index Register 3
```

When the assembly program finds coding in the address field of an ADD, it assembles the address field as though it had been the address field of a fetch preceding the addition. If the address field of an ADD is blank, it is assembled as an addition of the two top operands in the stack.

2.3.8.3 FORTRAN CODING

Some programmers may prefer coding in Polish notation with no addresses given for arithmetic instructions, and with operands always obtained from the stack. However, even if the one-address assembly format is generally preferred, there are some cases in which the no-address Polish format greatly simplifies programming. Consider the following equation, written in FORTRAN format.

$$A = B * C + D * E$$

The FORTRAN compiler would code the above statement in the following manner, using Polish notation which eliminates the need for temporary storage anywhere but in the stack.

FMS	B	Fetch B to Top of Stack
FMS	C	Fetch C to Top of Stack, Pushing B Down
MUL		Generate B*C
FMS	D	Fetch D to Top of Stack, Pushing B*C Down
FMS	E	Fetch E to Top of Stack, Pushing E and B*C Down
MUL		Generate D*E; B*C Moves Up
ADD		Add B*C to D*E
SSM	A	Store A

2.3.8.4 Hand Coding

To do the same operation as efficiently in hand coding, the following coding may be used.

FMS	B	Fetch B to Top of Stack
MUL	C	Generate B*C in Top of Stack
FMS	D	Fetch D to Top of Stack, Pushing B*C Down
MUL	E	Generate D*E in Top of Stack
ADD		Add B*C to D*E
SSM	A	Store A

The above is an example of how the no-address Polish format using intermediate results as operands can be used instead of the wasteful and more complex technique of placing intermediate results in temporary storage.

2. 3. 8. 5 Use of Address Fields

With respect to the use of address fields, instructions may be placed into three classes: instructions requiring an address field, instructions in which the use of the address field is optional, and instructions in which use of an address field is meaningless.

2. 3. 8. 5. 1 Instructions Requiring an Address Field. The instructions in which the use of the address field is mandatory are divided into three groups: (1) instructions which may not be indexed or indirectly addressed, (2) instructions which may be indexed but not indirectly addressed, and (3) instructions which may be indexed or indirectly addressed. Each group is considered below.

2. 3. 8. 5. 1. 1 Group (1). The instructions in which the use of the address field is mandatory, but which are neither indexable or indirectly addressable, are listed below. The use of the address field for each instruction is placed in parentheses.

RTS	Rearrange Top of Stack (Designation of Rearrangement Required)
LIT	Literal (Literal Value - Not Indexable when Greater than 2 ¹⁸)
X	Index (Index-Register Number)
XA	Index Augmented (Index-Register Number)
XM	Index and Modify Index Word (Index-Register Number)
XAM	Index Augmented and Modify Index Word (Index-Register Number)
ESP	Enter Executive and Scheduling Program (Entry Number)
SFCN	Perform Special Function (Function Number)

The following two pseudo-operations also belong in group (1):

EQU	Equate Symbol in Location Field to Coding in Address Field
REP	Repeat the Coding between REP and the Next "END" the Number of Times Designated

2. 3. 8. 5. 1. 2 Group (2). The three instructions in which the use of the address

field is mandatory and which are indexable but not indirectly addressable are:

SLIT	Short Literal (Literal Value)
FAS	Fetch from Address Register to Stack (Value, when used as an 18-Bit Literal)
LIT	Literal (Literal Value - Indexable when Less than 2^{18})

2.3.8.5.1.3 Group (3). The instructions in which the use of the address field is mandatory, and which are indexable or indirectly addressable are:

SB	Set Bit (Bit Number)
CLB	Clear Bit (Bit Number)
CHB	Change Bit (Bit Number)
INB	Insert Bit (Bit Number)
SJF	Set Up Jump if False (Jump Address)
SJT	Set Up Jump of True (Jump Address)
SJSF	Set Up Jump to Subroutine if False (Jump Address and BXR Increment)
SJST	Set Up Jump to Subroutine if True (Jump Address and BXR Increment)
JCB	Jump on Result of Condition-Bit Test (Bit Number)
JBT	Jump on Result of Bit Test (Bit Number)
II ϕ	Interrupt Input/Output Program (Module Number)
I ϕ	Initiate Input/Output Program (Module Number)
CCM	Communicate with Computer Module (Module Number)

The following all require a memory register, or LDB address:

FMA	Fetch from Memory to Address Register
AMA	Add from Memory to Address Register

FLA	Fetch from LDB to Address Register
FMS	Fetch from Memory to Stack
FML	Fetch Single Word from Memory to LDB
FBML	Fetch Word-Block from Memory to LDB
FCML	Fetch Character-Stream from Memory to LDB
FLS	Fetch from LDB to Stack
SSL	Store from Stack to LDB
SLP	Set LDB Pointer
SSM	Store from Stack to Memory
FAS	Fetch from Address Register to Stack (When Not Used as an 18-bit Literal)
FRS	Fetch from Register to Stack
SSR	Store from Stack to Register
DLM	Dump LDB to Memory
LML	Load from Memory to LDB
DRM	Dump from Registers to Memory
LMR	Load from Memory to Registers
L	

2.3.8.5.2 Instructions with Optional Address Fields. The instructions in which the use of the address field is optional are also divided into three groups, which are numbered from 4 to 6 to distinguish them from the previous groups. The groups are (4) those instructions not designating a fetch, and which may not be indexed or indirectly addressed, (5) instructions not implying a fetch, but which may be indexed, and (6) instructions implying a fetch operation.

2.3.8.5.2.1 Group 4. Instructions in which an address field is optional, but does not designate a fetch, and which are not indexable or indirectly addressable

are listed below. In the following, a non-blank address field designates an index number.

JXEZ	Jump on Index Equal Zero Test
JXES	Jump on Index Equal Stack Test
JXEL	Jump on Index Equal Limit Test
JXGL	Jump on Index Greater than Limit Test
JXGS	Jump on Index Greater than Stack Test
JXLL	Jump on Index Less than Limit Test
JXLS	Jump on Index Less than Stack Test
XRS	Replace Index Content by Top of Stack
JX--RS	Test for Jump, and Replace Index Content by Top of Stack
XA \emptyset	Add one to Index Content
JX--A \emptyset	Test for Jump, and Add One to Index Content
XAI	Add Index Increment to Index Content
JX--A1	Test for Jump, and Add Index Increment to Index Content
XAS	Add Top of Stack to Index Content
JX--AS	Test for Jump, and Add Top of Stack to Index Content
X $S\emptyset$	Subtract One from Index Content
JX-- $S\emptyset$	Test for Jump, and Subtract One from Index Content
XRL	Replace Index Content by Index Limit
JX--RL	Test for Jump, and Replace Index Content by Index Limit
XSS	Subtract Top of Stack from Index Content
Jx--SS	Test for Jump, and Subtract Top of Stack from Index Content

In the following computational mode instructions a non-blank address-field

denotes field definition, double-precision, or alternate field-definition.

INT	Integer
SPP	Specified-Point
FLT	Floating-Point
SIP	Significant-Point
FIX	Fixed Point
ALPH	Alphameric
LOG	Logical

2.3.8.5.2.2 Group 5. In the instructions listed below, an address field specification is optional. However, when the address-field is used, it does not imply a fetch.

SR	Shift Right
SL	Shift Left
RR	Rotate Right
RL	Rotate Left

For the above instructions, a blank address field designates a shift count already in the top word of the stack. Address-field specification of a shift count assembles as a literal placing the count in the top of the stack. Indexing of the shift count assembles as indexing of the literal. An asterisk in the operations field assembles as a fetch of the addressed word. Indexing, in this case, is interpreted as indexing of the indirectly-addressed shift-count.

JUMP	Unconditional Jump
------	--------------------

For the above instruction, a blank address field assembles as a jump to the address given by the most recently executed set-up jump. A non-blank address field assembles as a set-up jump followed by a jump, a jump to the location given by the address field. Indirect addressing of this jump address may be indicated by an asterisk in the operations field.

2.3.8.5.2.3 Group 6. The instructions listed below imply a fetch operation, and may be coded with or without specification in the address field. When the address field is blank, the instruction operates on one or two operands already

in the stack. When the address field is not blank, its contents are interpreted as the address of an operand. This address may be indexed or specified in any addressing mode. When any of these instructions is coded with an asterisk in the operations field, a non-blank address field is mandatory, and its contents are interpreted as the location of the indirect-addressing pointer designating the operand. Indexing may not be designated if the asterisk in the operation field is used. The mode of addressing is assumed to be the same for both access to the indirect-addressing pointer and the operand pointed at.

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
FLR	Float Remainder
SQR	Square Root
RND	Round
PØS	Set Positive
NEG	Set Negative
CHS	Change Sign
INS	Insert Sign
AND	AND
ØR	Inclusive OR
ØRX	Exclusive OR
FILL	Set to Full-Scale
CLR	Clear to ZERO's
CØM	Complement
IMP	Implication
EXT	Extract
INF	Insert Field
B0000	Boolean Function Zero
thru	
B1111	Boolean Function Fifteen
NØRM	Normalize
JUS	Justify
UNJ	Unjustify
DUP	Duplicate in Stack

DUPD	Duplicate Double-Precision in Stack
TRIP	Triplicate in Stack
QUAD	Quadruplicate in Stack
REV	Reverse in Stack
CYCU	Cycle Stack Up Once
CYCD	Cycle Stack Down Once
JGR	Jump on Result of Test for Greater
JGA	Jump on Result of Test for Absolute-Value Greater
JLS	Jump on Result of Test for Less
JLA	Jump on Result of Test for Absolute-Value Less
JEQ	Jump on Result of Test for Equality
JZE	Jump on Result of Test for Zero
JFS	Jump on Result of Test for Full-Scale
JSI	Jump on Result of Test of Sign

2.3.8.5.3 Instructions Not Requiring an Address Field

The instructions in which an address field never has any meaning are listed below.

SSH	Streaming Shift
STEP	Step Stack Up Once
RET	Return from Subroutine
RETI	Return to Interrupted Program
XS	Index By Top of Stack
NØP	No Operation
STØP	Stop Computer Module
ENM	Enter Normal Operational Mode
MEM	Initiate Memory-Module Program
WØRD	Fill Current Word with NOPs (Pseudo-Operation)
BLØCK	Fill Current Block with NOPs (Pseudo-Operation)

2.3.8.6 Indexing

Any address field which optionally results in a fetch may include indexing. Any of these instructions may be designated in the operation field as using indirect addressing by having an asterisk follow the mnemonic for the instruction, as follows.

ADD*	A
SUB*	A
AND*	A

If an asterisk is in the operation field but the address field is blank, it is flagged as an error and asterisk is ignored.

2.3.9 ADDRESS ARITHMETIC PERFORMED BY THE ASSEMBLY PROGRAM

The assembly program performs any address arithmetic that may be required by instructions for which addressing is permitted. Addition and subtraction are performed as indicated for symbols that are defined elsewhere, and for constants.

2.3.9.1 Negative Addresses and Subtraction

The address register, address fields in instructions, and content fields in index words are 18 bits in magnitude, and are unsigned. Negative self-relative addresses and all other negative address quantities are represented in the 2's complement form. This form is defined as the bit-by-bit complement of the magnitude of a negative binary number, plus 1 in the least significant bit-position. This form has been chosen instead of the 1's complement (bit-by-bit complement of magnitude without addition of 1) because the 1's complement has two valid representations of zero (all ONE's and all ZERO's) which tends to complicate some operations in which index registers modify addresses.

Effective subtraction in 2's complement arithmetic is accomplished by 18-bit unsigned addition with overflow bits discarded. This is the manner all address arithmetic is performed in the D851.

2.3.9.2 Truncation of Addresses

The result of address arithmetic is truncated to the size of the field designated to hold the result. In other words, an n-bit field will contain the least significant n bits of the result of address arithmetic. In the case of an instruction addressing main memory, an 18-bit field is used. An index designation may be 6 or 18 bits; the 18-bit designation is used whenever the assembly program determined that truncation to the six bits would eliminate any non-ZERO bits. The designation of an address in local data buffer or of a register is always truncated to six bits.

Address arithmetic is also permitted in the address fields of instructions operating on individual bits. In this case, the least significant six bits of the result are interpreted as a bit number. If the resulting bit number is greater than forty-seven, it is treated modulo forty-eight.

2.3.9.3 Coding of Constants

The constants involved in address arithmetic must be integers, and are interpreted as decimal unless specified as octal (or boolean) by having the letter "B" terminate a string of digits.

Five examples of how "constants" are interpreted are given below:

Case 1:	77
Case 2:	7.7
Case 3:	7B7
Case 4:	77B
Case 5:	88B

Case 1 is interpreted as a decimal because there is no terminating "B". Case 2 is invalid in address arithmetic because it is not an integer. Case 3 is interpreted as a location symbol because the "B" does not terminate it: this symbol must be defined elsewhere. Case 4 is valid for octal 77. Case 5 is valid as a defined symbol since it obviously can not be octal.

It is invalid to use as a symbol a string of numeric digits which excludes 8's and 9's, and which is terminated by a single letter "B", since this is the designation of an octal constant in address arithmetic. A string of numeric characters terminated by a single "B" is a valid symbol, if the numeric string contains at least one 8 or 9. However, such a symbol is not recommended.

2.4 LITERALS

A literal is an operand specified in the program string, rather than fetched from elsewhere in memory from an address specified by the program string. Any constant appearing in an input statement to the D851 FORTRAN compiler is compiled as a literal. Assembly language programs may use literals either by using one of the literal mnemonics in the operations field, or by using an equal sign (=) in the address field of any instruction for which optional addressing is permitted. For example, each of the following operations adds 1.5×10^{18} , a floating-point number, to the top of the stack. The LIT instruction is implied in the second example and places the designated constant, in floating-point format, in the top of the stack.

1. LIT 1.5E18
 ADD
2. ADD =1.5E18

The assembly program does not maintain a record of the computational mode being used in various parts of an assembled program. Therefore, the specification of the format to be used for literals must be supplied by the programmer. This is useful when a programmer wants to generate an index-setting as part of a program run in floating-point mode, and does not want the literal specifying the index-setting to be converted to floating-point format by either the assembler or the Computer Module.

2.4.1 DECIMAL LITERALS

Three types of decimal literals are recognized by the D851 FORTRAN compiler: decimal integers, floating-point numbers, and fixed point numbers. Decimal literals are specified by the character =; this character is followed by the decimal literal-type information described in the following paragraphs.

2.4.1.1 Decimal Integers, Literals

Decimal integers consist of a group of digits, from 0 to 9, which represent quantities 1 up to $2^{35} - 1$ for single precision computation, and $2^{82} - 1$ for extended double precision. Decimal integers may be preceded by a plus or minus sign. The two other forms of decimal literals use the characters E, B, and a decimal point. Therefore, decimal integers may be identified by the absence of these characters. Decimal integers are represented by the following examples:

- = -9
- = 947683

2.4.1.2 Floating-Point Literals

Floating-point literals have two parts: a principal part (mantissa) and an exponent. The floating-point number is identified relative to decimal integers by the presence of character E or a decimal point, and relative to fixed point numbers by the absence of character B. Floating point numbers are represented by the following examples:

```
= -9167482.1E12  
= .124
```

The principal part consists of a decimal number which may include a sign or contain a decimal point placed at either end or within the decimal number. The exponent contains the character E and a decimal integer which may be signed or unsigned. When the principal part of the number contains a decimal point, the exponent is not required. When used, however, the exponent must follow the principal part.

2.4.1.3 Fixed-Point Literals

Fixed-point literals have three parts: a principal part, an exponent, and a binary place. The fixed-point number is identified by the presence of the character B and is represented by the following examples:

```
= 6492.3B9  
= 94B18E6
```

The principal part consists of a decimal number which may include a sign or contain a decimal point placed at either end or within the decimal numbers.

The exponent contains the character E and a decimal integer which may be signed or unsigned. When the principal part of the fixed-point number contains a decimal point the exponent is not required. When used, however, the exponent must follow the principal part.

The binary-place contains the character B and a decimal integer which may be signed or unsigned. The binary-place must be present in a fixed-point number; it is placed after the principal, either before or after the exponent. If a binary-place contains more than two digits the number is truncated and only the first two digits recognized.

2.4.2 OCTAL LITERALS

An octal literal is identified by the characters = and \emptyset , followed by a string of digits, 0 through 7. Octal literals may be signed or unsigned and are represented by the following examples:

```
=    $\emptyset$ 147762
=    $\emptyset$ -35 77 $\emptyset$ 1364122
```

2.4.3 ALPHAMERIC LITERALS

An alphameric literal is identified by the characters = and H (Hollerith), followed by alphameric characters. Each of the six characters is interpreted as data even though one or more may be a blank or comma. Alphameric literals are represented by the following examples:

```
=   H $\emptyset$ , TACb (b = blank)
=   HRZ1276
```

When the literal is specified in the operations field, the mnemonic op-code may be any of the following. The rightmost column refers to data-generating pseudo-operations which are discussed elsewhere. The address-field coding, in each case, must correspond to the rules described for the designated data-generating pseudo-operation.

LIT	Decimal, Single Precision	DEC
LITD	Decimal, Double Precision	DECD
L \emptyset CT	Octal	\emptyset CT
LVFD	Variable Field Definition	VFD
LXR	Literal Specification of Index Register	CXR

When any of the above coding is used, the assembly program chooses the actual literal instruction, of the three described below, which allows the literal to be generated as specified with minimum storage requirements. For example, a zero is always assembled as a short (6-bit) literal.

Indexing of an alphameric literal is permitted if, and only if, the literal specified is less than 2^{18} . If effective indexing of a longer literal is required, it must be programmed by (1) indexing a short literal zero, which effectively places the index-contents in the stack, (2) generating the long literal without indexing, and (3) adding (ADD) the index-contents to the long literal in the stack. To be meaningful, this addition must be performed in fixed-point, alphameric or logical computational mode.

If a programmer generates a literal integer as an operand in a floating-point operation, the integer is automatically converted to floating-point when used as an operand. The result is always correct but time is taken for the conversion. The reverse is also possible: floating-point operand in an integer operation. However, conversion of floating-point to integer may result in an overflow or loss of significance; any fractional portion of a floating-point operand is always lost when used in integer operations.

2.5 D851 ARITHMETIC OPERATIONS

The seven basic computational modes of operation built into the D851 Computer Module are set up by the computational mode instruction which is described below. Details of each basic computational mode are given, both for single-length and double-length operands. The effects of field modification is discussed, and the section concludes with the sign conventions that are employed.

2.5.1 COMPUTATIONAL MODE INSTRUCTION

The computational mode instruction (op-code 03--) allows variations to be made in the way certain logical, arithmetic, shift, and jump instructions are executed, without requiring changes in the format of these instructions. The 03-- instruction sets up the mode of computation for all of the affected instructions which follow it, until a different 03-- instruction is used to set up a different computational mode. The 03-- instruction requires 0.1 microsecond of FINST time and 0.1 microsecond of ADVAST time to set up a computational mode.

2.5.1.1 Instructions Affected by Computational Mode

The instructions that are affected by the Computer Module's mode of computation are listed in Table 2-1.

TABLE 2-1
INSTRUCTIONS AFFECTED BY COMPUTATIONAL MODE

<u>LOGICAL</u>	<u>ARITHMETIC</u>
24 Implication	50 Add
25 Inclusive OR	51 Subtract
26 AND	52 Multiply
27 Exclusive OR	53 Divide
32 Complement	54 Square Root
33 Set to Full-Scale	55 Round
34 Clear to Zero	
<u>SHIFT</u>	<u>JUMP (on result of test for:)</u>
40 Shift Right	64 Greater
41 Shift Left	65 Absolute Value Greater
42 Rotate Right	66 Less
43 Rotate Left	67 Absolute Value Less
44 Normalize	70 Equality
	71 Zero
	72 Full Scale

2.5.1.2 Format of Computational Mode Instruction Word

The op-code for the computational mode instruction is 03 followed by a two-digit syllable that specifies the type of mode. The composition of the instruction word is shown in Figure 2-2.

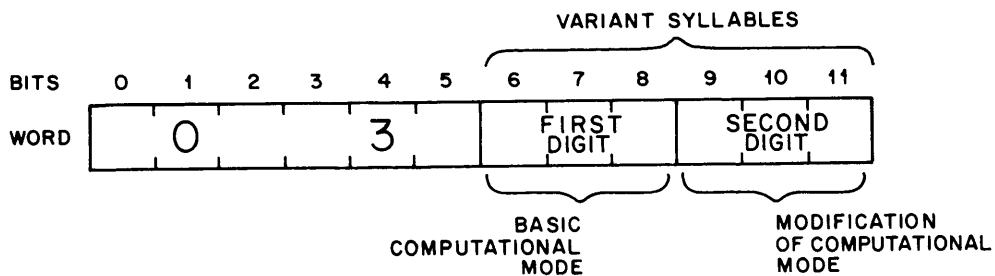


Figure 2-2. Composition of "03" Instruction Word

The first digit of the variant syllable determines which of seven basic computational modes is specified. These basic modes are listed in Table 2-2. The second digit of the variant syllable determines if the basic computational mode

TABLE 2-2

BASIC COMPUTATIONAL MODES

MNEMONIC CODE	BASIC COMPUTATIONAL MODE	FIRST DIGIT OF VARIANT SYLLABLE
INT	Integer	0
SPP	Specified Point	1
FLT	Floating Point	2
SIP	Significant Point	3
FIX	Fixed-Point (Fractional)	4
ALPH	Alphameric	5
LØG	Logical	6
	(undefined)	7

TABLE 2-3

POSSIBLE MODIFICATIONS TO BASIC COMPUTATIONAL MODES

MNEMONIC CODE SUFFIX	COMPUTATIONAL MODE MODIFIED BY	SECOND DIGIT OF VARIANT SYLLABLE
(none)	(no modification)	0
D	*Double Precision	2
F	Field Modified	4
ALT	Alternate Field-Defined	6

* In the alphameric and logical modes, "double precision" refers to "double length" operands (96 bits).

is field modified, uses double precision, or both. The possible four variations of the two modifications are listed in Table 2-3. For reference, a compilation of the op-codes and mnemonic codes for all possible computational modes is given in Table 2-4.

2.5.2 BASIC COMPUTATIONAL MODES

The way in which the 48-bit operands in the stack are treated is given below for each of the seven basic computational modes listed in Table 2-2.

2.5.2.1 Integer Mode

In the integer mode of computation, the 48-bit operands consist of a sign (bit 0) and 35 bits of magnitude. The radix point is to the right of bit 47, as shown in Figure 2-3.

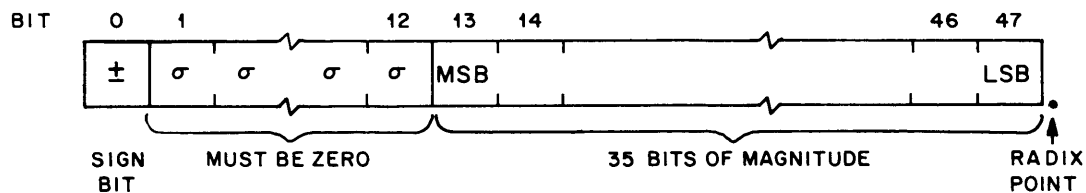


Figure 2-3. Integer Operand, Single Precision

Bit 13 is the most significant bit, and bit 47 is the least significant bit. Bits 1 through 12 must be ZERO's in the integer mode of operation because these bits contain the exponent of floating-point operands. If bits 1 through 12 are not all ZERO's, the processor assumes this non-zero condition indicates a floating-point word, and converts it to an integer word before proceeding.

In extended double precision operation (op-code 0302) the 47 magnitude bits of a second word are appended to the first word, giving 82 bits of magnitude. The radix point in this case is to the right of bit 47 in the second word.

2.5.2.2 Specified-Point Mode

Specified-point computation is performed as follows: Retain bits 1 through 12 of the top of the stack as the specified exponent, and step the stack up one. Enter floating-point operation but adjust any result so its exponent equals the specified exponent. If a result is too large to be so represented, floating-point overflow occurs. If a zero exponent occurs, a wrong-format error is indicated. The operands for specified-point computation are similar to those shown in Figure 3-3 for floating computation.

TABLE 2-4
CODES FOR ALL COMPUTATIONAL MODES

OP CODE	MNEMONIC CODE	OP CODE	MNEMONIC CODE
03 00	INT	03 40	FIX
03 02	INT D	03 42	FIX D
03 04	INT F	03 44	FIX F
03 06	INT ALT	03 46	FIX ALT
03 10	SPP	03 50	ALPH
03 12	SPP D	03 52	ALPH D
03 14	SPP F	03 54	ALPH F
03 16	SPP ALT	03 56	ALPH ALT
03 20	FLT	03 60	LOG
03 22	FLT D	03 62	LOG D
03 24	FLT F	03 64	LOG F
03 26	FLT ALT	03 66	LOG ALT
03 30	SIP	03 70	(undefined)
03 32	SIP D	03 72	(undefined)
03 34	SIP F	03 74	(undefined)
03 36	SIP ALT	03 76	(undefined)

2.5.2.3 Floating-Point Mode

Operands in the floating-point mode of computation use a sign (bit 0), 12 bits of exponent, and 35 magnitude bits for the mantissa, as shown in Figure 2-4. The sign applies to the mantissa.

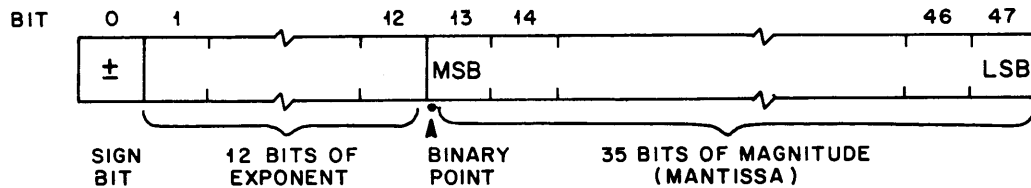


Figure 2-4. Floating-Point Operand, Single Precision

In extended double-precision computation (op-code 0324) the binary point remains to the left of bit 13 in the first word, and the 47 magnitude bits of the second word are appended to the 35 bits of the first word to provide a mantissa of 82 bits.

The exponent has a maximum value of 7777 (octal), and is interpreted as the value coded in the floating-point word minus 4000 (octal). Thus the floating-point operand is treated as

$$(\text{Magnitude}) \times 2^{(\text{exponent} - \text{octal } 4000)}$$

Floating-point overflow occurs if the exponent of a result is greater than octal 7777 after automatic normalization. Floating-point underflow occurs if the exponent of a non-zero result is less than or equal to 0000. A zero result is converted to have a zero exponent.

A zero mantissa resulting from a floating-point computation is not considered to be underflow. Only significance that exists but cannot be represented is underflow, and this is not the case if, for example, a zero mantissa results from subtracting a number from itself.

2.5.2.4 Significant-Point Mode

Significant-point computation is performed as follows: Enter the floating-point mode of computation but, instead of normalizing a result, retain as many leading ZERO's in the magnitude of the result as there were in the operand with the greater number of leading ZERO's. If a zero exponent occurs, a wrong-format error is indicated.

2.5.2.5 Fixed-Point Mode

Operands in the fixed-point (or fractional) mode of computation use a sign (bit 0) and 47 bits of magnitude, as shown in Figure 2-5.

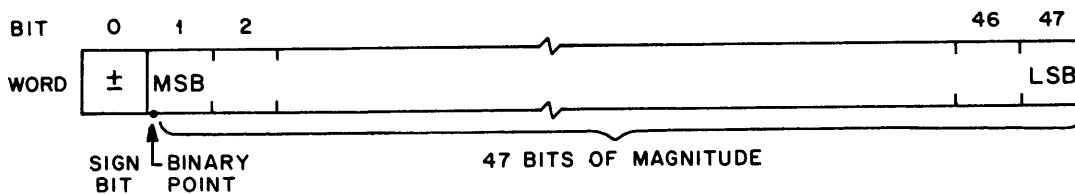


Figure 2-5. Fixed-Point Operand, Single Precision

In double-precision computation (op code 0342) the binary point remains at the left of bit 1 in the first word, and the 47 magnitude bits of the second word are appended to the first word, thus providing 94 magnitude bits. (Bit 0 in the second word has no significance in double-precision arithmetic modes where bit 0 in the first word is the sign bit.)

2.5.2.6 Alphameric Mode

In the alphameric mode bit 0 is not treated as a sign bit, but as a numeric bit which can be shifted along with the other 47 bits of magnitude. Therefore the sign is always considered positive. Double-length operands in the alphameric mode (op-code 03 52) provide 96 bits, designated as "magnitude" bits. The binary point is considered to be at the left of bit 0 in the first word, as shown in Figure 2-6. In the alphameric mode, an overflow is treated as fixed-point overflow; there is no end-around carry.

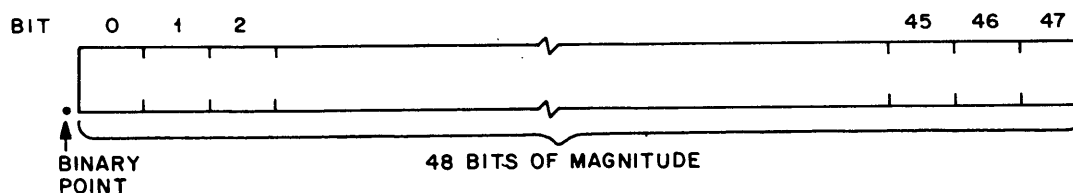


Figure 2-6. Alphameric Operand, Single Length

2.5.2.7 Logical Mode

The logical mode is identical to the alphameric mode except that an overflow causes end-around carry in the logical mode; there is no overflow indication given.

2.5.3 FIELD MODIFICATION OF BASIC COMPUTATIONAL MODES

Any of the seven computational modes that were shown in Table 3-1 may or may not be field-defined. When not field-defined, all bits of all operands are treated as found in the stack. When a word in the stack is field defined, only the bits of the operand that correspond to "ONE-bits" in the field register are subjected to the designated operation or instruction. If the operation causes a result to be returned to the stack, the result can only be put in the portion of the operand originally selected by the field register. The portion of the operand not selected by the field register remains unchanged in the stack. In the discussions of single-precision and double-precision field-defined modes which follow, words "A", "B", "C", and "D" refer to the four top words in the stack.

2.5.3.1 Field-Defined Modes with Single Length Operands

In field-defined words using one-operand instructions of 48 bits, only the portion of word "A" defined by "ONE bits" in the field register is subjected to the designated operation. The selected portion of the operand is obtained by "ANDing" (taking the logical product of) the operand together with the field register.

In two-operand instructions which are field defined the logical product of word "A" and the field register is used as one operand, and the logical product of word "B" and the field register is used as the other operand. If a result is returned to the stack, the "non-field" bits originally in word "B" are retained as the non-field bits of the result.

2.5.3.2 Alternate Field-Defined Modes (with Double-Length Operands)

Operations in the field-defined double-precision (double-length) mode using 96 bits of magnitude are essentially the same as operations described above for single-length (48 bit) operands. However, the field register has only 48 bits and is applied only to the first word of a double-length operand. The second word is not affected by the "ONE bits" in the field register. Therefore, only word "A" is subjected to the designated operation.

In two-operand instructions, the designated operation is only performed on the field-selected bits of words "A" and "C", since all bits of words "B" and "D" are not field defined. Thus the combination of field definition with double-length operands allows a form of operation upon alternate words of the stack.

2.5.4 SIGN CONVENTIONS

In integer, specified-point, (floating point, significant-point, and fixed-point computational modes, the sign bit of an operand is a true arithmetic sign: 0 = plus and 1 = minus. When these modes are executed in extended double precision, no use is made of the most significant bit of the least significant word of an operand. Shift instructions executed in the above computational modes do not shift or otherwise alter sign bits.

If an addition or subtraction results in a magnitude of zero, the sign bit is always set to ZERO; a zero result from addition or subtraction is always positive. In multiplication and division, however, the sign of a zero result follows the same rules as the sign of a non-zero result. In other words, a zero result from addition or subtraction is considered to be single-values and a true zero. When a zero results from multiplication or division, it is treated as an infinitesimal with a meaningful sign.

Sign bits are ignored in the logical or boolean operations which are not expected to be used frequently in these basically arithmetic modes. In comparison-type instructions, +0 is considered equal to -0 in arithmetic computational modes.

In logical and alphameric computational modes, there are no sign-bits; all operands are treated as positive. The most significant bit of a single-length operand and the most significant bits of each word of a double-length operand are treated as numeric bits. These leading bits are shifted along with the other bits, involved in logical or boolean operations, and added or subtracted like the other bits. In comparison-type instructions, a leading 1-bit makes a word larger than one with a leading 0-bit.

2.6 FIELD OPERATIONS

The D851 Computer Module contains a field register with a 48-bit mask which can be used for both contiguous and non-contiguous operations. The most important use of the field register is to control character-defined operations. Each character occupies successive bits in a word, and thus has a contiguous field. The D851 can also use the field register where non-contiguous or split fields are necessary, such as in command and control type programs. These programs may use the state of each bit in a word to denote whether or not a specific routine is required on the current pass. The setting of these bits is accomplished by routines which determine the requirements. However, the same routine is often required as a result of a number of different routines; each such evaluation recognizes the need for several routines in different combinations. The capability of using non-contiguous fields in such control operations is more convenient than being restricted to the insertion of a contiguous field by each evaluation routine.

2.6.1 Rules

The general rules for field definition are:

1. AND each operand of an affected instruction to the field register.
2. Perform the designated operation as though the logical product(s) had been the full operands.
3. If a result is generated, AND the field register to the result as normally generated.
- 4a. For a single-operand instruction, AND the operand to the complement of the field register, and OR this logical product to the logical product generated in step 3.
- 4b. For a two-operand instruction where both operands are obtained from the stack, AND the operand deeper in the stack to the complement of the field register, and OR this logical product to the logical product generated in step 3.
- 4c. For a two-operand instruction where one operand is obtained from memory, local data buffer, or a register, AND the operand originally in the stack to the complement of the field register, and OR this logical product to the logical product generated in step 3.

It should be noted that the preceding rules cause the result returned to the stack to have non-field bits (bits corresponding to ZERO's in the FR) equal to the non-field bits of the original single operand, the non-field bits of the operand originally deeper in the stack, or the non-field bits of the operand originally in the stack.

2.6.2 Shifting

When field definition is used with an instruction causing an effective shift, such as the actual shift instructions, multiply, divide, square root, or any floating-point operation, there may be the loss of some bits that are of interest.

For example, if a shift is performed with field definition, any bit shifted out of the field is lost unless it is shifted end-around, through the entire word, and to the other end of the field. However, field definition is most effectively used with the logical-type instructions (OR, AND, IMP, ORX, Bnnnn, CLR, or FILL). With these instructions, the operation is bit-by-bit, and there is no confusion between bits in the fields of operands and bits retained in the result that is returned to the stack.

2.6.3 Arithmetic Operations

When add or subtract is performed with field definition, the processor automatically recognizes overflow out of the field. In logical mode this operation generates an end-around carry to the right-most bit within the field. In the

alphanumeric mode overflow is discarded as usual. In any other computational mode, either a condition bit is set or an overflow interrupt is caused, depending on the setting of the mask register.

In addition or subtraction with a non-contiguous field, carry is executed from bit to bit of the defined field.

If the normalize (NORM) or shift and count (SAC) instruction is used with field definition, only the bits within the field are examined to determine the most significant 1-bit. However, this bit is shifted to the most significant bit of the mantissa or magnitude (as defined by the basic computational mode), and not to most significant bit within the field. If the most significant bit of the mantissa or magnitude is not defined within the field, the most significant 1-bit is lost. The barrel shift register (BSR) contains the bit position of the most significant 1-bit within the word, not within the field.

If the field register is set to all ONE's except for bit 0 (the sign-bit), field definition may be used for one form of absolute-value arithmetic. Consider, as an example, the following coding performed in field definition with only the sign bits being excluded from the field.

FMS	JØE
ADD	TØM

The magnitude of the result is the sum of the magnitudes of JOE and TOM; the sign of the result is the sign of JOE.

2.6.4 Implementing and Terminating Field Definition

Field definition may be superimposed on any computational mode by the use of the letter "F" in the address field of a mode-changing instruction, as in the following examples.

FLT	F
INT	F
ALPH	F
LØG	F
FIX	F
SPP	F

Although field definition may be designated in any mode, it will probably be used almost exclusively in the logical (LØG) or alphanumeric (ALPH) modes. To remain in the same computational mode but eliminate field definition, the basic mode mnemonic should be repeated in the operations field without the "F" in the address field.

As an example, the first of the following two instructions would cause the Computer Module to enter the logical mode of computation with field definition. The second instruction would allow the Computer Module to remain in the logical mode, but without field definition.

```
LØG      F
LØG
```

Of course, in actual use the two instructions above would be separated by intervening operations utilizing field definition.

2.7 JUMP OPERATIONS

A jump requires program specification of two items: (1) the address which is the destination of the jump, and (2) the conditions under which the jump is to be executed. In the D851 these items are specified by two separate instructions: the jump address instruction and a jump execution instruction.

2.7.1 Jump Address

There are four instructions which specify the jump address.

SJF	Set up Jump if False.
SJT	Set up Jump if True.
SJSF	Set up Jump to Subroutine if False.
SJST	Set up Jump to Subroutine if True.

Each of the above instructions defines a jump address in self-relative or absolute address form. Indexing and indirect addressing can be applied conventionally to modify jump addresses. The jump address must precede the instruction which actually executes the jump in the program string. It must be remembered that the true or false control of conditional jumps, and the indication of whether the jump is to a subroutine, is specified by the setup jump instruction rather than with the jump execution instruction.

The true/false control doubles the effective number of tests that can be performed without doubling the number of instructions that must be implemented and remembered. For example, when the "jump on greater" (JGR) instruction is preceded by SJT, JGR does execute "jump on greater", however, when preceded by SJF, JGR executes "jump on less than or equal to". Similarly, SJF followed by JEQ means "jump on unequal"; SJF followed by JZE means "jump on non-zero", and so on. An unconditional jump "JUMP" ignores the true/false indication used for setting up the jump.

The immediate effect of SJF or SJT is the computation of the jump address, in the designated computational mode and the transmission of the computed address to the Jump Control Register (JCR) along with a 0-bit for SJF or a 1-bit for SJT and an indication that the jump is not to a subroutine. The jump address includes the destination syllable, effectively a twenty-one bit address. However, the syllable number is not included in the address computation; it is literally specified in the program string. The following octal coding of an SJT instruction illustrates this.

20 14 00 02 05

The first syllable, 20 designates the instruction as some variant of set up jump. The first three bits of the next syllable are 001. The first zero designates self-relative, rather than absolute, addressing; the second zero designates a non-subroutine jump; and the 1-bit specifies a jump if the next test gives a true result. The next octal digit, 4, is the syllable-number of the addressed word which will be first executed if the jump occurs. The final three syllables, 00 02 05, give the self-relative jump address. This jump, if it occurs, is to syllable four of the word 205 (octal) after the instruction word containing the first syllable of this SJT instruction.

If jump addresses are always designated by location symbols or in program points, the assembly program properly computes the self-relative jump address and automatically inserts the correct destination-syllable number. It should be noted, although not needed for most programming, that the eight syllables of an instruction word are numbered from 0 through 7, with syllable 0 being the left-most one, the start of the instruction word. In the example given above, the jump to syllable 4 is to the fifth syllable of the word.

2.7.2 Jump Execution

When any set up jump instruction is executed, the JCR is changed. This signals the communication unit to automatically fetch the four-word block containing the addressed word. Thus, in almost all cases, the jumped to words are fetched before the words are needed as the result of a jump occurring. These words are held in a "siding" until replaced by another change of the JCR. If a loop is programmed which includes only one jump (the jump back to the beginning of the loop) the instruction words at the beginning of the loop are retained in the Computer Module until after the looping has been terminated. If a tight loop is programmed (a loop which fits within a four-word memory block) the instructions constituting the entire loop are retained in the Computer Module until the loop is terminated.

When a loop is to be iterated many times, its speed may be increased by the use of the "BLOCK" pseudo-operation, forcing the loop to begin at the start of a four-word block. Coding for simple loops may be compressed by specifying the iteration address before looping starts; this may only be done if the only jump in the loop is the iteration return. For example, the following

coding adds words from memory, indexing the words by index register 1, subtracting one from this index amount on each iteration and iterating if the index contents are still greater than zero.

SJF	LOOP	Set up Iteration Return Outside of Loop
BLOCK		Start LOOP at Start of New Block by Filling with NOPs.
LOOP ADD	WORDS/1	Add WORD from Memory, Indexed by XR1.
JXEZSØ		Iterate if XR1 = 0, subtract 1 from XR1.

The index-number need not be specified for the JXEZSØ instruction. Since only index register 1 is used during the loop, it is obviously the last one used when the index test is performed.

A conditional jump which tests an index and which either does not modify the index or modifies by the use of the self-contained increment, plus one or minus one, is executed faster than a conditional jump which requires the use of an operand from the top of the stack. The higher speed is achieved because, if the stack is not required, ADVAST of the instruction processor executes the jump and continues processing past it without waiting for final processing.

2.7.3 Conditional Jump Classifications

There are three classifications of conditional jump instructions, in terms of the coding in the address field.

<u>Class</u>	
1	Specification of Jump Condition Required in Address Field
2	Specification of Jump Condition Optional in Address Field
3	Jump Condition Never Specified in Address Field

The instructions in each class are listed in subsection 2.3.8.5.1. Class 1 includes instructions like JBT (Jump on Bit) which requires the specification of a bit-number in the address field. Class 2 includes all of the jump on index variants (JX----) in which the address field specification of an index number is optional. If an index number is omitted the last-used index is tested. Class 3 includes those instructions in which the jump condition is fully specified by the operations field. For example, when the address field is blank JGR (jump on greater) means "test for the operand deeper in the stack being greater than the operand at the top". When the address field is not blank it designates optional addressing; the fetch of a word is assembled ahead of the jump instruction. Therefore, the following coding means "test for the word at the top of the stack being greater than the word at location JØE."

JGR JØE

This order was chosen to simplify the use of Polish notation in compilation. The expression "A > B" is changed by the compiler to AB > and is compiled as:

```
FMS            A
FMS            B
JGR
```

2.7.4 Subroutines

A subroutine may be entered by any of the instructions which may execute a non-subroutine jump, if the jump instruction is preceded by jump to subroutine if false (SJSF) or jump to subroutine if true (SJST) instruction. There are three differences between jumps to subroutines and other jumps. First, a jump to subroutine must retain sufficient information to return to the parent program. Second, the return information is saved in index register zero; the BXR is stepped to provide a new word defined as index register zero. Third, a subroutine jump can only be to the beginning of a word. All system subroutines start at the beginning of words; the WORD pseudo-operation is used to force the beginning of any subroutine included as part of another program to the beginning of a word.

The coding for a subroutine jump set-up may include a decimal integer, representing the desired stepping of BXR, at the end of the address specification. If this is omitted, the BSR is stepped by 1. Maximum speed is obtained by stepping in multiples of four, for example:

1, 4, 8, 12, 16, 20, 24

An asterisk instead of a stepping-count designates the use of a count which the program has placed in the stack. This is illustrated in the following example.

```
1. SJST        SUBR, 4
2. SJST        SUBR
3. SJST        SUBR, *
4. SJST        SUBR, 5
```

The first example steps the BSR by 4 when and if the subroutine is entered; the four is entered in the program string. The second example generates an increment of 1. The first and second examples step the BXR at maximum speed because 1 and 4 are used. Examples three and four are performed more slowly. The third example uses the increment the program has placed in the stack but requires the advanced station (ADVAST) to wait for the final station

(FINST) to supply this value. The fourth example generates 5 as a literal in the top of the stack and then requires the same delay as the third example. The numbers given preference are coded in the three bits used by SJT and SJF for the syllable number. The following table shows the octal coding used for these three bits.

CODING	BXR INCREMENT
0	1
1	4
2	8
3	12
4	16
5	20
6	24
7	Obtained from top of stack.

Any specified increment, except the preferred ones, is assembled as a literal followed by the last coding in the table.

When a subroutine is entered, the return information is stored in index register zero, in the following format.

BITS	CONTENTS
30-47	Old BXR Setting
27-29	Syllable Returned To
9-26	Absolute Word Address Returned To
8	Fixed-Point Overflow Condition Bit
7	Floating-Point Overflow Condition Bit
6	Integer Overflow Condition Bit
5	Floating-Point Underflow Condition Bit
0-4	Computational Mode

The overflow and underflow condition bits are cleared when a subroutine is entered, and are taken from index register zero and re-stored when a subroutine returns to the parent program. This permits both the parent program and the subroutine to each use overflow independently for their internal control. The saving and restoration of computational mode allows, for example, a fixed-point subroutine to be executed by a floating-point program without complication. However, the mode is not changed on entry to a subroutine so a subroutine can operate in whatever mode it is entered.

2.8 OPERATIONS INVOLVING LOCAL DATA BUFFER

The local data buffer (LDB) is a 64-word buffer within the D851 Computer Module. It is used as a "scratchpad" memory when a program specifies "Fetch from Local Data Buffer to Stack" (FLS) and "Store from Stack to Local Data Buffer" (SSL), or when the letter "L" is in the address-field of other instructions as described under optional addressing. The final station (FINST) requires the same time (0.1 microsecond) to accept a word from the LDB as to accept a word from a Memory Module; however, when the LDB is accessed the communication unit is not involved. The LDB provides a convenient storage for parameters generated or fetched by a parent program and used by a subroutine.

2.8.1 Instructions Using Local Data Buffer

The following three instructions use the local data buffer to provide additional look-ahead capabilities.

FML	Fetch from Memory to Local Data Buffer
FBML	Fetch Block of Words from Memory to Local Data Buffer
FCML	Fetch Characters from Memory to Local Data Buffer

The address fields for these instructions utilize the same rules as the fetches and stores from and to memory, and the optional addressing. When any one of these instructions is executed, the computed address is transferred to the current location in the LDB. The first and second tag bits of the word in this location are then set to 11. This "empty" condition signals the communication unit to fetch the word addressed by the contents of the tagged location and to replace the word, including the tag bits, by the word fetched from the Memory Module. This fetch is executed in parallel while the processor continues to execute the program. If the processor addresses the fetched-to location with an FLS instruction before the empty word has been replaced, the program is delayed until the fetch is completed.

2.8.2 Fetch Single Word to LDB

The FML instruction fetches a single word from memory to the LDB, as described in the preceding paragraph. Although this fetch occurs in parallel with program execution, the normal operation of the advanced station almost always makes this time saving unnecessary. However, when a parent program is storing parameters for a subroutine, it may store a parameter it already has in the stack using the SSL instruction, or it may perform indirect addressing, using the FML instruction, to specify the main memory address of the parameter. Since the subroutine is automatically delayed when it attempts to fetch an indirectly-addressed parameter before it arrives from memory,

the subroutine need not know whether direct or indirect addressing is being used by the parent program. Therefore, a subroutine may be written which assumes that the parameters are always in the local data buffer which is entered by using various mixes of indirect and direct addressing without complicating the subroutine.

2.8.3 Fetch Word Block/Character Stream to LDB

The block-transfer (FBML) and the character-streaming (FCML) may permit considerable time savings in any program which accesses a large number of data words with little computation per data word, a data processing program rather than one which performs scientific computation. The words in Memory Modules are arranged in blocks of four 52-bit words (1 parity, 3 tag, and 48 general bits each). The communication unit, which limits the speed of certain data-processing programs, is tied up an equal amount of time for a single word fetch from memory as for a four-word block fetch. Therefore, four-at-a-time fetching cuts communication time by a factor of four. Four-fetch operations are not used for fetching words from memory directly to the stack because of the programming complexity of having four words at a time entering the central portion of the processor. Block and character transfers to the local data buffer do effectively utilize the four-fetch; the buffering of LDB allows the program to process one word at a time while the communication unit is fetching them four times as fast.

2.8.4 Fetch Word Block to LDB

A four-word memory block is defined as starting with an absolute memory address ending with two ZERO bits. The "BLOCK" pseudo-operation may be used to instruct the assembly program to start a new four-word block. The loading portion of ESP starts each program at the start of a four-word block so that BLOCK pseudo-operation in the assembly program always knows the proper number of no-operation instructions needed to complete the current block.

The BLOCK pseudo-operation generates no-operation (NOP) code to complete the block when preceded by an instruction, and generates ZERO's when preceded by a data-generating pseudo-op.

The number of words involved in FBML is inserted by the program in the stack. The ADVAST waits for the final station to provide this value. The starting address of the block in memory is placed in bits 30-47 of the first-affected word in the local data buffer, as is the address for a single-word fetch. The number of words in the block is specified by bits 42-47 of the top of the stack and inserted in bits 25-29 of the first-affected LDB location. For a single-word fetch, the sign-bit (bit 0) of the "empty" word is set positive. A minus sign designates the start of a block.

Only the first-affected word in LDB is tagged "empty" when the FBML instruction is executed. The "empty" tag is moved down as each word is filled so that the first word yet to be filled is always tagged "empty". This protects the program against premature fetch from LDB to the stack if, and only if, the fetches from LDB are in the same order as the fetches to LDB: the same as the order of words in Memory Modules.

Bit 1 of the first-affected word, the most-significant magnitude bit, is made a ZERO when FBML is executed, and a ONE when FCML is executed. Bit 1 is neither affected nor checked on FML because bit 0 is a ZERO, thus specifying a single-word fetch to LDB.

2.8.5 Character Stream Fetch

When a program initiates a character stream to LDB operation, FCML, it must precede this instruction with the designation in the top of the stack for: the number of characters, the bit position of the start of the first character, and the bit length of a character. These items are arranged in the stack in the following bit positions and transmitted to the first-affected word of the LDB as shown.

<u>Item</u>	<u>Bits in Stack</u>	<u>Bits in LDB Word</u>
Count	42-47	24-29
Start	36-41	18-23
Length	30-35	12-17

The count is six bits and therefore cannot exceed 64, the size of the LDB. If the starting bit position, or the length of a character, is greater than 47, it is interpreted modulo 48. Full-word characters are recognized, and a character may start anywhere in a memory word, and may overlap two consecutive memory words. However, it is not possible with a single FCML instruction to fetch any string of characters which are not arranged consecutively in memory.

Each character, as it arrives in the LDB, is right-oriented in the receiving location; unused bits to the left of the character are cleared to ZERO in the LDB. Thus, the characters are arranged so that, when fetched to the stack, they are all oriented similarly and operations using two consecutive characters as operands are simplified.

2.8.6 Local Pointer

Every fetch to LDB initiated by FML, FBML or FCML goes to the LDB location given by the setting of the "local pointer" (LP) at the time of execution of the initiating instruction. There is a special instruction "set local pointer" (SLP) which allows the program to initiate a series of fetches to the LDB and

to maintain control of the affected LDB locations. Each time FML, FBML or FCML is executed, LP controls the first word or the only word affected in the LDB. LP is incremented by the number of words affected by the instruction: after FML is executed, LP is incremented by 1, after FBML it is incremented by the size of the block, and after FCML it is incremented by the number of characters. If a programmer wishes to place words from memory into the LDB, other than into consecutive LDB locations, he may intersperse successive FML, FBML or FCML instructions with SLP designating the desired LDB destination for each fetch. This operation must be performed carefully to prevent overlapping destination blocks if SLP instructions are interspersed with FBML instructions; there is no way of predicting the result of accidental overlap.

2.9 STACK MANIPULATION

The RTS instruction permits rapid rearrangement of the top few words of the stack in any desired order. This instruction facilitates duplicating the top word so it may be squared, or stored and retained. The top two words may quickly be reversed (in 0.1 microsecond) preceding a non-commutative operation such as subtract, divide, or implication. Certain cyclic permutations for subroutines using a small number of parameters repeatedly are also facilitated by use of the RTS instruction.

2.9.1 Variant Syllable

In practice, the variant syllable of the RTS instruction word is treated as three groups of two bits each, as shown in Figure 2-7.

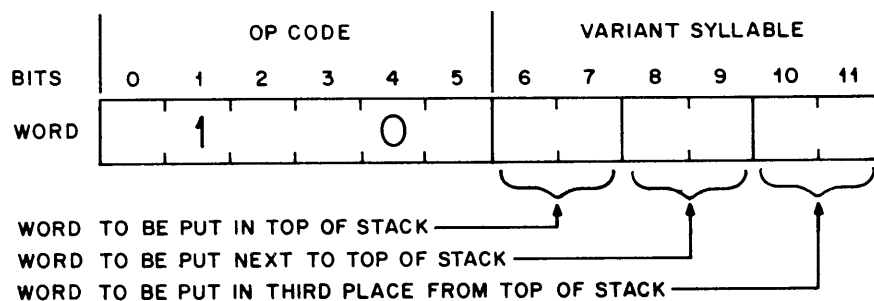


Figure 2-7. RTS Instruction Word

Each two-bit group can be coded as shown in Table 2-5 so that it will contain any one of the four words (A, B, C, or D) initially at the top of the stack. The arrangement of the top eight words of the stack before the RTS instruction is performed is shown in Figure 2-8.

TABLE 2-5
TWO-BIT CODE FOR GROUPS IN RTS VARIANT SYLLABLE

STATE OF BITS IN GROUP	DESIRED WORD FROM STACK ARRANGEMENT BEFORE EXECUTION OF RTS INSTRUCTION
00	A
01	B
10	C
11	D

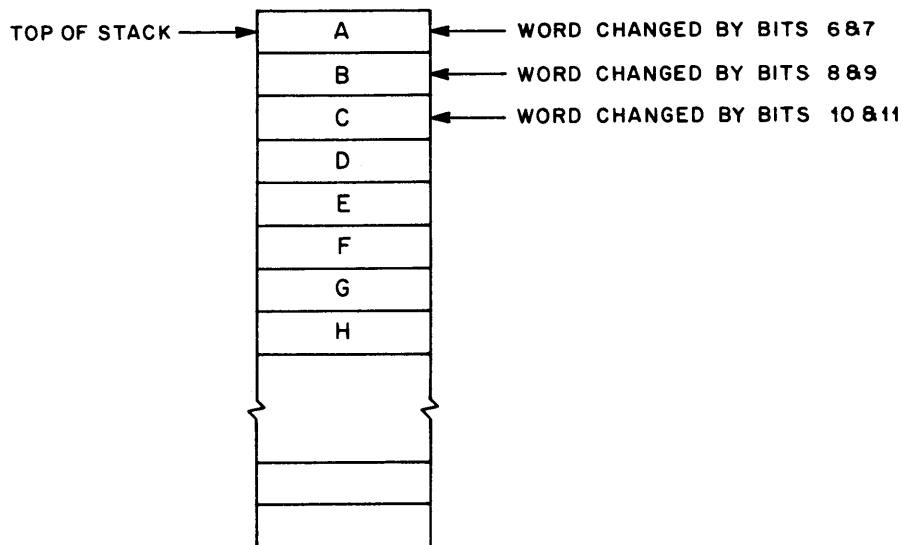


Figure 2-8. Arrangement of Stack Before RTS Instruction

Each of the three groups in the RTS variant syllable denote the word wanted in one of the three top stack positions. Any of the top four words not specified as being placed in one or more of the top three final positions in pushed down into the stack. The relative positions of unspecified words are maintained.

To illustrate the use of the variant syllable, suppose the stack is to be rearranged so that the final position of the top eight words is CABD EFGH. The word to be moved to the top of the stack is originally in position C, so bits 6 and 7 of the RTS variant syllable will be io (a ONE and a ZERO), as indicated by Table 2-5. The word to be second from the top of the stack is A, so bits 8 and 9 are both ZERO's. The word to be put third place from the top of the stack is B, making bits 10 and 11 a ZERO and a ONE respectively. The structure of this RTS CAB word is given in Figure 2-9 which shows that the bits in the variant syllable determine the "41" in the RTS CAP op-code of 1041.

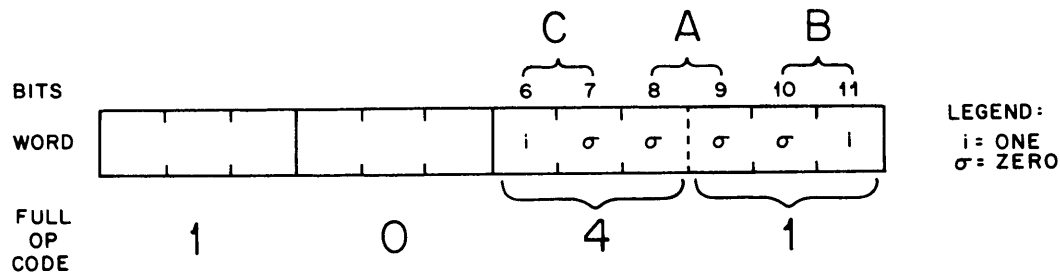


Figure 2-9. Word Structure for RTS CAB Instruction

RTS Execution Times

The ADVAST requires 0.1 microsecond for each RTS instruction. The time required for the FINST depends upon the variant syllable; this time is given in Table 2-6 for all 64 possible RTS instructions resulting from specifying the desired three top stack words.

Two additional stack rearrangements that are considered useful are given in Table 2-7. Each complete rearrangement requires the consecutive execution of two RTS instructions, and therefore requires a total of 0.2 microsecond for ADVST. The total execution time for FINST is given in Table 2-7.

TABLE 2-7.
RTS MACRO-INSTRUCTIONS

MNEMONIC CODE	OP CODE	FINST EXEC. TIME	FINAL ARRANGEMENT OF STACK	OPERATION
DUPD	10 05	0.2	ABAB CDEF	Duplicate top of stack, double precision
	10 04			
QUAD	10 00	0.3	AAAA BCDE	Quadruplicate top of stack
	10 01			

TABLE 2-6

RTS (REARRANGE TOP OF STACK) INSTRUCTIONS

MNEMONIC CODE	OCTAL CODE	FINST EXEC. TIME	FINAL ARRANGEMENT OF STACK	OPERATION
RTS AAA (TRIP)	10 00	0. 2	AAAB CDEF	(Triplicate top word)
RTS AAB (DUP)	10 01	0. 1	AABC DEFG	(Duplicate top word)
RTS AAC	10 02	0. 1	AACB DEFG	
RTS AAD	10 03	0. 3	AADB CEFG	
RTS ABA	10 04	0. 1	ABAC DEFG	
RTS ABB	10 05	0. 1	ABBC DEFG	
RTS ABC	10 06	0. 1	ABCD EFGH	(No rearrangement)
RTS ABD	10 07	0. 2	ABDC EFGH	
RTS ACA	10 10	0. 1	ACAB DEFG	
RTS ACB	10 11	0. 1	ACBD EFGH	
RTS ACC	10 12	0. 1	ACCB DEFG	
RTS ACD	10 13	0. 2	ACDB EFGH	
RTS ADA	10 14	0. 3	ADAB CEFG	
RTS ADB	10 15	0. 2	ADBC EFGH	
RTS ADC	10 16	0. 2	ADCB EFGH	
RTS ADD	10 17	0. 3	ADDB CEFG	
RTS BAA	10 20	0. 1	BAAC DEFG	
RTS BAB	10 21	0. 1	BABC DEFG	
RTS BAC (REV)	10 22	0. 1	BACD EFGH	(Reverse top 2 words)
RTS BAD	10 23	0. 2	BADC EFGH	
RTS BBA	10 24	0. 1	BBAC DEFG	
RTS BBB	10 25	0. 2	BBBA CDEF	
RTS BBC	10 26	0. 1	BBCA DEFG	
RTS BBD	10 27	0. 3	BBDA CEFG	
RTS BCA	10 30	0. 1	BCAD EFGH	
RTS BCB	10 31	0. 1	BCBA DEFG	
RTS BCC	10 32	0. 1	BCCA DEFG	
RTS BCD (CYCU)	10 33	0. 2	BCDA EFGH	(Cycle top 4 words up one)
RTS BDA	10 34	0. 2	BDAC EFGH	
RTS BDB	10 35	0. 3	BDBA CEFG	
RTS BDC	10 36	0. 2	BDCA EFGH	
RTS BDD	10 37	0. 3	BDDA CEFG	

TABLE 2-6 (cont'd)

RTS (REARRANGE TOP OF STACK) INSTRUCTIONS

MNEMONIC CODE	OCTAL CODE	FINST EXEC. TIME	FINAL ARRANGEMENT OF STACK	OPERATIONS
RTS CAA	10 40	0. 1	CAAB DEFG	
RTS CAB	10 41	0. 1	CABD EFGH	
RTS CAC	10 42	0. 1	CACB DEFG	
RTS CAD	10 43	0. 2	CADB EFGH	
RTS CBA	10 44	0. 1	CBAD EFGH	
RTS CBB	10 45	0. 1	CBBA DEFG	
RTS CBC	10 46	0. 1	CBCA DEFG	
RTS CBD	10 47	0. 2	CBDA EFGH	
RTS CCA	10 50	0. 1	CCAB DEFG	
RTS CCB	10 51	0. 1	CCBA DEFG	
RTS CCC	10 52	0. 2	CCCA BDEF	
RTS CCD	10 53	0. 3	CCDA BEFG	
RTS CDA (REVD)	10 54	0. 2	CDAB EFGH	(Reverse top double- precision words)
RTS CDB	10 55	0. 2	CDBA EFGH	
RTS CDC	10 56	0. 3	CDCA BEFG	
RTS CDD	10 57	0. 3	CDDA BEFG	
RTS DAA	10 60	0. 3	DAAB CEFG	
RTS DAB (CYCD)	10 61	0. 2	DABC EFGH	(Cycle top 4 words down one)
RTS DAC	10 62	0. 2	DACB EFGH	
RTS DAD	10 63	0. 3	DADB CEFG	
RTS DBA	10 64	0. 2	DBAC EFGH	
RTS DBB	10 65	0. 3	DBBA CEFG	
RTS DBC	10 66	0. 2	DBCA EFGH	
RTS DBD	10 67	0. 3	DBDA CEFG	
RTS DCA	10 70	0. 2	DCAB EFGH	
RTS DCB	10 71	0. 2	DCBA EFGH	
RTS DCC	10 72	0. 3	DCCA BEFG	
RTS DCD	10 73	0. 3	DCDA BEFG	
RTS DDA	10 74	0. 3	DDAB CEFG	
RTS DDB	10 75	0. 3	DDBA CEFG	
RTS DDC	10 76	0. 3	DDCA BEFG	
RTS DDD	10 77	0. 4	DDDA BCEF	

2.10 TAG BITS

All words in a D851 system are 52 bits long; functionally, these word bits may be grouped: (1) 48 bits for instruction syllables or data sign and magnitude, (2) 1 bit for parity checking, (3) and 3 bits called "tag bits." The 48 bits comprising group (1) above are the only bits shifted, added, multiplied, or otherwise manipulated in the usual data processing or computation techniques. When a bit number is specified in a bit manipulation instruction, only these 48 bits are involved. If a bit number is specified greater than 47, it is treated modulo 48; i. e., the bit number involved is the number specified minus 48. Access to tag bits require special instructions that are discussed in this section.

The parity bit is never accessible by the program. Parity is automatically checked on all transmission accessible by the program. Parity is also automatically checked on all transmission between modules and on many transmissions within modules. When the content of any word is changed, correct parity for the new bit pattern is automatically generated. Except for ESP being automatically entered by an interrupt when a parity error is detected, no program is ever concerned with the parity bit.

The three tag bits are typical of the many software provisions that will be found in the D851 system to enhance system control and assist in such operations as debugging, etc. While tag bits are never referred to by number in a program, it is convenient to discuss them as the first, second, and third tag bits; and for convenience of discussion, they shall be considered to be oriented with the first on the left and the third on the right. Tag bits may be referenced, in assembly language input, either in terms of this assumed orientation or in terms of mnemonic designation of the various tag-bit functions. To simplify the discussion which follows, the function of the third tag bit is explained before the functions of tag bits one and two are taken up.

2.10.1 THIRD TAG BIT

The third tag bit is used for a variety of functions, e. g., (1) for debugging information, (2) for providing essential information during two-computer operation where both modules are working in the same data area, and (3) for information transmission within a single program.

2.10.1.1 Debugging

The third tag bit is used primarily for providing debugging information. It has no control function; its use is fundamentally different from the uses of the second and third tag bits. When a program is being debugged, the instructions and data for the program are loaded into memory with the

third tag bit set to ZERO in each word. Read-out from memory is destructive, with automatic regeneration. The third tag bit is always regenerated as a ONE, regardless of its state when read, and regardless of whether it is being fetched as data or as instructions.

When a program undergoing debugging is being run, it is often valuable for the programmer to know the portions of the program which were executed and those portions which were not. The third tag bit is set to ONE in each instruction word executed, but remains ZERO for every instruction word not executed. The debugging portion of the Utility System, running under ESP control, contains provision for post-mortem examination of a program being debugged with the print-out indicating which sections were executed. Additionally, the print-out indicates which portions of the data area were operated upon.

2.10.1.2 Two-Processor Operation

The third tag bit may also be used if two Computer Modules are operating on the same data area. The first Computer Module to access a Memory Module word leaves it automatically tagged as "used"; the second Computer Module can determine this easily. Since a Memory Module locks out all accesses until the regeneration cycle has been completed, there is no way for the second Computer Module to fetch an already-fetched word before it has been tagged as used.

2.10.1.3 Information Transmission

The third, as well as the other tag bits can also be used for information transmission within a single program. A Computer Module can generate the setting of the tag bit as a ZERO or a ONE and store the word, so tagged, into memory. When a word is stored from a Computer Module to a Memory Module, the Memory Module accepts and stores the tag bits as the Computer Module had set them. When a Computer Module fetches a word from memory, the word is transmitted with the tag bits set as they were in memory, before the regeneration following the fetch.

2.10.2 FIRST AND SECOND TAG BITS

The first and second tag bits are used together in different configurations to provide the four independent control functions discussed below. For convenience, the configurations are referred to as: "00X", "01X", "10X", and "11X"; their mnemonic coding is given in the text. The "X" in the configurations represents the third tag bit.

2.10.2.1 No-Operation ("00X")

The "00X" configuration denotes the absence of control information, and is a tag bit no-operation. The address-field coding may be either "00X" or the mnemonic "NOP."

2.10.2.2 Empty ("11X")

The configuration coded as "11X" or "EMPTY" is ignored by both the advanced and final stations. Its control function is only effective in the local data buffer where it causes block or single transfers in parallel with other processing. This configuration is automatically generated by the instructions initiating fetches to the local data buffer. When the EMPTY configuration is generated in the stack and the tagged word is stored in the local data buffer, it causes a fetch to the local data buffer. When an "empty" word is stored in a Memory Module, it allows a type of indirect addressing via the local data buffer. This is explained in the sections dealing with indirect addressing and the local data buffer.

2.10.2.3 Interrupt ("01X")

The configuration coded as "01X" or "INT" causes an interrupt when a word so-tagged is fetched to the top of the stack or when execution of the word as an instruction is started. Since an interrupt always enters ESP, the interrupt tag bit configuration is mostly used by ESP and must always be used with ESP cooperation. This configuration is used, among other things, for snapshot dumps by tagging instructions; it is also used for many of the control functions associated with snag-bit interrupts. For instance, as a snag-bit interrupt, it functions as a response to the presence of critical data, and is used to facilitate computer control of program areas, e. g., to lock out a second Computer Module from accessing a data area already accessed by another Computer Module. This configuration also provides a convenient way of marking the bottom of the stack so that an undebugged program is interrupted if it attempts to store more from the stack than has been placed in it.

2.10.2.4 Jump ("10X")

The configuration coded as "10X" or "JUMP" causes a jump to occur when a word so-tagged arrives at the top of the stack as data. It is never effective in words fetched as instructions. The conditions under which a jump tag is effective are precisely the same as those under which interrupt tagging of data is effective.

2.10.3 TAG-BIT INSTRUCTIONS

Two instructions are provided for setting tag bits and, after they have been set, for testing them ; these instructions are briefly described below.

2.10.3.1 STB Instruction

The instruction which sets tag bits in the Computer Module is, mnemonically, STB (Set Tag Bit). The address field contains information on the tag bit(s) involved and the setting desired. The tag bits may be designated by their assumed orientation or their function. Thus, the first two of the following examples set the third tag bit to ZERO (unused) and the last two, to ONE (used).

STB	XX0
STB	UNUSED
STB	XX1
STB	USED

In the first and third of the above examples, the X's denote that the first and second tag bits are unaffected by the operation. When the operation is denoted mnemonically by function, the third tag bit is also the only one affected.

2.10.3.2 JTB Instruction

The testing of the tag bits is accomplished in a manner equivalent to that used for setting them. The mnemonic operation is JTB (Jump on Tag Bits), and the address field contents are identical to those used for STB. That is, "XX0" means "test the third bit for ZERO, ignore the first and second"; "USED" means "test for the used tag-bit indication", and so on.

2.10.4 CONDITIONS AFFECTING TAG-BIT INTERRUPT AND JUMP FUNCTIONS

This discusses the major considerations and conditions for effective tag-bit interrupt and jump operations. Essentially, the subject matter is approached from the standpoints of two basic situations, i. e., (1) situations where the tag bits exist and (2) situations where tag bits do not exist, and are to be inserted. In general, D851 software design provides four ways by which tag-bit interrupt and jump functions can be simply and effectively performed to meet the needs of any situation and to derive full value from the tag-bit feature; these ways are discussed in some detail. As is often the case in complex systems, there is a "right" as well as a "wrong" way, and since the "wrong" way is not already immediately apparent to the programmers unfamiliar with the D851, - in fact, it may initially appear to be feasible, - this discussion of the ways to effect the tag-bit function also includes an example of the "wrong" way.

In the D851, "The Easiest Way is the Best and the Correct Way." The first point to note concerning D851 tag-bit interrupts and jumps, is that both are inoperable in the control mode, and can be made inoperable in the normal mode through ESP setting of the mask register. This latter feature permits a routine to examine tag-bit settings without being interrupted. However, a normal mode program, which has not been properly set via the mask register to ignore tags bits, would be interrupted before it had an opportunity to change the "INT" to "NØP", or before it had an opportunity to execute a jump on the state of interrupt tag bits (JTB).

A second point to note is that the execution of a tag-bit interrupt or jump function by a tagged data word occurs only when the data word is at the top of the stack. As previously indicated, there are four ways to ensure correct execution of the tag-bit interrupt or jump, where the tagged data word is properly at the top of the stack, and further more will meet the needs in situations where tag bits exist or where they do not exist. These four ways, described in detail later, may be summed up as follows:

- (1) Generating a new word with a tag bit (Case 1)
- (2) Tag-bit insertion into word already in the memory (Case 2)
- (3) Moving a tagged word up through stack to top (Case 3)
... by stepping stack, or storage of word ahead of
it, or by using word ahead as an operand
- (4) Fetching a tagged word from storage to top of stack (Case 4)

To describe the conditions of execution of a tag-bit function by a tagged data word, it is necessary to consider each case separately.

2.10.4.1 Case 1

When the STB instruction is executed, the effect of the tag-bit function set-up is locked out until the tagged word has been removed from the top of the stack. The tagged word, for example, can be stored in memory and the tag-bit function has no effect until the word is fetched back. Some care must be exercised by a programmer setting up a tag-bit interrupt if he wishes to prevent the interrupt from being effective until later. Assume, for example, that the programmer wishes to store a word containing zero except for the tag-bit interrupt. Assume further that he wishes to set up the zero by a literal (LIT0), and to store the tagged word in LØCX in memory by a Store from Stack to Memory instruction (SSM LØCX). The following coding would accomplish the desired result. (This is the correct and recommended procedure).

```

LIT      0
STB      INT
SSM      LØCX

```

The above coding introduces the tag bits into a word already at the top of the stack and the interrupt is locked out until after the tagged word is stored.

The following coding, on the other hand, would cause an immediate interrupt.

```

STB      INT
LIT      0
AND
SSM      LØCX

```

While the tag bits in the above situation are inserted into a word already at the top of the stack, the lock-out, however, is in effect only until the literal zero is generated, which pushes the tagged word down into the stack. The AND instruction brings the already-tagged word out of the lower position in the stack, the lock-out is no longer in effect, and an immediate interrupt therefore occurs.

2.10.4.2 Case 2

Consider now coding to insert interrupt tag bits into a word already in memory.

```

FMS      LØCX
STB      INT
SSM      LØCX

```

The above coding accomplishes the desired result and is the correct procedure; no interrupt occurs when the tag bits are inserted and the tagged word is stored into memory. The following coding would also insert interrupt tag bits into a word already in memory but would cause an interrupt before the word is returned to memory.

```

LIT      0
STB      INT
ADD      LØCX
SSM      LØCX

```

The problem in the foregoing coding is that "ADD LOCX" assembles as a fetch followed by an add. The fetch forces the tagged zero down into the stack, cancelling the lock-out, with the result that add now causes an interrupt.

When inserting a tag bit into a word already in memory, the programmer must keep the following firmly in mind:

"When two tagged words are to be used as operands and combined in a logical or arithmetic instruction, the tag bits of the result will be the tag bits of one of the operands. If the case is where both tagged words are in the stack, the tag bits of the deeper word will be retained; the tag bits of the word ahead will be lost. If the situation involves one tagged word in memory and the other in the stack, the tag bits of the word in the stack will be retained, and the tag bits of the newly introduced word will be lost"

With the foregoing clearly understood, the programmer should have no difficulty when coding, and situations involving the possible loss of a desired tag function will be avoided. For a more concrete example covering the foregoing situation, consider the following coding

FMS	LØCX
LIT	0
STB	INT
ADD	
SSM	LØCX

In the above example there would be no immediate interrupt, and no storage of a tagged word in memory. The interrupt is locked out, when the STB is executed, until after the ADD is executed. But the result of the ADD is tagged, as was LØCX, and the newly introduced interrupt tag bits are lost.

Since tag bits are never involved in ordinary logical or arithmetic instructions, any two-operand instruction is non-commutative as far as tag bits are concerned. The rule for tag bits is the same as that for all operations in which some bits are not involved; i. e., bits not involved in the operation are the same in the result as they were in the operand originally in the stack or in the operand deeper in the stack, whichever is the applicable situation.

2.10.4.3 Case 3

When a tagged word is brought to the top of the stack from a position lower in the stack, the tag-bit function is executed only when the tagged word is used as an operand, not when it arrives at the top of the stack. The word "operand" as used here includes any instruction which alters the word, uses it to generate a result, tests it, compares it with another word or stores it

from the stack to any other place in any module. However, a word tagged for interrupt or jump may be pushed down from the top of the stack to a deeper position in the stack without executing the interrupt or jump. When tag-bit interrupts are used to mark the bottom of the stack, to guard against stack underflow in an undebugged program, the stack may become empty without an interrupt. Only when the program attempts to make use of the words below the bottom of the stack, - words which have no meaning to the program, - is the program interrupted.

2.10.4.4 Case 4

When a tagged word is fetched from a position, not in the stack, to the top of the stack, the tag-bit function is executed immediately after the word arrives at the top of the stack. To interpret this case it is necessary to differentiate between macro-instructions and individual machine-language instructions. For example, the following coding is a macro-instruction:

```
ADD    LØCX
```

The equivalent coding in individual machine-language instructions which are generated by the assembly program is as follows:

```
FMS    LØCX  
ADD
```

If the word in LØCX is tagged for interrupt, the interrupt occurs immediately after the fetch from memory to stack (FMS), before the add. Thus, adding a tagged word from memory results in interrupt with the fetched word at the top of the stack with the operand originally in the stack pushed one position deeper, and without addition having been performed.

2.10.5 JUMP TAG AND INTERRUPT TAG OPERATION

It has been previously explained: (1) that the tag-bit configuration "10X" causes a jump to occur when a word so tagged arrives at the top of the stack as data, (2) that it is never effective in words fetched as instructions, and (3) that conditions for effective jump-tagging are the same as for interrupt-tagging of data.

Interrupt-tagging is effective regardless of the contents of the non-tag 48 bits in the tagged word and generates the same interrupt and the same entry to ESP independent of the non-tag bits. Jump-tagging, on the other hand, requires the non-tag bits to contain jump-control information. When a jump-tagged word causes a jump, the contents of the word are transferred to the jump control register (JCR) and an unconditional jump generated. The bit position of information in a jump-tagged word is as follows:

BITS	CONTENTS
30-47	Absolute address of jump destination word

The absolute address of the jump-destination word may be obtained, regardless of the addressing mode being used in the program, by the "Fetch Absolute Address to Stack" instruction (FAS), which computes the absolute address required and fetches this address to bits 30-47 (right-oriented) of the top of the stack after pushing the stack down one to make room and clearing bits 0-29. Thus, the following coding will place a jump-tagged word in the top of the stack.

FAS	LØCX
STB	JUMP

The above coding generates a jump to the first syllable (syllable number 0) of LØCX, which in the above example is referenced in self-relative addressing. Since it would always be clumsy for a program to determine the syllable number of a destination other than the start of a word, a "WØRD" pseudo-operation must be used to force the destination syllable to the beginning of a word.

Jump-tagging allows termination of a many-iteration loop to occur automatically when a terminating word is fetched from memory and without termination testing being required in every iteration. It also permits a programmer to use a special routine to process certain data words without repeated programmed testing to see if the word being processed requires some sort of special handling. Although there are expected to be many uses found for jump-tagging, there are several cautions to be observed in its use.

First, a jump-tagged word in memory contains an absolute address. If the program is interrupted by a higher-priority program and is relocated before restarting, ESP must be informed that jump tagging is being used so that the absolute addresses may be changed before restart. However, ESP can easily find the jump-tagged words by searching through memory with the "JTBJUMP" instruction. In fact, the only reason that ESP need be informed is that the time required for the search is wasted if there are no jump tags being used. If a particular D851 installation is expected to use jump-tagging at all frequently, the ESP for that installation will always search for jump-tagging before relocating an interrupted program.

A tag bit interrupt or jump is recognized by the final station when a data word is tagged. In general, the advanced station will have fully or partially executed a number of instructions past the instruction which fetched the tagged data word. Thus, index registers may have been incremented or refilled and a program must be prepared to either restore the index settings in effect at the time of the fetch or ignore the indexes which may have been affected. Obviously, if jump tagging is used to determine termination of an iterative procedure, the indexes being incremented during the iteration may be of no further interest after termination.

In the case of a tag bit interrupt, ESP contains a routine which restores all indexes, except those refilled (XRL), to their values at the time of the fetch of the tagged data word. This routine is available as a systems macro for any program using tag bit jumps by the word "RESTØRE" in the op field.

A tag-bit interrupt inserted in an instruction word does not cause any problems since the interrupt is recognized by the advanced station which stops immediately and waits for the final station to empty the final-processing queue. Thus, the effect of the interrupt is as though the advanced and final stations were interrupted together.

2.11 ASSEMBLY ERROR INDICATIONS

All input cards to the assembly program are printed with sequential numbers assigned by the assembly program. When any input card contains coding that violates any of the rules given in this manual, the violation is indicated in the left margin of the assembly print-out, alongside the print-out of the card. Furthermore, a separate list of errors is printed, giving the card numbers and descriptions of the errors found.

An attempt will be made to interpret all erroneous coding in terms of what the programmer intended. For example, an undefined symbol results in the definition of the symbol, by the assembly program, as the location of an otherwise unused word in the program's data area. However, there is no way for the assembly program to determine if two undefined symbols were intended to be the same, with one mis-spelled or mis-punched. It is recommended that a programmer adhere to the rules, rather than deliberately violate the rules to utilize his knowledge of the actions taken when rules are violated. If the assembly-program error list contains a large number of indications of deliberate rule violations, some of the accidental or real errors may escape the programmer.

There are no errors which can cause the assembly program to refuse to assemble the remainder of the program. Thus, a programmer need not correct an error if his first debugging runs do not involve execution of the erroneous portion. Corrections are also unnecessary if the action of the assembly program happens to be what the programmer intended by his erroneous coding. However, it is strongly recommended that a program be corrected to eliminate all apparent errors at the time essential changes are made. This technique prevents the error output of the assembly program from becoming confusingly voluminous.

2.12 PSEUDO-OPERATIONS

The "EQU" and the "BØØL" pseudo-operations are discussed below.

2.12.1 The "EQU" Pseudo-Operation

The mnemonic "EQU" in the op-code field of assembly input is meaningful only when a location symbol is specified in the symbol field. This pseudo-operation defines the location symbol as equal to whatever is designated in the address field. However, to accommodate the special requirements of addressing-mode specification, the coding in the address field is recorded as it is found, rather than converted to binary at the point at which the assembly program encounters the "EQU" pseudo-op. The effect of definition via "EQU" is to replace a composite address by a single mnemonic symbol.

In addition to permitting addressing modes to be specified when a symbol is defined, indexing of a symbolic location may also be specified at the time of definition. If indexing is indicated both when the symbol is defined and when it is referenced, the result is multiple indexing. In the following example, LØCX is indexed for the third time when reference is made to LØCY.

```
LØCY EQU LØCX/1/2
      ØP LØCY/3
```

Since multiple indexing by a single index register is permitted, the following coding would result in triple indexing by index register 1.

```
LØCY EQU LØCX/1
LØCZ EQU LØCY/1
      ØP LØCZ/1
```

The assembly program properly sorts the information inserted in an address field by reference to symbols defined elsewhere, and by direct specification. As an example, consider the following:

```
LØCY EQU LØCX 1, A
      ØP LØCY + 5
```

The above coding has the same effect as the following:

```
ØP LØCX + 5/1, A
```

Symbolic reference to index registers is permitted, and these symbols may be defined by "EQU" pseudo-operations. However, if an addressing mode is specified in the address field of an "EQU" card, the symbol may not be used as an index specification.

2.12.2 The "BØØL" Pseudo-Operation

This pseudo-operation is used to define an octal constant, which then may be used anywhere else in the program. In this case, as opposed to "EQU", a constant numeric is computed when the assembly program encounters the "BØØL" pseudo-operation. All numeric constants in the address field of a "BØØL" are interpreted as octal integers. The operators are interpreted as boolean rather than arithmetic, according to the following rules:

- + inclusive OR
- * AND
- Exclusive OR
- / Complement

2.13 MEMORY-BOUNDS REGISTER

The memory-bounds register specifies the maximum and minimum main memory addresses a program is permitted to change. The bounds are ignored in control mode, and ignored in normal mode when the mask register bit controlling memory-bounds is a ZERO. When a program is set to observe memory bounds, and when a store, dump or Memory Module program is initiated by the program, the memory bounds are automatically checked to determine if any attempt is being made to access any address outside of the memory bounds; if so the program is interrupted and processed under ESP control.

The primary purpose of memory bounds is to protect the operating system (ESP) and the associated debugging aids against inadvertent change by an undebugged program. The bounds may also be used to protect an operational program against an undebugged program when they are running simultaneously on two processors, or when the debugging run starts before all the output from a previous production run has been removed from memory. Furthermore, the interrupt on memory-bound violation provides useful debugging information if the bounds are clamped tightly around the area a programmer expects to write into. ESP contains provisions for memory-bound modification upon normal-program request, as long as the requested bounds do not violate the total allowed area imposed by ESP. Therefore, a program which uses different data areas at different times may be partially debugged with the assurance that any violation of the area currently in use is promptly detected and reported. It is desirable to keep the instruction area of a program being debugged outside of the program memory bounds.

The memory-bounds register, mnemonically designated "MBR", consists of 36 bits which correspond to bits 12-47 of the top of the stack. The register may be examined by any program at any time by using the following coding.

FRS MBR

The memory-bounds register may be set by the following coding. (If an attempt is made to change the MBR in normal mode, when set to interrupt on bound-violation, an interrupt is generated.)

SSR MBR

The most significant 18 bits designate the lower bound, and the least significant 18 bits designate the upper bound.

2.14 NON-COMMUTATIVE OPERATIONS

There are five instructions which are non-commutative:

SUB	Subtract
DIV	Divide
IMP	Implication
JGR	Jump on Result of Test for Greater
JLS	Jump on Result of Test for Less

Also, all field-defined operations on two operands are non-commutative since the non-field bits of the result are set to equal the non-field bits of one of the operands.

In all five non-commutative instructions, the operand normally written to the left in algebraic expressions is the operand deeper in the stack when both operands are in the stack, or the operand in the stack if one operand is in memory. As an example, "A-B" becomes "AB-" in Polish notation and is coded as follows.

FMS	A
FMS	B
SUB	

or:

FMS	A
SUB	B

Similarly, the test for "A>B" becomes "AB>" and is coded as follows.

```
FMS    A
FMS    B
JGR
```

or:

```
FMS    A
JGR    B
```

The convention for field definition is that the non-field bits of the result of a two-operand operation are always equal to the non-field bits of the operand deeper in the stack, when both operands are in the stack, or to the non-field bits of the operand in the stack when one operand is in memory.

2.15 STORAGE QUEUE

Three separate operations implement a Store from Stack to Memory instruction (SSM). First, the advanced station computes the store address and records it. Second, when the instruction reaches the final station the word at the top of the stack is stored in the storage queue, and flags are set to tell the communication unit to store the information from the storage queue into the absolute Memory Module address recorded by the advanced station. Finally, the communication unit stores the information from the storage queue into the Memory Module.

These three operations are always performed sequentially, but asynchronously. In particular, the communication unit stores from storage queue to a Memory Module as its lowest priority operation. The storage queue accommodates a maximum of eight words destined for Memory Modules: this capacity insures that there would rarely be delay of the final station because of a full storage queue.

Whenever the advanced station computes a Memory Module address for a store or a fetch operation, this address is compared with the following addresses: (1) words already awaiting Memory Module storage in the storage queue, (2) the addresses computed for words to be placed in the storage queue by SSMs not yet executed by the final station, and (3) the addresses in associative memory of index words held in the Computer Module. If any of these addresses is identical to the newly computed address, the designated operation is automatically modified so the program fetches the newest version of any word, whether in a Memory Module or elsewhere in the processor. The following sequence of FORTRAN statements demonstrate address updating.

FORTRAN code	X = Y*Z	
	W = X	
Machine code	FMS	Y
	FMS	Z
	MUL	
	SSM	X
	FMS	X
	SSM	W

In the above coding, it is certain that the advanced station will have executed the "FMS X" before the final station had executed the preceding "SSM X". Although the compiler could have replaced the SSM - FMS sequence with a DUP (duplicate) instruction placed before the SSM, this would have saved no time and would be incorrect if the second source-language statement were numbered and possibly a jump destination. The automatic comparison between a newly computed address and previously computed addresses guarantees that the fetch accesses the word just stored, not the word the store replaces in memory.

2.16 TIMING

A D851 Processor Module executes programs with several of its units operating in parallel, but asynchronously. Four of these units directly affect the execution time of a program; these units are the communication unit (COMM), the syllable determination unit (SYLD), the advance station (ADVAST), and the final station (FINST).

2.16.1 Determining Program Execution Time

To determine precise timing, it may be necessary to consider the timing of each unit individually. However, most programs tend to have their timing dominated by the execution time of one unit. Therefore a good approximation to the timing of a program may be obtained by considering only the dominant unit.

In "scientific" programs and others requiring a lot of computations, the final station (FINST) is almost always the dominant unit, because of the arithmetic performed at this station. Therefore program execution time may be closely approximated in most cases by merely adding the FINST times given in Table 2-8, for each instruction that is executed. Times for the ADVAST and COMM are also in the table, to allow calculation of precise timing. (Timing requirements for SYLD are given below.)

TABLE 2-8

TIMING FOR TYPICAL INSTRUCTIONS

NAME	μ SEC REQUIRED IF NO DELAYS OCCUR			TYPES OF DELAYS POSSIBLE	REMARKS
	ADVAST	COMM	FINST		
ADD	0.1		0.4	1, 2, 3	Add top two words of stack. Timing assumes single-precision floating-point with normalized result.
DUP	0.1		0.1	1, 2, 3	Duplicate top word of stack.
FLT	0.1		0.1	1, 2, 3	Enter floating-point computational mode.
FMS	0.2	(0.5)	0.1	1, 2, 3, 4, 5	Fetch from memory to stack. COMM is not involved if addressed word is already in processor or already being fetched or stored within processor.
FRS	0.2		0.1	1, 2, 3	Fetch from register to stack.
INST		0.5		6	Fetch a four-word block of instructions. Automatically executed when ADVAST has started execution of previous block or when a new jump address is given, except that instructions already in processor are not fetched.
JUMP	0.1		0.1	1, 2, 3	Unconditionally jump to address previously set up.
JXELS \emptyset	0.3		(0.1)	1, 2, 3	Test content of last-used index for equality to self-contained limit. Subtract one from content after testing. Jump to previously set-up address if test result agrees with true/false indication set-up. FINST is involved and delay-types 2 and 3 are possible only if jump occurs.

*Delays are listed in Table 2-9.

TABLE 2-8 (cont'd)

TIMING FOR TYPICAL INSTRUCTIONS

NAME	μ SEC REQUIRED IF NO DELAYS OCCUR			TYPES OF DELAYS POSSIBLE	REMARKS
	ADVAST	COMM	FINST		
LIT	0.1		0.1	1, 2, 3	Transmit full-word literal from program string to top of stack.
MUL	0.1		0.4 to 1.0	1, 2, 3	Multiply top two words of stack. Timing assumes single-precision floating-point with normalized result.
NØP	0.1			1	No operation.
SJF	0.2			1	Set up jump address if next test gives a "false" result or for an unconditional jump. INST may follow if the addressed instructions are not already in processor. Instruction words fetched in response to SJF are retained until the next set-up jump instruction.
SLIT	0.1		0.1	1, 2, 3	Transmit short (six-bit) literal from program string to top of stack.
SQM		0.5		6	Store from storage queue to memory. Automatically executed when a word has been placed in the storage queue by ADVAST or FINST.
SSM	0.2		0.1	1, 2, 3, 7	Store from stack to memory by placing word in storage-queue for a subsequent SQM to transmit to memory. In the timing summaries, the time for SQM is included with that for SSM. In step-by-step timing, the two are considered separately.

TABLE 2-8 (cont'd)

TIMING FOR TYPICAL INSTRUCTIONS

NAME	μ SEC REQUIRED			TYPES OF DELAYS POSSIBLE	REMARKS
	IF NO DELAYS OCCUR	ADVAST	COMM FINST		
SSM(X)	0.2		0.1	1, 2, 3, 7	Store from stack to index word in memory. Storage is actually to associative memory in local storage. A word is displaced from associative memory and sent to main memory if the index word addressed by SSM(X) is not already in associative memory. In this case only will SQM follow and is delay-type 7 possible.
SSR	0.2		0.1	1, 2, 3	Store from stack to register.
SUB	0.1		0.4	1, 2, 3	Subtract top word of stack from stack word. Timing assumes single-precision floating-point with normalized result.
TRIP	0.2		0.1	1, 2, 3	Triplicate top word of stack.
X	0.2	(0.5)		1, 7, 8	Modify next address by adding contents of specified index word to address register. Alternately used to specify index to be operated upon by following instruction. COMM is not involved if the specified index is already in associative memory. Delay-types 7 and 8 are possible only if the word is not already in associative memory.
XA \emptyset	0.2			1	Add one to index word most recently specified.
XM	0.3	(0.5)		1, 7, 8	Perform operation "X". Then modify index word by adding self-contained signed increment to content.

Occasionally the FINST may be delayed by having to wait to receive or store a word. In this event, one of the other units temporarily becomes the dominant unit. To understand how this might occur, a general description of the way instruction and data words are handled is given below. (Delays and their causes are listed in Table 2-9.)

2.16.2 Instruction Words

In the execution of a program, instruction words (and data words) are fetched and stored by COMM, which automatically keeps ahead of SYLD on instruction fetches.

Each instruction is first examined by SYLD, which groups the syllables into discrete instructions for further processing. SYLD requires 0.1 microsecond for any instruction which is entirely contained within one instruction word, and 0.2 microsecond for any instruction overlapping two instruction words. The time required by SYLD may almost always be ignored, and is not mentioned for individual instructions. However, it should be noted that SYLD timing for macro-instructions depends upon the number of actual machine instructions contained within a macro-instruction.

The discrete instructions composed by SYLD are transferred to the ADVAST. Some of these instructions are completely processed by the ADVAST; such processing includes address computation, with indexing and indirect addressing, and the initiation of data fetches.

The remaining instructions are sent to the FINQ for eventual execution by the FINST. The FINQ temporarily stores instructions until the FINST can execute them. The FINQ can hold eight instruction words, which is ordinarily sufficient to keep the FINST operating without being held up waiting for an instruction word. When the FINQ is full, the ADVAST is prevented from sending any more instructions to the FINQ until the FINST can execute one or more of the stored instructions. The FINST performs all operations involving the data stack, including all arithmetic operations on data.

2.16.3 Data Words

The COMM fetches all data for the Computer Module, in response to the ADVAST computation of absolute memory address. If either of these stations requires a word which has not yet been fetched, the station requiring the word is delayed until the fetch has been completed.

When a word is to be stored from the Computer Module to a Memory Module, the word is placed in an eight-word storage queue, together with the absolute memory address it is destined for. The communication unit sends words for the storage queue to Memory Modules in the same order in which they are put into the storage queue. If a station attempts to store a word in the storage queue when the queue is full, the station is delayed and cannot

TABLE 2-9

TYPES OF DELAYS

DELAY	ABBREVIATION	EXPLANATION
1	ADVAST delayed by INST	The advanced station may be delayed in its execution of any instruction if the communication unit has not yet fetched all syllables of the instruction.
2	ADVAST delayed by FINST	Any instruction requiring the final station may delay the advanced station if the eight-instruction queue waiting final processing is full. This delay is considered normal since most programs are limited by the speed of final processing. This delay, in fact, serves to prevent the advanced station from running too far ahead of the final station.
3	FINST delayed by ADVAST	The final station may be delayed if it is ready for an instruction before the advanced station has passed it into the final processing queue, if the queue has become empty. In most programs this should only occur when starting before the advanced station has had an opportunity to get its normal distance ahead of the final station.
4	ADVAST delayed by COMM	When the advanced station has computed a fetch address but the communication unit is not ready to accept it, the advanced station waits until communications is ready to start the fetch. Data-fetching is the highest priority operation for the communication unit.
5	FINST delayed by COMM	The final station cannot accept a fetched word to the top of the stack until the communication unit has finished fetching the word. This delay is actually caused by the advanced station starting the fetch too late and is, therefore, similar to delay type 3.

TABLE 2-9 (cont'd)

TYPES OF DELAYS

DELAY	ABBREVIATION	EXPLANATION
6		The communication unit operates in response to operations previously performed by the final or advanced stations. Since COMM always waits for some other unit to operate, no delays of COMM are considered. If a program is limited by communication speed, this will show up as delays of other stations by COMM.
7	ADVAST delayed by STQ	The storage queue has an eight-word capacity. When a word is to be placed in a full queue by a SSM instruction, the advanced station will wait until the communication unit has made room by executing SQM. This delay may also occur when a word is displaced from associative memory but the storage queue is full.
8	ADVAST delayed by ASM	If an index word is specified which is not already in associative memory, the advanced station is delayed until the word has been fetched from a Memory Module. Each such fetch requires the storage of a word from associative memory to the storage queue. Hence, a later SQM is required and delay-type 7 may occur.

continue until the COMM makes room by executing a store from the storage queue to a Memory Module.

SECTION 3

INSTRUCTION REPERTOIRE

The instructions described in this section are those recognized by the assembly program. For convenience and clarity of presentation, these instructions have been categorized as follows:

- Arithmetic
- Logical
- Shift
- Stack Manipulation
- Literal
- Bit Manipulation
- Jump
- Index
- Indirect Addressing
- Fetch and Store
- Control
- Pseudo Operations

The above categories of instructions are further broken down on the following pages to show the individual instructions comprising each category and the order in which they will be discussed throughout the remainder of this section.

It should be noted that Pseudo Operations will be expanded to include such operations as EQU and END.

LIST OF INSTRUCTIONS

Arithmetic	Logical (cont'd)
ADD	B1101
SUB	B1110
MUL	B1111
DIV	
FLR	
SQR	
RND	Bit Manipulation
	SB
	CLB
	CHB
Shift	INB
	PØS
SR	NEG
SL	CHS
SAC, NORM	INS
RR	STB
RL	
JUS	
UNJ	
SSH	Literal
	LIT
	SLIT
Logical	
AND, SCL, LMP, B0001	
ØR, SST, LAD, B0111	Fetch and Store
ØRX, SCM, HAD, BBC, B0110	
FILL	FMS
CLR	FML
CØM, NØT	FBML
IMP, B0010	FCML
EXT	FLS
INF	SSL
B0000	SLP
B0011	SSM
B0100	FAS
B0101	DLM
B1000	LML
B1001	DRM
B1010	LMR
B1011	FRS
B1100	SSR

LIST OF INSTRUCTIONS

Indirect Addressing

AMA
FMA
FLA

Stack Manipulation

RTS
DUP
DUPD
TRIP
QUAD
REV
REVD
CYCU
CYCD
STEP

Index

X
XA
XM
XMA
XS

Jump

SJF
SJT
SJSF
SJST
JTB
JXEZ
JXEL
JXES
JXGL
JXGS
JXLL
JXLS
JX__RS
JX__AØ

Jump (cont'd)

JX__AI
JX__AS
JX__SØ
JX__RL
JX__SS
JCB
JGR
JGA
JLS
JLA
JEQ
JZE
JFS
JBT
JSI
JUMP
RET
RETI

Control

STØP
NØP
ENM
ESP
SFCN
CCM
IØ
MEM
IIØ
CCB
INT
SPP
FLT
SIP
FIX
ALPH
LØG

Pseudo

BLØCK
WØRD

3.1 ARITHMETIC INSTRUCTIONS

The operation of all arithmetic instructions, except FLR, is affected by the current computational mode. The formats of operands are given below for each mode.

Fixed-Point - Single-Precision

Bit 0	Algebraic Sign
Bits 1 - 47	Magnitude (binary point is left of bit 1)

Fixed-Point - Double-Precision

First Word:

Bit 0	Algebraic Sign
Bits 1 - 47	Magnitude (binary point is to left of bit 1)

Second Word:

Bit 0	Ignored
Bits 1 - 47	Continuation of Magnitude

Floating-Point - Single-Precision

Bit 0	Algebraic Sign
Bits 1 - 12	Exponent
Bits 13 - 47	Mantissa

Floating-Point - Double-Precision

First Word:

Bits 0	Algebraic Sign
Bits 1 - 12	Exponent
Bits 13 - 47	Mantissa

Second Word:

Bit 0	Ignored
Bits 1 - 47	Continuation of Mantissa

Integer – Single-Precision

Bit 0	Algebraic
Bits 1 - 12	ZERO
Bits 13 - 47	Integer Magnitude (binary point is to right of bit 47)

Integer – Double-Precision

First Word:

Bit 0	Algebraic Sign
Bits 1 - 12	ZERO
Bits 13 - 47	Magnitude

Second Word:

Bits 0	Ignored
Bits 1 - 47	Continuation of Integer Magnitude (binary point is to right of bit 47)

Specified-Point

Same format as floating point (see above), but the exponent of the result of all computations is adjusted to agree with the exponent in the specified-point register.

Significant-Point

Same format as for floating point (see above), except that the exponent of the result is adjusted to agree with the exponent of the operand which contains the larger number of leading zeros in the mantissa.

Alphameric – Single-Precision

Bits 0 - 47	Magnitude
-------------	-----------

Alphameric – Double-Precision

First Word:

Bits 0 - 47	Magnitude
-------------	-----------

Second Word:

Bits 0 - 47	Magnitude
-------------	-----------

NOTE: Alphameric mode ignores overflow; hence, it may be used if the operand is in unsigned two's complement form.

Logical

Same format as for alphameric mode (see above).

NOTE: Logical mode ignores overflow; hence, it may be used if the operand is in unsigned one's complement form (end-around carry is propagated).

ADD ADD

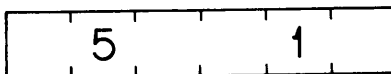


OPERATION. Add two operands obtained from stack, or add one operand from stack and one from memory. Place result in stack. Make all zero results positive.

POSSIBLE CR INDICATIONS. Overflow may occur, and can be detected in all modes except alphameric and logical. Underflow may occur and can be detected in integer, floating-point, specified-point, significant-point modes.

.....

SUB SUBTRACT



OPERATION. Subtract operand in top of stack from stack operand beneath it, or subtract operand in memory from operand in top of stack. Place result in stack. Make all zero results positive.

POSSIBLE CR INDICATIONS. Overflow may occur, and can be detected in all modes except alphameric and logical. Underflow may occur and can be detected in integer, floating-point, specified-point, and significant-point modes.

MUL MULTIPLY



OPERATION. Multiply two operands obtained from stack, or multiply one operand from stack and one from memory. Place most significant half of result in stack and the least significant half in discard register. Derive sign of result algebraically from signs of operands.

POSSIBLE CR INDICATIONS. Overflow may occur, and can be detected in the integer, floating-point, specified-point, and significant-point modes. Underflow may occur and can be detected in integer, floating-point, specified-point, and significant-point modes.

.....

DIV DIVIDE

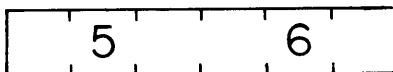


OPERATION. Divide operand deeper in stack by operand at top of stack, or divide operand from stack by operand from memory. Place quotient in stack and remainder in discard register. Derive sign of quotient algebraically from signs of operands.

POSSIBLE CR INDICATIONS. Overflow may occur in all modes except alphameric and logical. Underflow may occur in integer, floating-point, significant-point, and specified-point modes.

.....

FLR FLOAT REMAINDER



OPERATION. Convert operand at top of stack or operand from memory, from fixed-point to floating-point format. Make sign of result agree with sign of operand deeper in stack; make exponent of result 36 less than exponent of operand deeper in stack. Place result in top of stack.

POSSIBLE CR INDICATIONS. Underflow may occur in the integer, floating-point, specified-point, or significant-point computational modes as a result of exponent generation for the result.

.....

SQR SQUARE ROOT



OPERATION. Compute square root of either an operand from stack or one from memory. Place result in stack. Make sign of root equal to sign of operand. Place pseudo-remainder in discard register.

POSSIBLE CR INDICATIONS. Illegal operand condition bit set by negative operand. Overflow and underflow may be detected in the specified-point mode.

.....

RND ROUND



OPERATION. Add algebraically most significant bit of discard register to magnitude or mantissa of operand in top of stack or operand from memory, in manner specified by current computational mode.

POSSIBLE CR INDICATIONS. Overflow or underflow may occur in all modes except alphameric and logical.

3.2 SHIFT INSTRUCTIONS

The computational mode affects only the SR, SL, RR, RL, SAC, and NORM instructions; the manner in which these instructions are affected is explained below.

Alphameric and Logical Modes: The sign is shifted as magnitude

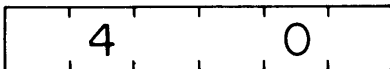
All other Modes: The sign is not shifted

The length of the operand to be shifted can be either single or double length as determined by the current computational mode.

Field definition can be used to shift only the logical product of FR and operand. The shift result is combined with the non-field bits of the operand.

For SR, SL, RR, and RL, the shift count is obtained from the top of the stack, and the operand deeper in the stack is shifted. If the shift count is put in the address field, it assembles as a literal preceding the shift.

SR SHIFT RIGHT

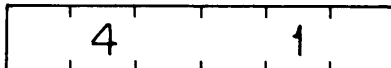


OPERATION. Shift operand in stack right in end-off manner, by amount specified modulo 128.

POSSIBLE CR INDICATIONS. None

.....

SL SHIFT LEFT



OPERATION. Shift operand in stack left in end-off manner, by amount specified modulo 128.

POSSIBLE CR INDICATIONS. None

.....

SAC SHIFT AND COUNT
NORM NORMALIZE } * *



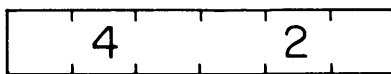
OPERATION. Shift operand in stack left in end-off manner until most significant magnitude or mantissa bit, as specified by current computational mode, is a ONE. Place number-of-places-shifted count in barrel shift register.

NOTE: When no shift is required to normalize the operand, the contents of the barrel shift register after normalization is zero. If the operand contains no ONES, then the contents of the barrel shift register contains 127 (octal 177) as the count.

POSSIBLE CR INDICATIONS. None

.....

RR ROTATE RIGHT

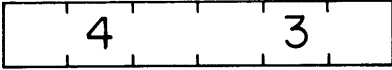


OPERATION. Shift operand in stack right, and end-around by amount specified modulo 128.

POSSIBLE CR INDICATIONS. None

* * All mnemonics express same operator

RL ROTATE LEFT



OPERATION. Shift operand in stack left, and end-around by amount specified modulo 128.

POSSIBLE CR INDICATIONS. None

.....

JUS JUSTIFY

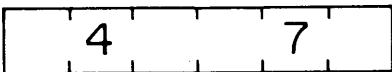


OPERATION. Shift (rotate) operand in stack right, and end-around by amount equal to number of ZERO's to right of least significant ONE in field register. Place shift count in barrel shift register.

POSSIBLE CR INDICATIONS. None

.....

UNJ UNJUSTIFY



OPERATION. Shift (rotate) operand in stack left and end-around by amount equal to the number of ZERO's right of least significant ONE in field register. Place shift count in the barrel shift register.

POSSIBLE CR INDICATIONS. None

SSH STREAMING SHIFT



OPERATION. Shift operand by amount and in manner specified by contents of the address register. These specifications are given below.

BIT(S)	VALUE	MEANING
11-17	Any	Shift count (0 to 128) which applies to any shift designated below:
10	ZERO	Shift end-around (rotate)
10	ONE	Shift end-off
9	ZERO	Shift left
9	ONE	Shift right
8	ZERO	Shift sign bit(s)
8	ONE	Do not shift sign bit(s)
7	ONE	Shift top of stack (single length)
6	ONE	Shift second stack word (single length)
5	ONE	Shift third stack word (single length)
4	ONE	Shift fourth stack word (single length)
3	ONE	Shift first and second stack words (double length)
2	ONE	Shift second and third stack words (double length)
1	ONE	Shift third and fourth stack words (double length)
0	ONE	Shift field register

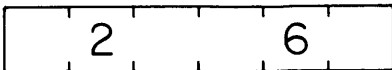
The list above indicates that more than one shift can be performed with a given address register word, and that words in the stack and/or the contents of the field register can be shifted. If two or more separate shifts are designated, they are performed sequentially, with the shift designated by a higher-numbered address register bit always preceding a shift designated by a lower-numbered bit. The capability of shifting either the stack or the field register allows the character stream to be processed by two techniques. In the first technique the stream is made to flow by an imaginary stationary point of attack by shifting the stack. In the second technique the point of attack is made to pass the stream by shifting the field register. This Streaming Shift is the only D851 instruction which must be preceded by indexing or indirect addressing. As for the "index" or count word, it may be any word in the system, and may be addressed in any addressing mode.

POSSIBLE CR INDICATIONS. None

3.3 LOGICAL INSTRUCTIONS

Computational modes affect logical instructions; the effect of these modes is briefly explained as follows: (1) the length of the operand can be either single or double length as determined by the current computational mode; (2) the sign position is included in the logical operations, only in the alphameric and logical modes; (3) field definition may be used to logically affect only certain field(s) within a word.

AND	AND	} * *
SCL	SELECTIVE CLEAR	
LMP	LOGICAL MULTIPLY	
B0001	AND EXPRESSION OF BOOLEAN TRUTH TABLE	



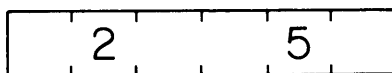
OPERATION. AND two operands from stack or AND one from memory and one from stack. Place result in stack.

	Bit of Operand in Top of Stack or Deeper in Stack	Bit of Operand in Memory or in Top of Stack	Bit of Result
	0	0	0
	0	1	0
	1	0	0
	1	1	1

POSSIBLE CR INDICATIONS. None

.....

OR	OR	} * *
SST	SELECTIVE SET	
LAD	LOGICAL ADD	
B0111	OR EXPRESSION OF BOOLEAN TRUTH TABLE	



* * All mnemonics express the same operator

OPERATION. OR two operators from stack, or OR one from memory and one from stack. Place result in stack.

	Bit of Operand in Top of Stack or Deeper in Stack	Bit of Operand in Memory or in Top of Stack	Bit of Result
	0	0	0
	0	1	1
	1	0	1
	1	1	1

POSSIBLE CR INDICATIONS. None

.....

ØRX	EXCLUSIVE OR	} * *
SCM	SELECTIVE COMPLEMENT	
HAD	HALF-ADD	
BBC	BIT-BY-BIT COMPARE	
BO110	EXCLUSIVE OR EXPRESSED AS BOOLEAN TRUTH TABLE	



OPERATION. Make bit-by-bit comparison between two operands from stack, or between one from stack and one from memory. Put a ONE in those bit positions whose operand bit pairs are not alike. Place result in stack.

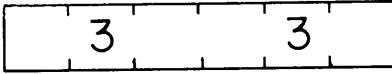
	Bit of Operand in Top of Stack or Deeper in Stack	Bit of Operand in Memory or in Top of Stack	Bit of Result
	0	0	0
	0	1	1
	1	0	1
	1	1	0

POSSIBLE CR INDICATIONS. None

.....

* * All mnemonics express the same operator

FILL SET TO FULL SCALE

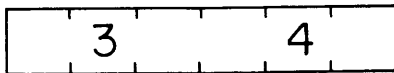


OPERATION. Set all bits in operand, either from stack or memory, to ONE.
Place result in stack.

POSSIBLE CR INDICATIONS. None

.....

CLR CLEAR TO ZERO



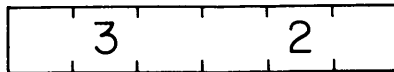
OPERATION. Clear all bits either from operand in memory, or from operand in stack. Place result in stack.

POSSIBLE CR INDICATIONS. None

.....

~~COM~~ COMPLEMENT
 NOT LOGICAL INVERSION

} * *



OPERATION. Logically complement those bits in operand, which may be either from stack or memory. Place result in stack.

POSSIBLE CR INDICATIONS. None

* * All mnemonics express same operator

IMP IMPLICATION
 BOO IO IMPLICATION EXPRESSION OF BOOLEAN TRUTH TABLE } * *



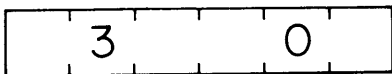
OPERATION. Indicate whether operand deeper in stack implies operand at top of the stack or whether operand in stack implies operand in memory; see truth table.

Bit of Operand in Top of Stack or Deeper in Stack	Bit of Operand in Memory or in Top of Stack	Bit of Result
0	0	0
0	1	0
1	0	1
1	1	0

POSSIBLE CR INDICATIONS. None

.....

EXT EXTRACT

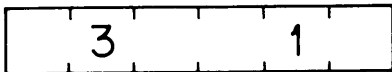


OPERATION. AND field register and operand in stack, or in memory. Place result in stack.

POSSIBLE CR INDICATIONS. None

.....

INF INSERT FIELD



* * All mnemonics express same operator

OPERATION. Using the field register as a mask, let the defined field remain unchanged in operand in memory, or in word at top of stack, if both operands are in stack. Insert non-field portion of operand deepest in stack into corresponding portion of the word in memory or at top of the stack. Place result in top of stack.

Example:

Field Register:	0	1	0
Word in Memory:	a	b	c
Word in Stack:	d	e	f
Result in Top of Stack:	d	b	f

POSSIBLE CR INDICATIONS. None

.....

B0000

OCTAL CODE. 11; 34

OPERATION. Assemble this macro as a STEP; follow by a CLEAR. Place result in top of stack. (Two operands in top of stack are lost; if an address is used, one operand is lost.)

POSSIBLE CR INDICATIONS. None

.....

B0011 "B TRUE"

OCTAL CODE. 11

OPERATION. Assemble instruction as a STEP. Place result, which is equal to operand deeper in stack, at top of stack.

POSSIBLE CR INDICATIONS. None

B0100 "REVERSE IMPLICATION"

OCTAL CODE. 10; 22; 32

OPERATION. Assemble this macro as a REV; follow by an IMP. Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B0101 "A TRUE"

OCTAL CODE. 10; 22; 11

OPERATION. Assemble this macro as a REV and STEP; result is equal to word which was in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1000 "NOR"

OCTAL CODE. 26; 34

OPERATION. Assemble this macro as an $\bar{O}R$; follow by $C\bar{O}M$. Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1001 "MATERIAL EQUIVALENCE OPERATIONS"

OCTAL CODE. 25; 34

OPERATION. Assemble this macro as an ORX ; follow by COM . Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1010 "NOT A"

OCTAL CODE. 10; 22; 11; 34

OPERATION. Assemble this macro as: REV , STEP , and COM . Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1011 "REVERSE IMPLICATION NOT"

OCTAL CODE. 10; 22; 32; 34

OPERATION. Assemble this macro as: REV , IMP , followed by COM . Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1100 "B NOT"

OCTAL CODE. 11; 34

OPERATION. Assemble this macro as a STEP followed by COM. Place complement of operand, which is deeper in stack, at top of stack. (Two operands in top of stack are lost; if an address is used, one operand is lost)

POSSIBLE CR INDICATIONS. None

.....

B1101 "IMPLICATION NOT"

OCTAL CODE. 32; 34

OPERATION. Assemble this macro as IMP; follow by COM. Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1110 "NAND"

OCTAL CODE. 26; 34

OPERATION. Assemble this macro as an AND; follow by COM. Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

B1111

OCTAL CODE. 11; 27

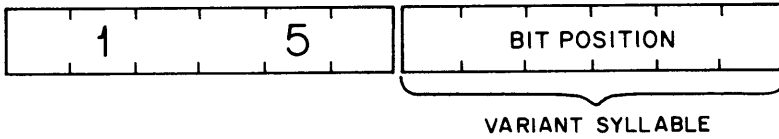
OPERATION. Assemble this macro as a STEP; follow by a FILL. Place result in top of stack. (Two operands in top of stack are lost; if an address is used, one operand is lost.)

POSSIBLE CR INDICATIONS. None

3.4 BIT-MANIPULATING INSTRUCTIONS

These instructions are not affected by the current computational mode. The bit number is specified by the optional addressing feature. Only bits 0 through 47 may be specified. (The address will be interpreted as modulo 48 if a bit address in excess of 47 is used.)

SB SET BIT

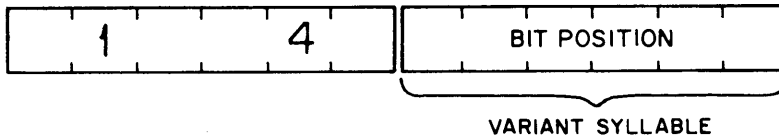


OPERATION. Set bit in stack operand, as specified by addressing, to ONE.

POSSIBLE CR INDICATIONS. None

.....

CLB CLEAR BIT

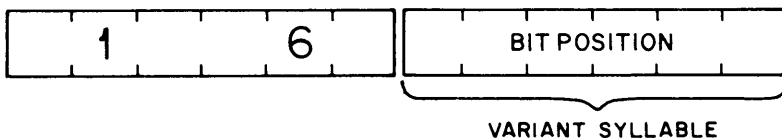


OPERATION. Clear bit in stack operand, as specified by addressing, to ZERO.

POSSIBLE CR INDICATIONS. None

.....

CHB COMPLEMENT BIT

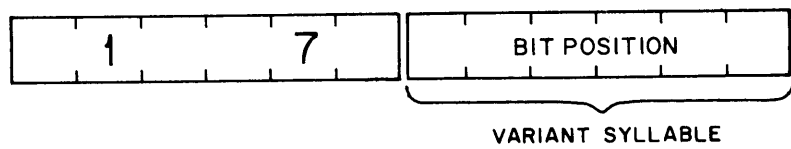


OPERATION. Complement bit, as specified by addressing, in stack operand.

POSSIBLE CR INDICATIONS. None

.....

INB INSERT BIT

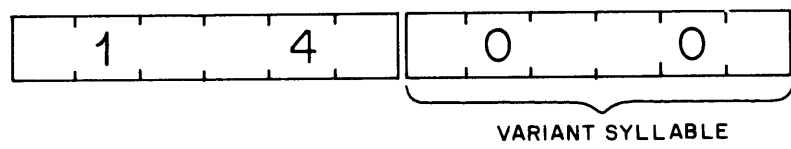


OPERATION. Insert designated bit from operand in top of stack into operand deeper in stack and place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

POS SET POSITIVE

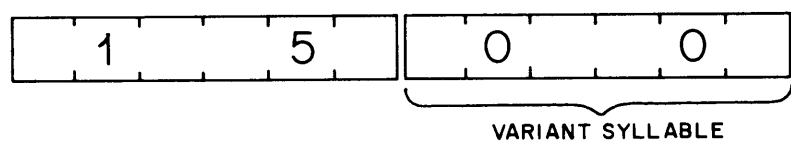


OPERATION. Make specified operand positive.

POSSIBLE CR INDICATIONS. None

.....

NEG SET NEGATIVE

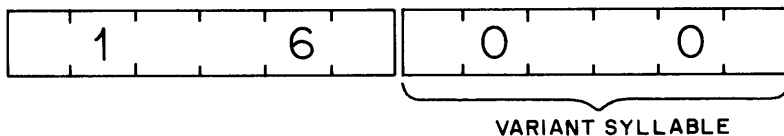


OPERATION. Make specified operand negative.

POSSIBLE CR INDICATIONS. None

.....

CHS CHANGE SIGN

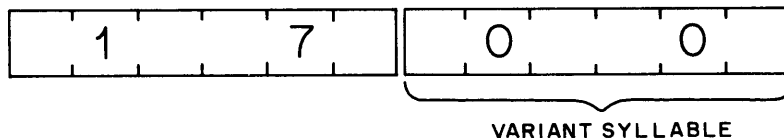


OPERATION. Complement sign of operand.

POSSIBLE CR INDICATIONS. None

.....

INS INSERT SIGN

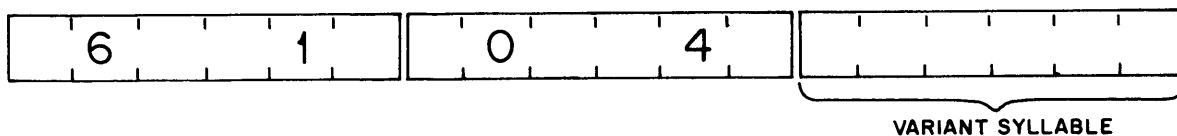


OPERATION. Insert sign of operand at top of stack into sign of operand beneath, or insert sign from operand in memory into stack operand. Place result in top of stack.

POSSIBLE CR INDICATIONS. None

.....

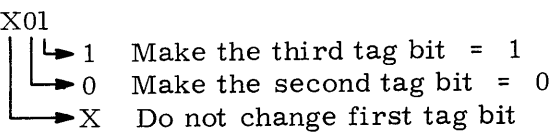
STB SET (OR CLEAR) TAG BIT(s)



OPERATION. Set up tag bits of top of stack in manner specified by variant syllable. Do not change any other bits in word.

The assembly program address-field coding relating to variant specification is shown below. Specification of the values of the three tag bits are as follows: (1) use three-digit positions (analogous to the three tag bits), (2) in each position, use the following convention:

- X - Do not change the bit value
- 0 - Make the bit = 0.
- 1 - Make the bit = 1

Example: X01


In addition, mnemonics may be used for variant specification, as indicated in the chart below.

Variant Specification Mnemonics	Variant Syllable Value	Variant Syllable Meaning
CLEAR	70	Clear all tag bits to "0" (000)
FILL	77	Set all tag bits to "1" (111)
USED	44	Set third tag bit to "1" (XX1)
UNUSED	40	Set third tag bit to "0" (XX0)
NØP	30	Set first and second bits to "0" (00X)
INT	31	Set first tag bit to "0"; Set second tag bit to "1"; Tag bit value = (01X) (Done to cause interrupt)
JUMP	32	Set first tag bit to "1"; Set second tag bit to "0"; Tag-bit value = (10X) (Done to cause jump)
EMPTY	33	Set first tag bit to "1"; Set second tag bit to "1"; Tag bit value = (11X)

POSSIBLE CR INDICATIONS. None

3.5 LITERAL INSTRUCTIONS

Literal instructions permit the use of a data word (or syllable) to be taken from the program string. In addition to the two specific literal instructions, FAS (Fetch from AR to Stack), with absolute addressing specified, may be used to generate an 18 bit literal.

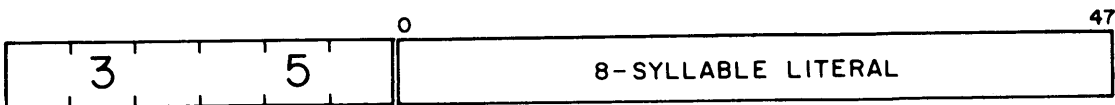
Literal instructions are not affected by the current computational mode.

Any of the following literal expressions may assemble as either LIT, SLIT, or FAS, depending upon the length of the literal specified. The SLIT is used by the assembler whenever possible. ZERO s are inserted to the left of the field specified by SLIT or FAS when entered in the top of the stack.

NOTE: The type of literal to be used may be indicated by the following mnemonics:

- LIT Decimal literal
- LHØL Holerith literal
- LØCT Octal literal
- LVFD Variable field-defined number bits, format/field-defined
- LXR Index register expressed as a literal
- LBIN Binary literal
- LITD Decimal literal, double-precision

LIT ENTER LITERAL WORD IN STACK

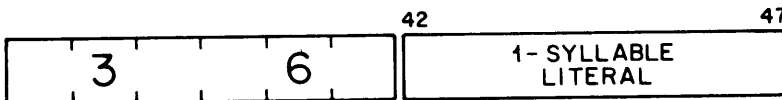


OPERATION. Enter eight program syllables that follow in top of stack as a word.

POSSIBLE CR INDICATIONS. None

.....

SLIT ENTER SHORT LITERAL IN STACK



OPERATION. Enter program syllable that follows into the six least significant bits of the cleared top of stack.

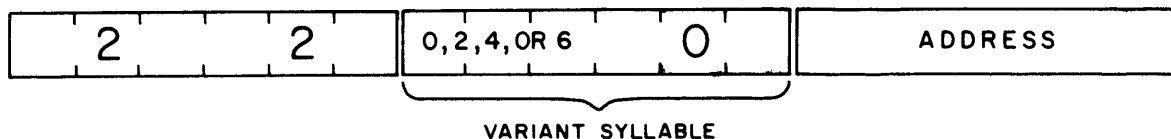
POSSIBLE CR INDICATIONS. None

.....

3.6 FETCH AND STORE INSTRUCTIONS

These instructions are not affected by the current computational mode; the number of words fetched is a function of the fetch instruction.

FMS FETCH FROM MEMORY TO STACK

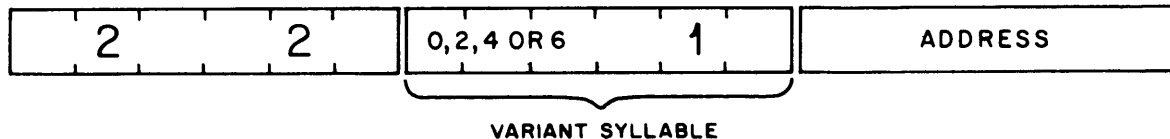


OPERATION. Fetch from memory to stack. Push contents of stack down to make room for new word.

POSSIBLE CR INDICATIONS. Incorrect parity bit assignment will be detected and may cause interrupt.*

.....

FML FETCH FROM MEMORY TO LDB



OPERATION. Initiate fetch of single word from Memory Module to LDB location designated by current contents of LDB pointer. Retain memory address

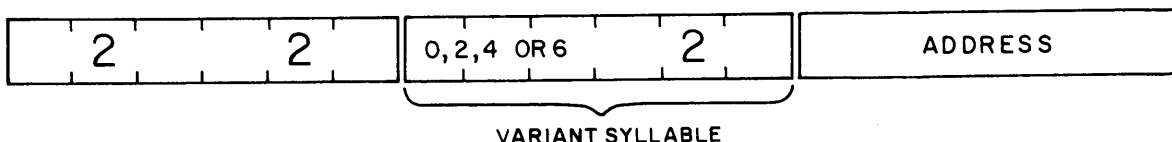
* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

and single word fetch indication (two most significant bits = 00) in LDB location until replaced by word from memory. Increment LDB pointer by one.

POSSIBLE CR INDICATIONS. Incorrect parity will be detected as in FMS.

.....

FBML FETCH BLOCK FROM MEMORY TO LDB

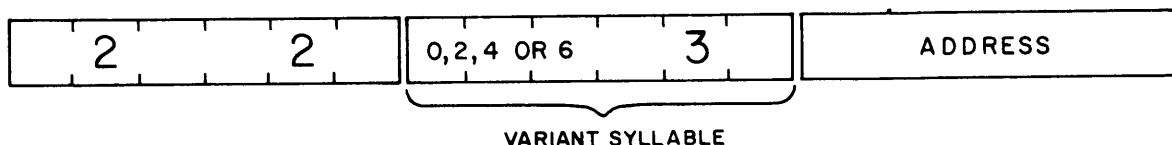


OPERATION. Initiate transfer of (N) words from Memory Module to LDB; make (N) the six least significant bits of word in top of stack. Transfer block to LDB beginning at address specified by current setting of LDB pointer. Set two most significant bits of LDB address equal to 01; set bits 24 - 29 equal to the six least significant bits of the top of the stack and the 18 least significant bits equal to the memory address. Continue until first word of block from memory reaches LDB and overwrites contents. Increment LDB pointer by (N).

POSSIBLE CR INDICATIONS. Incorrect parity will be detected as in FMS.

.....

FCML FETCH CHARACTER STREAM TO LDB



OPERATION. Initiate transfer of (N) characters of size (M), beginning at character (P); (N), (M) and (P) are specified by the 18 least significant bits of the top of the stack. Transfer this information to bits 12 - 29 and set the two most significant bits to 10 of the address in LDB currently indicated by LDB pointer.

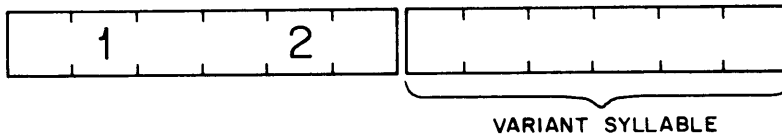
(This information and the empty tag-bit flag are overwritten when the characters begin streaming in from memory, — characters may overlap words.)

Increase the local data pointer by (N).

POSSIBLE CR INDICATIONS. Incorrect parity detected as in FMS.

.....

FLS FETCH FROM LDB TO STACK



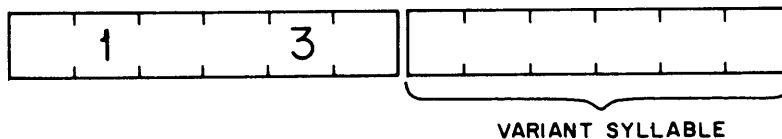
OPERATION. When tag bits indicate that the word in LDB has been fetched from memory, fetch word to stack.

If tag bits indicate that the transfer from the Memory Module is not completed, wait for word to arrive from memory and proceed as above.

POSSIBLE CR INDICATIONS. Incorrect parity detected and condition register bit set, — may cause interrupt.*

.....

SSL STORE FROM STACK TO LDB

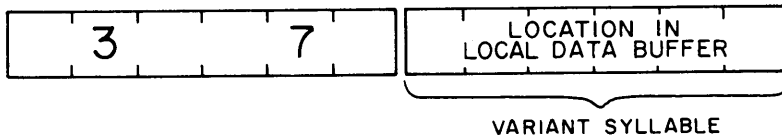


OPERATION. Store word at top of stack in address of LDB as specified by variant syllable.

POSSIBLE CR INDICATIONS. None

* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

SLP SET LDB POINTER

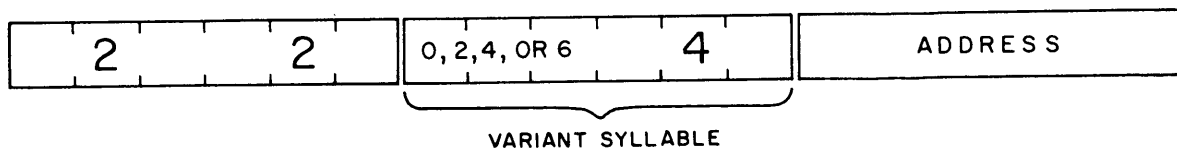


OPERATION. Store variant syllable in LDB pointer, which now points to LDB location involved in next FML, FBML or FCML.

POSSIBLE CR INDICATIONS. None

.....

SSM STORE FROM STACK TO MEMORY

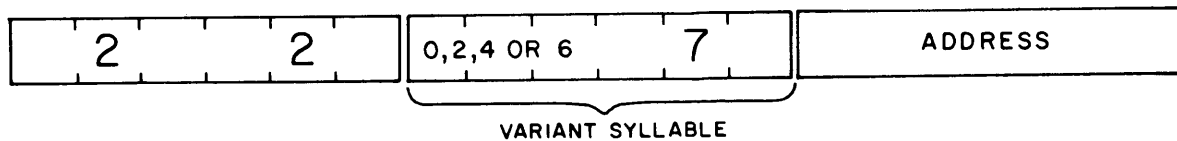


OPERATION. Store top of stack in memory address specified.

POSSIBLE CR INDICATIONS. None

.....

FAS FETCH ABSOLUTE ADDRESS FROM AR TO STACK

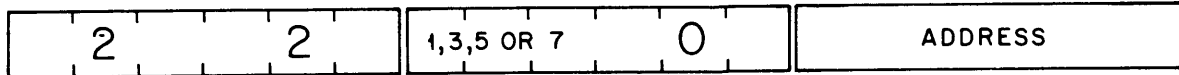


OPERATION. Clear word at top of stack and place contents of address register in the 18 least significant bits. Clear ASR.

NOTE: When used with absolute address, FAS becomes an 18-bit indexible literal. FAS is generated by the assembly program in response to coding of a literal when the value of the literal is greater than $(2^6 - 1)$ but less than 218.

POSSIBLE CR INDICATIONS. None

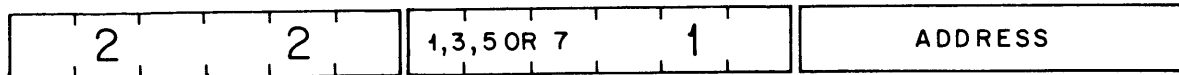
DLM DUMP LDB INTO MEMORY



OPERATION. Dump words from local data buffer into Memory Module beginning at address specified in instruction address field, — the last address of local data buffer to be transferred is in top of stack, the first word is always location zero.

POSSIBLE CR INDICATIONS. Parity errors can be discovered on transfers to the memory.*

LML LOAD DATA BLOCK FROM MEMORY INTO LDB

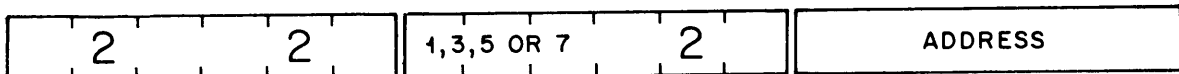


OPERATION. Initiate transfer of a block of (N) words from memory at address specified to local data buffer, beginning at address zero. (N) is specified by the six least significant bits of the top of the stack.

POSSIBLE CR INDICATIONS. Incorrect parity may be detected and interrupt may occur.

.....

DRM DUMP REGISTERS IN MEMORY

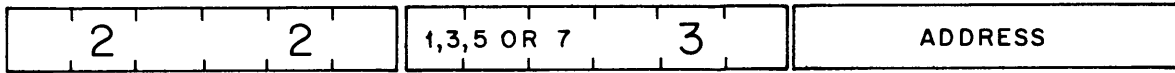


OPERATION. Dump registers into memory beginning at address specified in the instruction address field; the address of the last register to be dumped is in the top of the stack; the first register transferred is register number zero.

POSSIBLE CR INDICATIONS. None

* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

LMR LOAD MEMORY INTO REGISTERS

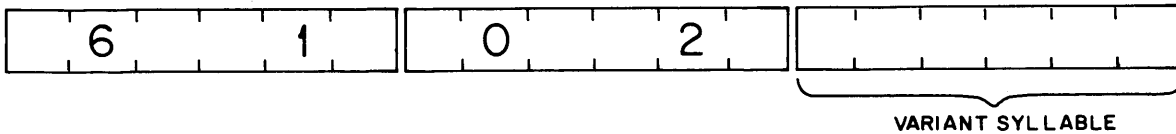


OPERATION. Initiate loading of registers from memory beginning at address specified and beginning at address zero. (N) is specified by the six least significant bits of the top of the stack.

POSSIBLE CR INDICATIONS. Incorrect parity may be detected and interrupt may occur.

.....

FRS FETCH REGISTER TO STACK

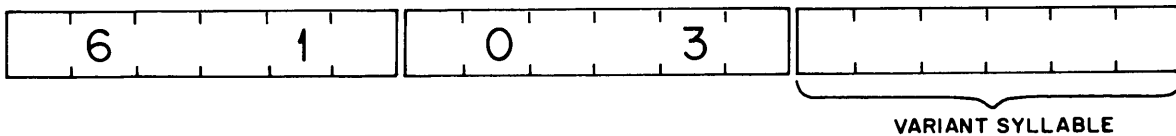


OPERATION. Fetch contents of register designated, mnemonically or numerically in address field, to top of stack.*

POSSIBLE CR INDICATIONS. None

.....

SSR STORE FROM STACK TO REGISTER



OPERATION. Store contents of top word of stack to register designated mnemonically or numerically in address field.

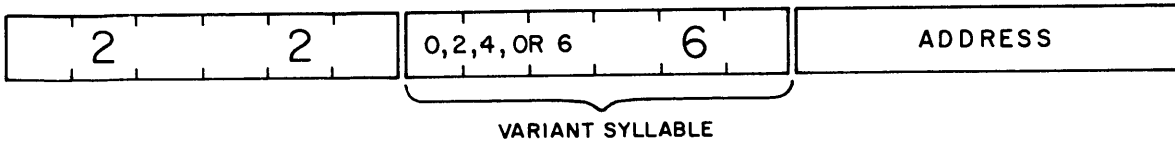
POSSIBLE CR INDICATIONS. If the designated register is protectable in normal mode; if the SSR instruction is executed in normal mode; and if the mask register bit protecting the designated register is set; the changing of the register is inhibited, and interrupt occurs.

* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

3.7 INDIRECT ADDRESSING INSTRUCTIONS

The current computational mode has no effect on any of these instructions, since the arithmetic unit is not used to perform address arithmetic.

AMA ADD MEMORY AND AR

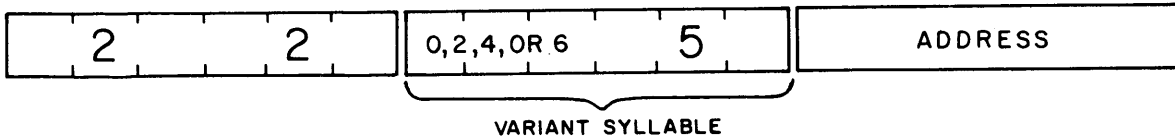


OPERATION. Add contents of the 18 least significant bits of the memory locations specified and contents of address register. Place sum modulo 2^{18} in address register.

POSSIBLE CR INDICATIONS. None

.....

FMA FETCH FROM MEMORY TO AR

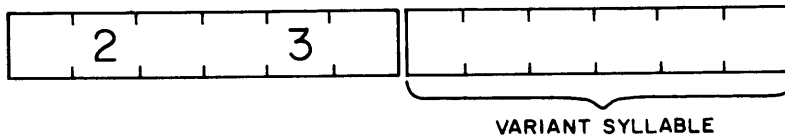


OPERATION. Replace address register with the 18 least significant bits of memory location specified.

POSSIBLE CR INDICATIONS. Memory parity failure will be detected and condition register bit will be set, - may cause interrupt.*

.....

FLA FETCH LDB WORD TO AR



* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

OPERATION. Replace address register with the 18 least significant bits of location specified in the LDB.

POSSIBLE CR INDICATIONS. Memory parity failure will be detected and condition register bit will be set, — may cause interrupt.

.....

3.8 STACK-MANIPULATING INSTRUCTIONS

These instructions are not affected by the current computational mode. For programming ease, two instructions are included for double-precision operands: DUPD and REVD.

The stack consists of three logically contiguous yet discrete elements, i. e. ,

Operand Element

Extension Element

Memory Element

The operand element consists of the four words at the top of the stack, — the maximum number of words which are required for any instruction.

The extension element consists of the 16 words of stack, immediately beneath the operand element, and deeper in the stack. Words "pushed down" from the operand element are parity checked as they enter this element. Although the words in this element cannot be specified in the instructions listed below, the following instructions cause words to be "pushed down" into this element; i. e. , DUP, DUPD, TRIP, QUAD, and RTS with certain variants. A word will pass from the extension element to the operand and the extension element of the stack are located in the Computer Module to enable the most rapid accessing. Words which are "pushed down" through the extension element are passed on to the memory element automatically.

The memory element of the stack is of program-defined length. The memory stack element length may most easily be determined by:

Setting the memory bounds register to define the upper address limit.

Setting the tag bits to interrupt when the lower limit address operand reaches the top of the stack.

Setting the stack pointer to the address which contains the word tagged for interrupt.

The Stack Pointer (SP) is a register containing the memory address of the next word to be fetched to the extension element or to receive a word from the extension element. As words are received from the extension element, the contents of this SP are counted up, and counted down when words pass from the memory to the extension element. If the memory element of the stack has been set up as recommended above, the following error conditions will be detected:

Memory Stack Overflow:

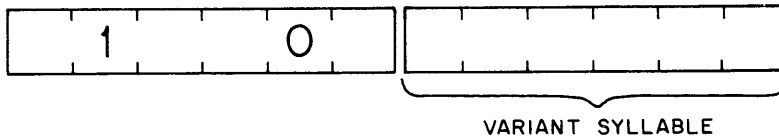
The number of words transferred to memory has exceeded the number allowed for in programmed specification, — the "exceed memory bounds" condition will activate an interrupt.

Memory Stack Underflow:

The operand leaving the top of the stack was designated initially to be the bottom of the stack, — tag-bit interrupt will be activated.

.....

RTS REARRANGE TOP OF STACK



OPERATION. Rearrange contents of three top stack words as specified by variant syllable. Push unspecified operands deeper in stack and beneath the three which may be specified; stack operands, from top down, are mnemonically denoted by alphabets: A, B, C, D.

Variant specifications are as follows:

First mnemonic character:

Indicates which of the former stack operands will be at top of stack

Second mnemonic character:

Indicates which of former stack operands will be next position deeper in stack

Third mnemonic character:

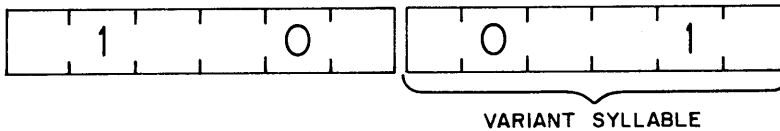
Indicates which of former stack operands will be second position deeper in stack

For the full repertoire of RTS variant possibilities, refer to subsection 2.9, Stack Manipulation.

POSSIBLE CR INDICATIONS. If one of the top three words in the stack is tagged to cause a jump or an interrupt; if an RTS instruction is executed which would move a word from below the tagged word to a position above it; and if the mask register is set to recognize tag-bit operations, — the tagged word has the same effect as if it were used as an operand, — and the RTS operation is inhibited.

.....

DUP DUPLICATE OPERAND IN STACK



OPERATION. Set top two positions of stack equal to operand specified, — may be top of stack or word from memory.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

DUPD DUPLICATE DOUBLE LENGTH

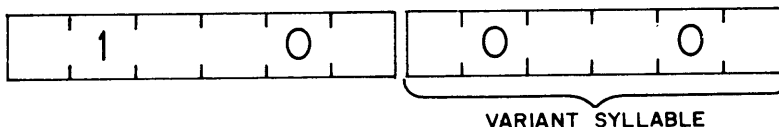
OCTAL CODE. 10; 05; 10; 04

OPERATION. Duplicate double-precision format operands in top of stack. Push operands located deeper in stack further down.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

TRIP TRIPLICATE OPERAND IN STACK



OPERATION. Set top three positions of stack equal to operand specified, - may be top of stack or word from memory.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

QUAD QUADRUPLICATE STACK OPERAND

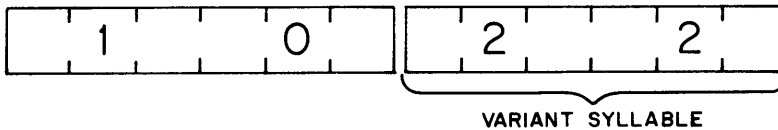
OCTAL CODE. 10; 00; 10; 01

OPERATION. Push three operands next to top of stack down three places. Make top four positions of stack equal to top of stack, or fetch operand from memory and quadruplicate.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

REV REVERSE OPERANDS IN STACK

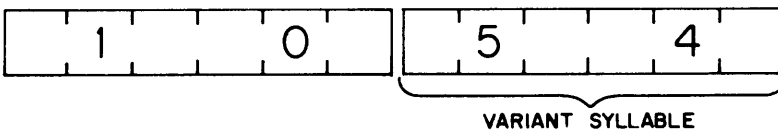


OPERATION. Reverse order of two words in top of stack, or fetch word from memory and reverse order in stack.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

REVD DOUBLE-LENGTH REVERSE

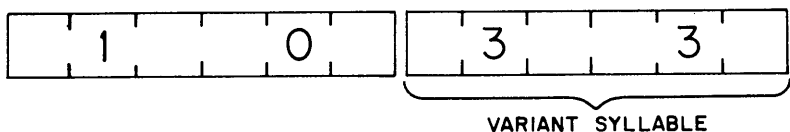


OPERATION. Reverse two double-precision operands in stack.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

CYCU CYCLE STACK UP

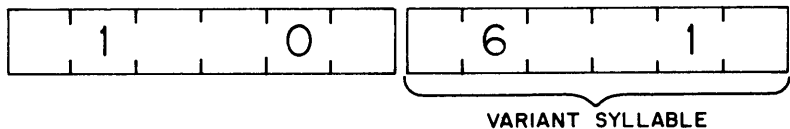


OPERATION. Cycle four words in top of stack up one. Push operand formerly at top of stack down beneath other three.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

CYCD CYCLE STACK DOWN

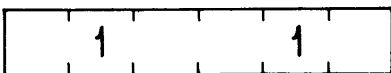


OPERATION. Push three operands at top of stack down one position. Place operand from fourth deepest position in top of stack.

POSSIBLE CR INDICATIONS. Same as RTS.

.....

STEP STEP STACK UP



OPERATION. Step entire stack and stack extension up one word, — operand previously at top is lost.

POSSIBLE CR INDICATIONS. Incorrect parity may be detected as the word leaves the stack extension and enters the position fourth from the top, — an interrupt is possible. *

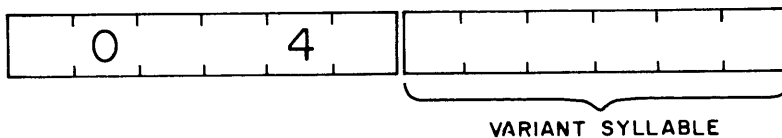
If the word originally at the top of the stack is tagged to cause an interrupt or a jump, and if the mask register is set to recognize tag bits, the tagged word has the same effect as if it were used as an operand and the STEP is inhibited.

.....

3.9 INDEX INSTRUCTIONS

The index instructions are not affected by the current computational mode. The address field specifies which index register, relative to the BXR, will be used. For index modification instructions, see pages 3-45, 3-46.

X INDEX

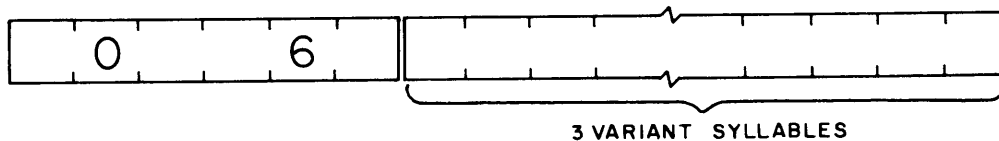


OPERATION. Add contents field of designated index to address of next instruction.

POSSIBLE CR INDICATIONS. None

.....

XA INDEX AUGMENTED



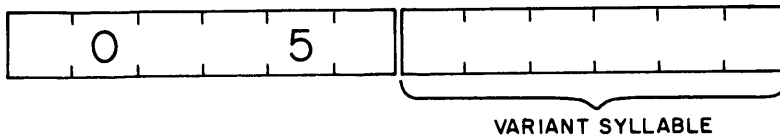
* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

OPERATION. Add contents field of designated index to address of next instruction.

POSSIBLE CR INDICATIONS. None

.....

XM INDEX AND MODIFY

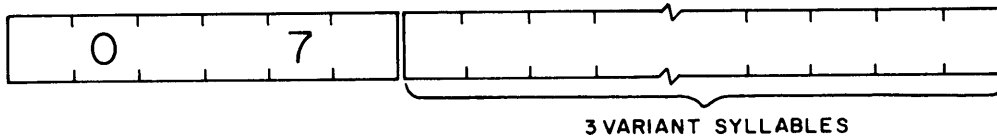


OPERATION. Add contents field of designated index to address of next instruction.
Add self-contained increment to the content.

POSSIBLE CR INDICATIONS. None

.....

XMA INDEX AUGMENTED AND MODIFIED

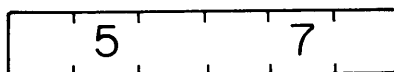


OPERATION. Add contents field of designated index to address of next instruction.
Add self-contained increment to the content.

POSSIBLE CR INDICATIONS. None

.....

XS INDEX BY TOP OF STACK



OPERATION. Specify top of stack to be used in modifying addressed fields of all following instructions requiring indexing.

POSSIBLE CR INDICATIONS. None

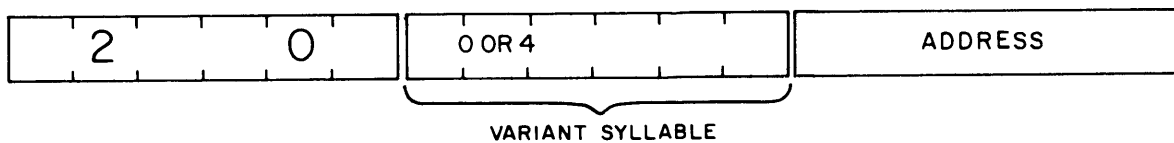
.....

3.10 JUMP INSTRUCTIONS

The jump class instructions which are affected by the current computational mode are: JGR, JGA, JLS, JLA, JEQ, JZE, and JFS. In each instruction, the comparison involves only those bits in the operands which are specified by the computational mode. Operands may be single or double precision, or field-defined.

If optional addressing is used with any of the jump instructions except JTB, SJF, SJSF, and SJST, they are assembled as a macro using the setup jump with the address, followed by the actual jump instruction. The address used with JBT indicates the bit to be tested in the top word of the stack.

SJF SET UP JUMP IF FALSE

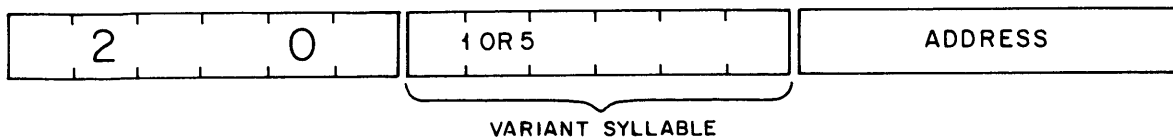


OPERATION. Place address of destination to which program control may be transferred; the syllable of destination; and condition false indicator in jump control register (JCR).

NOTE: Program control continues in normal sequence until jump operator encountered.

POSSIBLE CR INDICATIONS. None

SJT SET UP JUMP IF TRUE



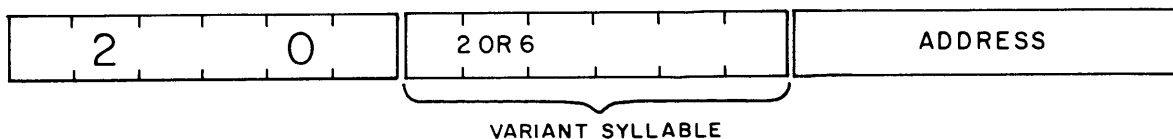
OPERATION. Place address of destination to which program control may be transferred; syllable of destination; and condition true indicator in jump control register.

NOTE: Program control continues in normal sequence until jump operator encountered.

POSSIBLE CR INDICATIONS. None

.....

SJSF SET UP JUMP TO SUBROUTINE IF FALSE

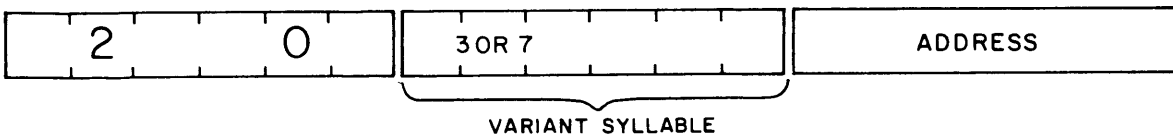


OPERATION. Place address of destination to which program control may be transferred; the syllable of destination; and condition false indicator in jump control register. Place amount by which base index register (BXR) may be incremented in JCR.

POSSIBLE CR INDICATIONS. None

.....

SJST SET UP JUMP TO SUBROUTINE IF TRUE



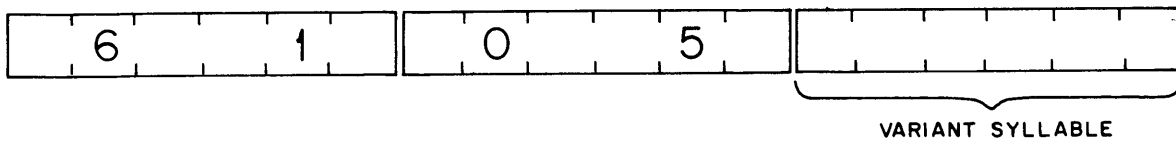
OPERATION. Place address of destination to which program control may be transferred; syllable of destination; and condition true indicator in jump con-

trol register. Place the amount that the base index register is to be incremented in the JCR.

POSSIBLE CR INDICATIONS. None

.....

JTB JUMP ON TAG BIT(S)

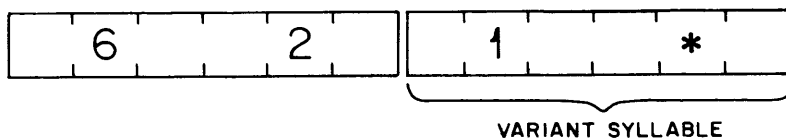


OPERATION. Test tag bits of word at top of stack with configuration of test bits specified by variant syllable. If result of this test agrees with true/false designation of last setup-jump instruction, jump; if not, continue in normal program sequence. Variant specifications accepted by assembly program agree with those for STB.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JXEZ JUMP ON RESULT OF INDEX EQUAL TO ZERO TEST

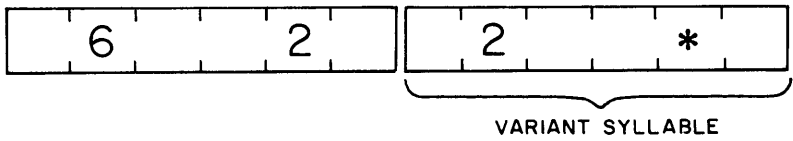


OPERATION. If index register designated in address field or index register most recently used in program has content field equal to zero, test result is true. If test result agrees with true/false indication of last setup-jump, jump to address setup. The "JXEZ" may be followed, in the operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

* Refer to pages 3-45, 3-46.

JXEL JUMP ON RESULT OF INDEX EQUAL TO LIMIT TEST

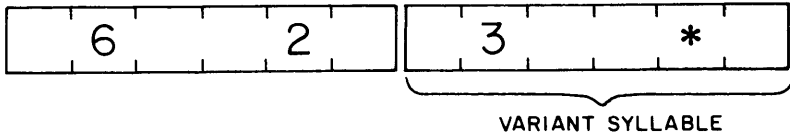


OPERATION. If index register designated in address field or index register most recently used in program has content field equal to its limit field, test result is true. If test result agrees with true/false indication of the last setup-jump, jump to address setup. The "JXEL" may be followed, in the operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

JXES JUMP ON RESULT OF INDEX EQUAL TO STACK TEST

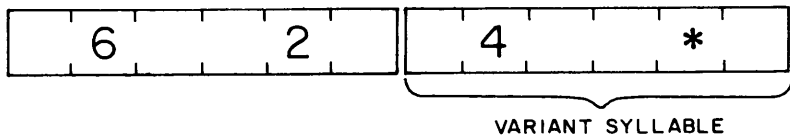


OPERATION. If index register designated in address field or index register most recently used in the program has a content field equal to the least significant 18 bits of top of stack (bits 30 - 47), test result is true. If test result agrees with true/false indication of last setup-jump, jump to address setup. The "JXES" may be followed in the operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

JXGL JUMP ON RESULT OF INDEX GREATER THAN LIMIT TEST



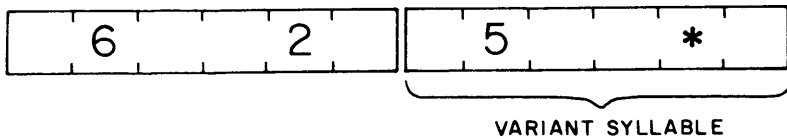
* Refer to pages 3-45, 3-46.

OPERATION. If index register designated in address field or index register last used in the program has a content field greater than its limit field, test result is true. If test result agrees with the true/false indication of last setup-jump, jump to address setup. The "JXGL" may be followed in the operation field by any of the allowed two-character mnemonics for index modification, modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

JXGS JUMP ON RESULT OF INDEX GREATER THAN STACK TEST

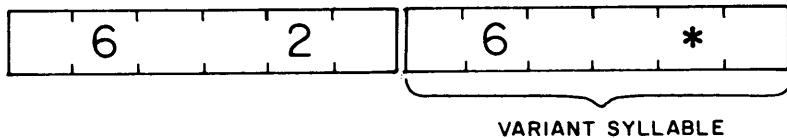


OPERATION. If index register designated in address field or index register last used in the program has a content field greater than the least significant 18 bits of the top of the stack (bits 30 - 47), the test result is true. If the test result agrees with true/false indication of last setup-jump, jump to address setup. The "JXGS" may be followed in the operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

JXLL JUMP ON RESULT OF INDEX LESS THAN LIMIT TEST



OPERATION. If index register designated in address field or index register most recently used in program has content field less than its limit field, test result is true. If test result agrees with true/false indication of the last setup-jump, jump to address setup. The "JXLL" may be followed in the

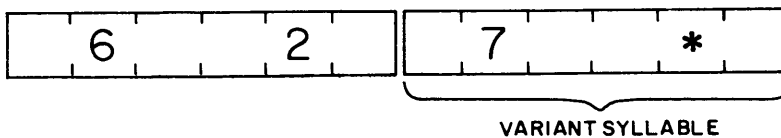
* Refer to pages 3-45, 3-46.

operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

JXLS JUMP ON RESULT OF INDEX LESS THAN STACK TEST



OPERATION. If index register designated in address field or index register most recently used in program has content field less than the least significant 18 bits of top of stack (bits 30 - 47), test result is true. If test result agrees with true/false indication of last setup-jump, jump to address setup. The "JXLS" may be followed in the operation field by any of the allowed two-character mnemonics for index modification; modification when called for, follows test.

POSSIBLE CR INDICATIONS. May be jumped-trapped or subroutine jump-trapped.

.....

Any combination of index test-jump and index modification instructions is possible. That is to say, any one of the 7 index test-jump instructions may be combined with any of the 7 index modification instructions, which follow, to provide 7 possible modifications for each index test-jump instruction, or a total of 49 possible permutations. To determine the complete value of the variant syllable synthesized in such combinations, replace the * indicated in the variant syllable block of the individual instruction by the value shown on the next page.

If only modification is desired, then the first digit of the variant syllable is zero, i. e.:

Instruction	Variant Syllable
XRS	01
XAØ	02

Instruction	Variant Syllable
XAI	03
XAS	04
XSØ	05
XRL	06
XSS	07

If no modification is desired, then the second digit of the variant syllable is zero, i. e. :

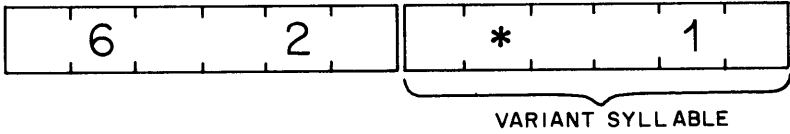
Instruction	Variant Syllable
JXEZ	10
JXEL	20
JXES	30
JXGL	40
JXGS	50
JXLL	60
JXLS	70

For other combinations see below.

Composite Instruction	Variant Value.	Composite Instruction	Variant Value	Composite Instruction	Variant Value
		JXESAI	3 3	JXGSRL	5 6
JXEZRS	1 1	JXESAS	3 4	JXGSSS	5 7
JXEZAØ	1 2	JXESSØ	3 5		
JXEZAI	1 3	JXESRL	3 6		
JXEZAS	1 4	JXESSS	3 7	JXLLRS	6 1
JXEZSØ	1 5			JXLLAØ	6 2
JXEZRL	1 6			JXLLAI	6 3
JXEZSS	1 7	JXGLRS	4 1	JXLLAS	6 4
		JXGLAØ	4 2	JXLLSØ	6 5
		JXGLAI	4 3	JXLLRL	6 6
JXELRS	2 1	JXGLAS	4 4	JXLLSS	6 7
JXELAØ	2 2	JXGLSØ	4 5		
JXELAI	2 3	JXGLRL	4 6		
JXELAS	2 4	JXGLSS	4 7	JXLSRS	7 1
JXELSØ	2 5			JXLSAØ	7 2
JXELRL	2 6			JXLSAI	7 3
JXELSS	2 7	JXGSRS	5 1	JXLSAS	7 4
		JXGSAØ	5 2	JXLSSØ	7 5
		JXGSAI	5 3	JXLSRL	7 6
JXESRS	3 1	JXGSAS	5 4	JXLSSS	7 7
JXESAØ	3 2	JXGSSØ	5 5		

XRS REPLACE INDEX BY TOP OF STACK
 JX__RS REPLACE INDEX BY TOP OF STACK AFTER TESTING

} * *

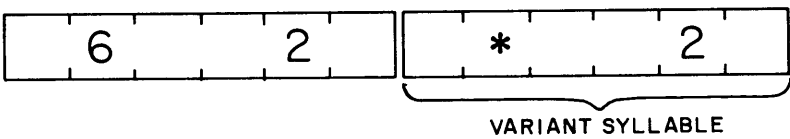


OPERATION. Replace contents field of index register designated in address field or index register most recently used in program by the least significant 18 bits of top of stack (bits 30 - 47).

POSSIBLE CR INDICATIONS. None

XA0 ADD ONE TO INDEX
 JX__A0 ADD ONE TO INDEX AFTER TESTING

} * *

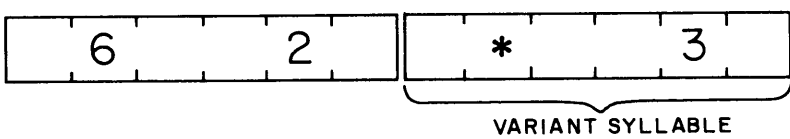


OPERATION. Add one to contents-field of index register designated in address field or index register most recently used in the program.

POSSIBLE CR INDICATIONS. None

XAI ADD INCREMENT TO INDEX
 JX__AI ADD INCREMENT TO INDEX AFTER TESTING

} * *

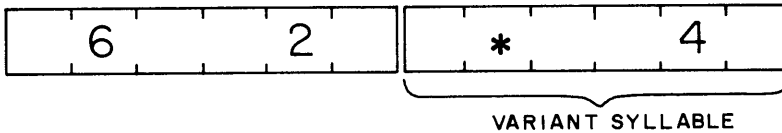


OPERATION. Add self-contained signed increment to contents-field of index register designated in address field or index register most recently used in the program.

POSSIBLE CR INDICATIONS. None

* Refer to pages 3-45, 3-46. * * All mnemonics express same operator.

XAS ADD STACK TO INDEX
 JX_AS ADD STACK TO INDEX AFTER TESTING } * *

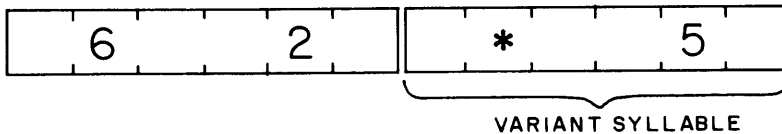


OPERATION. Add the least significant 18 bits of top of stack (bits 30 - 47) to contents field of index register designated in address field or index register most recently used in the program.

POSSIBLE CR INDICATIONS. None

.....

X~~S~~ SUBTRACT ONE FROM INDEX
 JX_~~S~~ SUBTRACT ONE FROM INDEX AFTER TESTING } * *

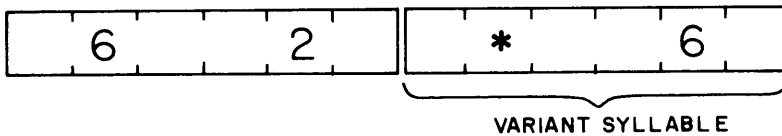


OPERATION. Subtract one from contents-field of index register designated in address field or index register most recently used in the program.

POSSIBLE CR INDICATIONS. None

.....

XRL REFILL INDEX CONTENT FROM LIMIT FIELD
 JX_RL REFILL INDEX CONTENT FROM LIMIT FIELD AFTER TESTING } * *



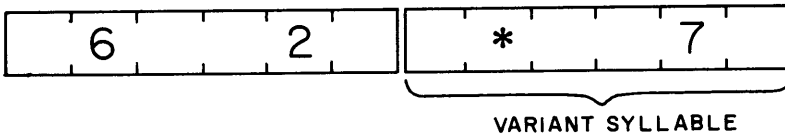
OPERATION. Replace contents-field of index register designated in address field or index register most recently used in program by its limit field.

POSSIBLE CR INDICATIONS. None

* Refer to pages 3-45, 3-46. * * All mnemonics express same operator.

XSS SUBTRACT TOP OF STACK FROM INDEX
 JX_ SS SUBTRACT TOP OF STACK FROM INDEX AFTER TESTING

} * *

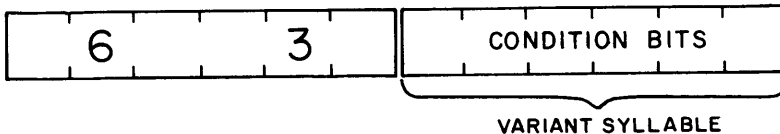


OPERATION. Subtract the least significant 18 bits of top of stack (bits 30 - 47) from index register designated in address field or index register most recently used in program.

POSSIBLE CR INDICATIONS. None

.....

JCB JUMP ON STATE OF CONDITION BIT

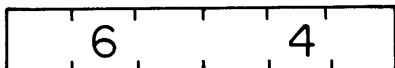


OPERATION. If condition register bit designated by contents of address field is a ONE, test result is true. If bit is ONE, if mask register is not set to cause interrupt on this condition, or, if condition is overflow or underflow, clear condition bit. If condition is overflow or underflow and if, JCB immediately follows instructions causing overflow or underflow, — inhibit interrupt even if called for by mask register. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May result in the clearing of the designated bit. May be jump-trapped or subroutine jump-trapped.

.....

JGR JUMP ON RESULT OF TEST FOR GREATER



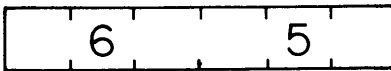
* Refer to pages 3-45, 3-46. * * All mnemonics express same operator.

OPERATION. If either operand in stack is greater than operand in memory, or operand deeper in stack is greater than operand at top of stack, test result is true; comparison is made ignoring sign bits. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JGA JUMP ON RESULT OF TEST FOR ABSOLUTE-VALUE GREATER

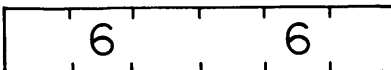


OPERATION. If either operand in stack is greater than operand in memory, or operand deeper in stack is greater than operand at top of stack, test result is true; comparison is made ignoring sign bits. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JLS JUMP ON RESULT OF TEST FOR LESS

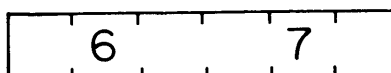


OPERATION. If either operand in stack is smaller than operand in memory, or operand deeper in stack is smaller than operand at top of stack, test result is true. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JLA JUMP ON RESULT OF TEST FOR ABSOLUTE-VALUE LESS

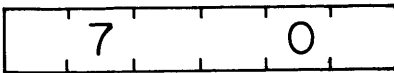


OPERATION. If either operand in stack is smaller than operand in memory, or operand deeper in stack is smaller than operand at top of stack, test result is true; comparison is made ignoring sign bits. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JEQ JUMP ON RESULT OF TEST FOR EQUALITY

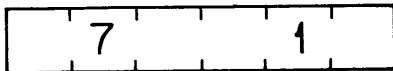


OPERATION. If either operand in stack is equal to operand in memory, or two top operands in stack are equal, test result is true. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JZE JUMP ON RESULT OF TEST FOR ZERO

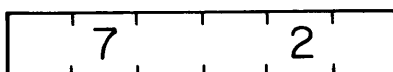


OPERATION. If mantissa or magnitude bits, as defined by current computational mode, are all ZERO's for either operand in top of stack or designated operand in memory, test result is true. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JFS JUMP ON RESULT OF TEST FOR FULL SCALE

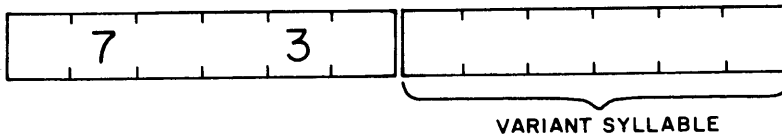


OPERATION. If mantissa or magnitude bits, as defined by current computational mode, are all ONE's, for either operand in top of stack or designated operand in memory, test result is true. If true/false result of the test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JBT JUMP ON RESULT OF BIT TEST IN TOP OF STACK

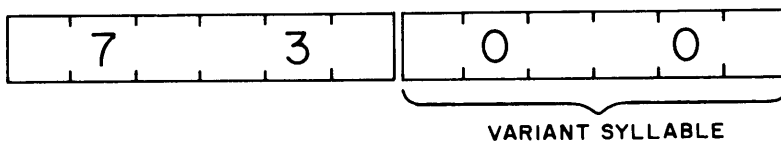


OPERATION. If designated bit of top word of stack is a ONE, test result is true. If the true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JSI JUMP ON RESULT OF SIGN TEST

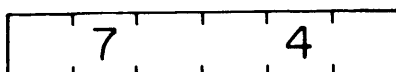


OPERATION. If either sign of word at top of stack or sign of designated word in memory is negative, the test result is true. If true/false result of test agrees with true/false designation of last setup-jump, jump.

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

JUMP JUMP UNCONDITIONALLY



OPERATION. Unconditionally jump either to jump-address previously set up or to jump-address designated in address field. .

POSSIBLE CR INDICATIONS. May be jump-trapped or subroutine jump-trapped.

.....

RET RETURN FROM SUBROUTINE



OPERATION. Return to parent program at syllable following instruction which entered subroutine. Restore computational mode, overflow and underflow bits, and setting of base index register as they were before subroutine was entered. Obtain restoring information from index register zero before BXR is restored.

POSSIBLE CR INDICATIONS. May be jump-trapped. May alter overflow and underflow bits.

.....

RETI RETURN TO INTERRUPTED PROGRAM AT POINT OF INTERRUPT



OPERATION. Resume interrupted program at point designated by contents of interrupt return register. If in control mode, enter normal mode.

POSSIBLE CR INDICATIONS. If attempted in normal mode with mask register set to inhibit RETI, inhibit the attempted return, and interrupt.

.....

3.11 CONTROL INSTRUCTIONS

These instructions are not affected by the current computational mode. Addressing in the normal usage does not apply; variant addressing is described for each instruction.

STOP STOP PROCESSOR



OPERATION. Control mode ONLY: Stop processor. In control mode, this instruction will cause an interrupt.*

POSSIBLE CR INDICATIONS. None

.....

NOP NO OPERATION



OPERATION. None

POSSIBLE CR INDICATIONS. None

.....

ENM ENTER NORMAL OPERATIONAL MODE

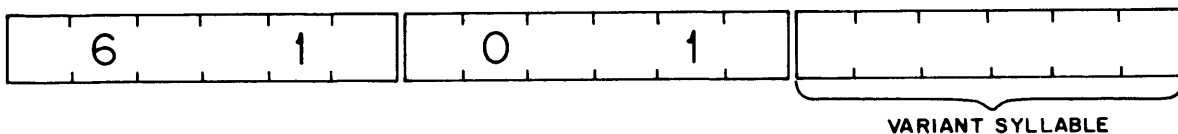


OPERATION. Cause processor to enter normal mode of operation when it is currently in control mode. Prohibit use while in normal mode. Execute no jump to a previously interrupted program.

POSSIBLE CR INDICATIONS. None

.....

ESP ENTER EXECUTIVE AND SCHEDULING PROGRAM



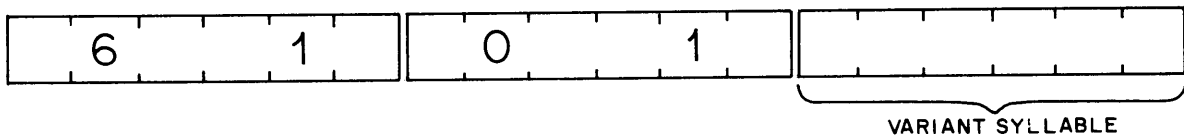
* Interrupt system explained in "D851 Modular Data Processor"; pp 3-23 to 3-31

OPERATION. Interrupt (enter control mode) with jump to location specified by the setting of the Base Interrupt Address Register (BIAR) and the number coded in the address field.

POSSIBLE CR INDICATIONS. None

.....

SFCN EXECUTE SPECIAL FUNCTION

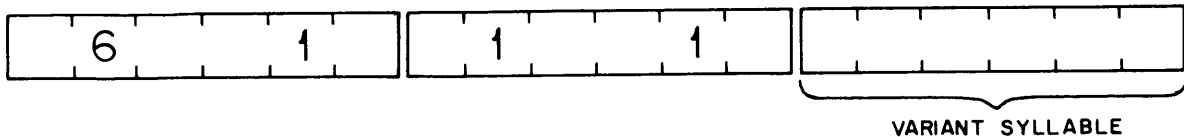


OPERATION. If designated special function device is attached to Computer Module, execute designated function and continue with program.

POSSIBLE CR INDICATIONS. If designated special function device is not attached to the Computer Module, interrupt as a function of the base interrupt address register and the designated special function number; when the device is not attached, SFCN becomes ESP.

.....

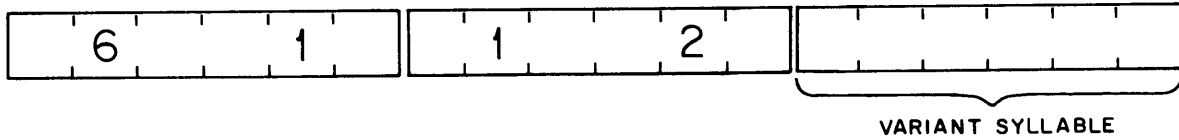
CCM COMMUNICATE WITH COMPUTER MODULE



OPERATION. Transmit word in top of stack to Computer Module designated by variant syllable; the receiving computer OR's the transmitted word to its condition register.

POSSIBLE CR INDICATIONS. The receiving computer may be interrupted, depending upon its mask register contents. Sending computer may be interrupted if not in normal mode.

I0 INITIATE INPUT/OUTPUT PROGRAM



OPERATION. Transmit top four words of stack as an input/output descriptor to Input-Output Module designated in address field. The descriptor is accepted and the input/output program initiated, — if the device required (as designated in the descriptor), is not busy.

POSSIBLE CR INDICATIONS. If the designated Input-Output Module is inoperable, or otherwise unable to accept the descriptor, the "no access to input/output" bit in the condition register is set. This bit is not set if the Input-Output Module is able to accept the descriptor but the required device is either inoperable or busy. However, either the initial descriptor or any descriptor linked within the initiated input/output program can call for the setting of any condition bit in any processor. If in normal mode, and if the mask register is set to prohibit input/output, interrupt will occur without transmission of the descriptors and without changing of the stack

.....

MEM INITIATE MEMORY MODULE PROGRAM

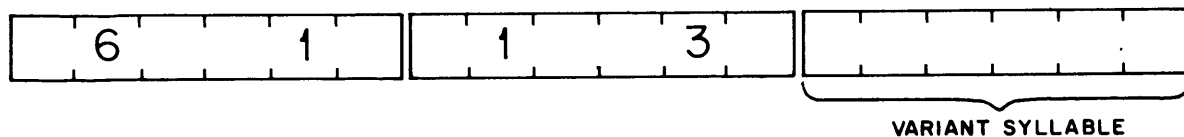


OPERATION. Transmit top four words of stack as memory descriptor to Memory Module which is designated, within descriptor, as first module involved in memory program so-initiated.

POSSIBLE CR INDICATIONS. When the Memory Module is completed, the "memory complete" bit in the condition register is set.

.....

I10 INTERRUPT INPUT/OUTPUT PROGRAM

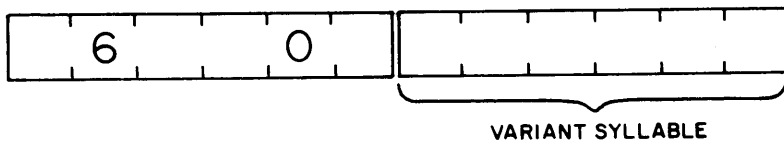


OPERATION. Transmit top four words of stack as an input/output descriptor to Input-Output Module designated in address field. The descriptor is accepted and the input/output program initiated, — if device required (as designated in descriptor), is operable.

POSSIBLE CR INDICATIONS. If the designated Input-Output Module is inoperable, or otherwise unable to accept the descriptor, the "no access to input/output" bit in the condition register is set. This bit is not set if the Input-Output Module is able to accept the descriptor but the required device is inoperable. However, either the initial descriptor or any descriptor link within the initiated input/output program can call for the setting of any condition bit in any processor. If in normal mode, and if the mask register is set to prohibit input/output, interrupt will occur without transmission of the descriptors and without changing of the stack

CAUTION: An interrupted input/output program does not leave any usable record of a point at which it was interrupted. An interrupted input/output program may have to be reinitiated from its start.

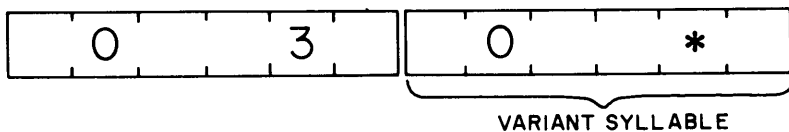
CCB CLEAR CONDITION BIT



OPERATION. Clear designate bit of condition register. Interpret bit number modulo 48. Operation forbidden for condition bits reserved for ESP; use when in normal mode and when specified bit is set to cause interrupt.

POSSIBLE CR INDICATIONS. Specified condition bit is clear, if an illegal operation interrupt is generated.

INT INTEGER MODE



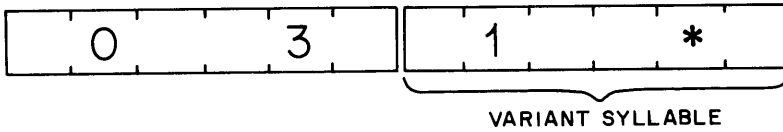
* To determine variant syllable value, refer to page 3-60

OPERATION. Enter integer mode.

POSSIBLE CR INDICATIONS. None

.....

SPP SPECIFIED-POINT MODE

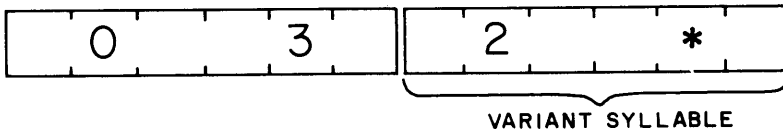


OPERATION. Enter specified-point mode.

POSSIBLE CR INDICATIONS. None

.....

FLT FLOATING-POINT MODE

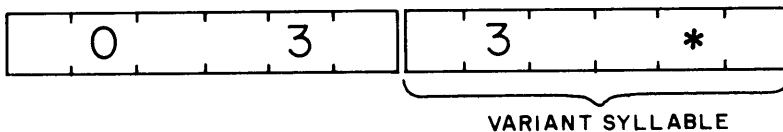


OPERATION. Enter floating-point mode.

POSSIBLE CR INDICATIONS. None

.....

SIP SIGNIFICANT-POINT MODE



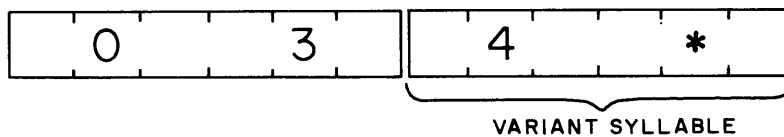
* To determine variant syllable value, refer to page 3-60

OPERATION. Enter significant-point mode.

POSSIBLE CR INDICATIONS. None

.....

FIX FIXED-POINT MODE

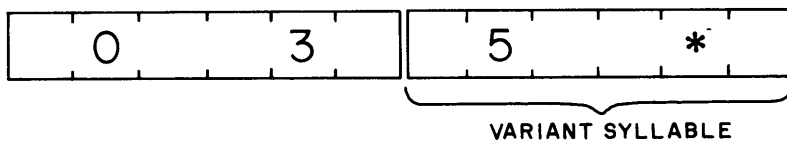


OPERATION. Enter fixed-point mode.

POSSIBLE CR INDICATIONS. None

.....

ALPH ALPHAMERIC MODE

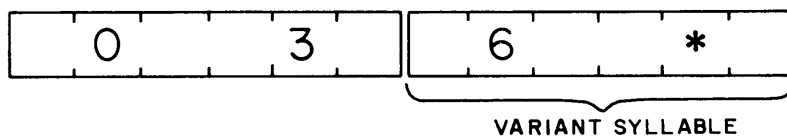


OPERATION. Enter alphameric mode.

POSSIBLE CR INDICATIONS. None

.....

LOG LOGICAL MODE



* To determine variant syllable value, refer to page 3-60

OPERATION. Enter logical mode.

POSSIBLE CR INDICATIONS. None

.....

Any combination of computational mode instructions and operand format designators (specified in the address field) may be used. The operand format designators recognized by the assembler are as follows.

Designator	Variant Syllable Value - 2nd Digit	Use
(blank)	0	Single-precision operands with no field definition
D	2	Double-precision operands with no field definition
F	4	Field-defined, single-precision operands
ALT	6	Field-defined, double-precision operands (use field definition on alternate operands)

For a more detailed discussion, refer to subsection 2.5, D851 Arithmetic Operations.

3.12 PSEUDO-OPERATIONS

The use of these assembler-oriented operators is not concerned with the current computational mode or optional addressing.

BLØCK FILL CURRENT FOUR-WORD MEMORY BLOCK WITH NOP'S

OCTAL CODE. 01 01 ... (Generate NOP'S as necessary to fill four-word block)

OPERATION. None

POSSIBLE CR INDICATIONS. None

.....

~~WORD~~ FILL CURRENT INSTRUCTION WORD WITH NOP's

OCTAL CODE. 01 01 ... (Generate NOP's as necessary to fill single word)

OPERATION. None

POSSIBLE CR INDICATIONS. None

.....

APPENDIX A

THE CASE FOR POLISH NOTATION

The rules for writing "correct" algebraic expressions yield the general form $y = A \text{ (operand) } B$. This form has a direct counterpart in the three address computer organization, but not in two or one address organizations. In general, two objections to the orthodox algebraic form exist. First, the formation rules permit a number of equivalent forms. The forms $y = a + b \times c - d$, $y = (a + (b \times c)) - d$, and $y = a + ((b \times c) - d)$ are considered equivalent by the reader, but they differ widely in symbol content and perhaps value when evaluated by a digital processor. Second, the order in which operands and operators appear does not correspond to the order desirable for machine evaluation. A typical machine language equivalent for the above is:

```
Clear Add c
Multiply b
Add a
Subtract d
Store y.
```

This brief example suggests two desirable features for a language to be interpreted by a machine: freedom from ambiguity and compactness of representation.

A notation invented by the Polish logician J. Lukasiewicz overcomes these objections to the orthodox algebraic form. Lukasiewicz demonstrated that the need for parentheses could be eliminated entirely if the operators were always written after their operands (suffix Polish notation) instead of between operands (infix notation). Thus $y = a + b \times c$ becomes $y \text{ abc } \times + =$ and $y = (a + b) \times c$ is written $y \text{ ab } + c \times =$. No ambiguous forms exist since the degree of the operator (the number of required operands) and its placement uniquely define its operands. Further, compactness is achieved by eliminating the need for extra symbols (i. e., parentheses or other bracket forms). An

additional advantage is gained since the degree of an operator no longer need be limited to two, as is the case with infix notation.

The desirability of employing the Polish notation as an intermediate representation in translating automatic programming languages such as FORTRAN, ALGOL or COBOL has been recognized. The representation is compact and unambiguous and retains the correspondence between symbols written by the programmer and those present in the Polish intermediate form. The translation of orthodox form in the programming language to Polish is easily accomplished with the aid of a pushdown storage (stack) and the recognition of the usual priority of arithmetic operations. In a pushdown stack, the element most recently stored is the first element removed. The order typical of a representative subset of arithmetic operators by increasing weakness is:

↑ (exponentiation), ×, NEG (unary -), +, -, = .

The translation rules for this subset and translation examples are presented on pages A-4 and A-5.

The translation from Polish to the desired machine language is a simple task since the operator, operand order to Polish notation more nearly corresponds to that desirable for machine language.

The advantages of Polish notation take on added significance when it is viewed as a machine language. If variable names are interpreted as addresses for fetch instructions which transfer operands from main storage to a pushdown stack in the arithmetic unit, it is unnecessary to associate addresses with arithmetic operators. The top one or two locations in the arithmetic pushdown stack are implicitly addressed by an arithmetic operator. Further intermediate results normally returned to temporary storage in a conventional machine are held in the top position of the pushdown stack. Thus several accesses to main storage to recover intermediate results can be saved. A single address format would demand six storage cycles (including instruction fetches) for each temporary variable generated, stored, and subsequently retrieved. The Polish format requires only three.

In summary, the Polish form has several striking advantages over conventional command languages of the single or multiple address types.

1. It provides a unique, compact representation of source language expressions, and maintains a correspondence between source language operators and machine language operators.
2. It permits a more compact storage of program since address specification as part of each instruction is unnecessary. Thus in comparison with a single address computer of traditional design, two address parts and one operation part are saved for each

temporary storage cell used in the evaluation of an expression. In the case of a subroutine parameter, one address part and two operation parts are saved.

3. When directly interpreted, Polish notation is intrinsically more efficient because explicit reference to temporary storage is not required.
4. Simple checks may be made during translation from programming language to prevent syntactically invalid programs. For example, assign a weight of 1 to operands and a weight of $1 - n$ to operators of degree n . The cumulative weight of an expression is 1 if it is well formed.

SIMPLIFIED TRANSLATION RULES

Arithmetic operations in order of decreasing strength are:

\uparrow , \times , NEG, +, -, =.

The translation of the input expression to its (suffix) Polish equivalent output proceeds according to the following rules:

1. Commence scanning the input string symbolwise from the left. Call the current symbol S; the accessible element in the push-down store E.
2. If S is an operand, transcribe it directly to the output.
3. If S is a left-hand bracket, enter it into the pushdown store.
4. If S is an operator, compare with E:
If E is not weaker, E is transcribed to output, and S is compared with the newly accessible element in the push-down stack until a left-hand bracket or a weaker operator or an empty store is encountered.
S is then entered in the pushdown store.
5. If S is a right-hand bracket, entries are transcribed from the pushdown stack to the output until a left-hand bracket is encountered; then the L. H. bracket is deleted.
6. After the last S, remaining entries in the pushdown stack are transcribed to the output.

EXAMPLE OF TRANSLATION TO POLISH NOTATION

<u>RULE</u>	<u>INPUT</u>	<u>PUSH DOWN</u>	<u>OUTPUT</u>
1	$y = (a + b) \times c$	Empty	
2	y		y
4	=	=	y
3	((y
		=	
2	a	(y a
		=	
4	+	+	y a
		(
		=	
2	b	+	y ab
		(
		=	
5)	=	y a b +
4	x	x	y a b +
		=	
2	c	x	y a b + c
		=	
6			y ab + cx =

APPENDIX C

NUMERICAL LISTING OF INSTRUCTIONS

This appendix consists of Tables C-1 and C-2. Table C-1 lists the D851 machine instructions in order of ascending values for the octal codes. The variant syllables, where specified, are represented in binary by using an "o" for a ZERO and an "i" for a ONE. Where a binary value is not assigned or need not be assigned, the undefined state of the bit is represented by a period. The letter "S" in a variant column indicates a syllable containing an address; in the case of a literal, the number of syllables required to contain the literal is indicated. The letter "x" indicates that indexing is possible.

Table C-2 lists instructions and pseudo-operations which may be used in assembly language programming. The octal code is given for machine instructions, while octal codes are given for instructions generated to assemble macro-instructions. The address field notation indicates what type of addressing option is possible for each instruction.

TABLE C-1
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC	OPERATION
00		STØP	Stop Processor (Control Mode Only)
01		NØP	No Operation
02		ENM	Enter Normal Operational Mode
03	ooo ...	INT	Enter Integer Computational Mode
03	ooi ...	SPP	Enter Specified-Point Computational Mode
03	oio ...	FLT	Enter Floating-Point Computational Mode
03	oii ...	SIP	Enter Significant-Point Computational Mode
03	ioo ...	FIX	Enter Fixed-Point Computational Mode
03	ioi ...	ALPH	Enter Alphameric Computational Mode
03	iio ...	LØG	Enter Logical Computational Mode
03	iii ...		Not Defined
03	... oo.		Enter Single Precision Non-Field Defined Mode
03	... oi.	D	Enter Double-Precision Computational Mode
03	... io.	F	Enter Field-Defined Single-Precision Computational Mode
03	... ii.	ALT	Enter Field-Defined Computational Mode on Alternate Operands
04	S	X	Index
05	S	XM	Index and Modify Index Word
06	SSS	XA	Index Augmented
07	SSS	XAM	Index Augmented and Modify Index Word
10	oo. ...	RTS	A-- Rearrange Top of Stack - Hold Top Word

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
10	oi. ...	RTS	B--	Rearrange Top of Stack - Second Word to Top
10	io. ...	RTS	C--	Rearrange Top of Stack - Third Word to Top
10	ii. ...	RTS	D--	Rearrange Top of Stack - Fourth Word to Top
10	..o o..	RTS	-A-	Rearrange Top of Stack - Top Word to Second
10	..o i..	RTS	-B-	Rearrange Top of Stack - Hold Second Word
10	..i o..	RTS	-C-	Rearrange Top of Stack - Third Word to Second
10	..i i..	RTS	-D-	Rearrange Top of Stack - Fourth Word to Second
10oo	RTS	--A	Rearrange Top of Stack - Top Word to Third
10oi	RTS	--B	Rearrange Top of Stack - Second Word to Third
10io	RTS	--C	Rearrange Top of Stack - Hold Third Word
10ii	RTS	--D	Rearrange Top of Stack - Fourth Word to Third
11		STEP		Step Stack Up One
12	Sx	FLS	ADDR	Fetch from Local Data Buffer to Stack
13	Sx	SSL	ADDR	Store from Stack to Local Data Buffer
14	Sx	CIB	BIT	Clear Bit in Top of Stack
15	Sx	SB	BIT	Set Bit in Top of Stack
16	Sx	CHB	BIT	Change Bit in Top of Stack
17	Sx	INB	BIT	Insert Bit in Stack
20	o.. ... SSSx	OP	ADDR	Self-Relative Addressing
20	i.. ... SSSx	OP	ADDR, A	Absolute Addressing

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
20	.oo ... SSSx	SJF	ADDR	Set Up Jump if False
20	.oi ... SSSx	SJT	ADDR	Set Up Jump if True
20	.io ... SSSx	SJSF	ADDR	Set Up Jump to Subroutine if False
20	.ii ... SSSx	SJST	ADDR	Set Up Jump to Subroutine if True
21				Not Defined
22	oo. ... SSSx	OP	ADDR	Self-Relative Addressing
22	oio ... SSSx	OP	ADDR, R	Addressing Relative to Base Address Register
22	io. ... SSSx	OP	ADDR, A	Absolute Addressing
22	ii. ... SSSx	OP	ADDR, X	Addressing Relative to Base Index Register
22	..o ooo SSSx	FMS	ADDR	Fetch from Memory Module to Stack
22	..o ooi SSSx	FML	ADDR	Fetch from Memory Module to Local Data Buffer
22	..o oio SSSx	FBML	ADDR	Fetch Block from Memory Module to Local Data Buffer
22	..o oii SSSx	FCML	ADDR	Fetch Characters from Memory Module to Local Data Buffer
22	..o ioo SSSx	SSM	ADDR	Store from Stack to Memory Module
22	..o ioi SSSx	FMA	ADDR	Fetch from Memory Module to Address Register
22	..o iio SSSx	AMA	ADDR	Add from Memory Module to Address Register
22	..o iii SSSx	FAS	ADDR	Fetch from Address Register to Stack
22	..i ooo SSSx	DLM	ADDR	Dump Local Data Buffer into Memory Module
22	..i ooi SSSx	LML	ADDR	Load Block from Memory Module into Local Data Buffer
22	..i oio SSSx	DRM	ADDR	Dump Registers into Memory Module
22	..i oii SSSx	LMR	ADDR	Load Block from Memory Module into Registers

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
23	Sx	FLA	ADDR	Fetch from Local Data Buffer to Address Register
24		IMP		Implication
25		ØR		OR
26		AND		AND
27		ØRX		Exclusive OR
30		EXT		Extract
31		INF		Insert Field
32		CØM		Complement
33		FILL		Set to Full Scale
34		CLR		Clear to Zero
35	SSSSSSSS	LIT	WORD	Literal
36	Sx	SLIT	WORD	Short Literal
37	S	SLP	ADDR	Set Local Data Buffer Pointer
40		SR		Shift Right (by Amount Already in Stack)
41		SL		Shift Left (by Amount Already in Stack)
42		RR		Rotate Right (by Amount Already in Stack)
43		RL		Rotate Left (by Amount Already in Stack)
44		NØRM		Normalize
45		SSH		Streaming Shift
46		JUS		Justify
47		UNJ		Unjustify
50		ADD		Add
51		SUB		Subtract
52		MUL		Multiply
53		DIV		Divide
54		SQR		Square Root

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
55		RND		Round
56		FLR		Float Remainder
57		XS		Index by Top of Stack
60	Sx	CCB	BIT	Clear Condition Bit
61	01 S	ESP		Enter ESP
61	01 S	SCFN	NUMBER	Enter Special Function
61	02 Sx	FRS	REG	Fetch Register to Stack
61	03 Sx	SSR	REG	Store from Stack to Register
61	04 o.. ...	STB	X--	Set Tag Bits - Do Not Change First Bit
61	04 i.. o..	STB	O--	Set Tag Bits - Set First Bit to ZERO
61	04 i.. i..	STB	1--	Set Tag Bits - Set First Bit to ONE
61	04 .o. ...	STB	-X-	Set Tag Bits - Do Not Change Second Bit
61	04 .i. .o.	STB	-O-	Set Tag Bits - Set Second Bit to ZERO
61	04 .i. .i.	STB	-1-	Set Tag Bits - Set Second Bit to ONE
61	04 ..o ...	STB	--X	Set Tag Bits - Do Not Change Third Bit
61	04 ..i ..o	STB	--0	Set Tag Bits - Set Third Bit to ZERO
61	04 ..i ..i	STB	--1	Set Tag Bits - Set Third Bit to ONE
61	05 o.. o.o	JTB	X--	Jump on Tag Bits - Do Not Test First Bit
61	05 i.. o..	JTB	0--	Jump on Tag Bits - Test First Bit for ZERO
61	05 i.. i..	JTB	1--	Jump on Tag Bits - Test First Bit for ONE

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
61	05 .o. .o.	JTB	-X-	Jump on Tag Bits - Do Not Test Second Bit
61	05 .i. .o.	JTB	-0-	Jump on Tag Bits - Test Second Bit for ZERO
61	05 .i. .i.	JTB	-1-	Jump on Tag Bits - Test Second Bit for ONE
61	05 ..o ..o	JTB	--X	Jump on Tag Bits - Do Not Test Third Bit
61	05 ..i ..o	JTB	--0	Jump on Tag Bits - Test Third Bit for ZERO
61	05 ..i ..i	JTB	--1	Jump on Tag Bits - Test Third Bit for ONE
61	10	MEM		Initiate Memory Module Program
61	11 Sx	CCM	MODULE	Communicate with Processor Module
61	12 Sx	I \emptyset	MODULE	Initiate Input-Output Program
61	13 Sx	II \emptyset	MODULE	Interrupt Input-Output Program
62	ooo ...	X--		Modify Index without Testing
62	ooi ...	JXEZ--		Jump on Result of Index Equal Zero Test
62	oio ...	JXEL--		Jump on Result of Index Equal Limit Test
62	oii ...	JXES--		Jump on Result of Index Equal Stack Test
62	ioo ...	JXGL--		Jump on Result of Index Greater than Limit Test
62	ioi ...	JXGS--		Jump on Result of Index Greater than Stack Test
62	iio ...	JXLL--		Jump on Result of Index Less than Limit Test
62	iii ...	JXLS--		Jump on Result of Index Less than Stack Test
62	... ooo	JX--		Test Index without Modifying Index
62	... ooi	-X--RS		Replace Index Content by Top of Stack

TABLE C-1 (Cont'd)
D851 MACHINE INSTRUCTIONS

OCTAL CODE	VARIANT SYLLABLES	MNEMONIC		OPERATION
62	... oio	-X--A \emptyset		Add One to Index Content
62	... oii	-X--AI		Add Self-Contained Increment to Index Content
62	... ioo	-X--AS		Add Top of Stack to Index Content
62	... ioi	-X--S \emptyset		Subtract One from Index Content
62	... iio	-X--RL		Refill Index Content from Limit Field
62	... iii	-X--SS		Subtract Top of Stack from Index Content
63	Sx	JCB	BIT	Jump on State of Condition Bit
64		JGR		Jump on Result of Test for Greater
65		JGA		Jump on Result of Test for Absolute Value Greater
66		JLS		Jump on Result of Test for less
67		JLA		Jump on Result of Test for Absolute Value Less
70		JEQ		Jump on Result of Test for Equality
71		JZE		Jump on Result of Test for Zero
72		JFS		Jump on Result of Test for Full Scale
73	Sx	JBT	BIT	Jump on Result of Bit Test in Top of Stack
73	ooo ooo	JSI		Jump on Result of Sign Test in Top of Stack
74		JUMP		Jump Unconditionally As Previously Set Up
75		RET		Return from Subroutine
76		RETI		Return to Interrupted Program
77		---		Stop Processor (Control Mode Only)

TABLE C-2
D851 ASSEMBLY LANGUAGE INSTRUCTIONS

OCTAL CODES AND VARIANT SYLLABLES	MNEMONIC	ADDRESS FIELD	OPERATION
01 01 01 ...	WORD		Generate Enough NOP's to Fill Current Instruction Word
01 01 01 ...	BLOCK		Generate Enough NOP's to Fill Current Four-Word Instruction Block
04 S	X	NUMBER	Index Address Portion of Next Instruction
04 S ---		ADDRESS/NUMBER	Index Address Portion of Current Instruction
05 S	XM	NUMBER	Index Address Portion of Next Instruction; Modify Index Word
05 S ---		ADDRESS/*NUMBER	Index Address Portion of Current Instruction; Modify Index Word
06 SSS	X	NUMBER	Index Address Portion of Next Instruction
06 SSS	XA	NUMBER	Index Address Portion of Next Instruction
06 SSS ---		ADDRESS/NUMBER	Index Address Portion of Current Instruction
07 SSS	XM	NUMBER	Index Address Portion of Next Instruction; Modify Index Word
07 SSS	XAM	NUMBER	Index Address Portion of Next Instruction; Modify Index Word
07 SSS ---		ADDRESS/*NUMBER	Index Address Portion of Current Instruction; Modify Index Word
10 00	TRIP		Triplicate Top Word of Stack
22 . . o ooo SSS 10 00	TRIP	ADDRESS	Triplicate Word Fetched from Memory Module
12 S 10 00	TRIP	ADDRESS, L	Triplicate Word Fetched from Local Data Buffer
10 01	DUP		Duplicate Top Word of Stack
22 . . o ooo SSS 10 01	DUP	ADDRESS	Duplicate Word Fetched from Memory Module
12 S 10 01	DUP	ADDRESS, L	Duplicate Word Fetched from Local Data Buffer
10 22	REV		Reverse Two Top Words of Stack
22 . . o ooo SSS 10 22	REV	ADDRESS	Reverse Top Word of Stack and Word Fetched from Memory Module
12 S 10 22	REV	ADDRESS, L	Reverse Top Word of Stack and Word Fetched from Local Data Buffer
10 33	CYCU		Cycle Top Four Words of Stack Up One
10 54	REVD		Reverse Two Top Double-Precision Operands in Stack
10 61	CYCD		Cycle Top Four Words of Stack Down One
10 00 10 01	QUAD		Quadruplicate Top Word of Stack
22 . . o ooo SSS 10 00 10 01	QUAD	ADDRESS	Quadruplicate Word Fetched from Memory Module
12 S 10 00 10 01	QUAD	ADDRESS	Quadruplicate Word Fetched from Local Data Buffer
10 05 10 04	DUPD		Duplicate Top Double-Precision Operand in Stack

TABLE C-2 (Cont'd)
D851 ASSEMBLY LANGUAGE INSTRUCTIONS

OCTAL CODES AND VARIANT SYLLABLES	MNEMONIC	ADDRESS FIELD	OPERATION
12 Sx	FMS	ADDRESS, L	Fetch from Local Data Buffer to Stack
12 S ---		ADDRESS, L	Use Word Fetched from Local Data Buffer as Operand in Current Instruction
13 Sx	SSM	ADDRESS, L	Store from Stack to Local Data Buffer
14 00	PØS		Set Top of Stack Positive
22 . . o 000 SSS 14 00	PØS	ADDRESS	Set Word Fetched from Memory Module Positive
12 S 14 00	PØS	ADDRESS, L	Set Word Fetched from Local Data Buffer Positive
15 00	NEG		Set Top of Stack Negative
22 . . o 000 15 00	NEG	ADDRESS	Set Word Fetched from Memory Module Negative
12 S 15 00	NEG	ADDRESS, L	Set Word Fetched from Local Data Buffer Negative
16 00	CHS		Change Sign in Top of Stack
22 . . o 000 SSS 16 00	CHS	ADDRESS	Change Sign in Word Fetched from Memory
12 S 16 00	CHS	ADDRESS, L	Change Sign in Word Fetched from Local Storage
17 00	INS		Insert Sign from Top Word of Stack into Second Stack Word
22 . . o 000 17 00	INS	ADDRESS	Insert Sign from Memory Word into Top Word of Stack
12 S 17 00	INS	ADDRESS, L	Insert Sign from Local-Storage Word into Top Word of Stack
20 o SSSx		ADDRESS	Compute Jump-Address as Self-Relative
20 i SSSx		ADDRESS, A	Compute Jump-Address as Absolute
20 . o . 000 SSSx			Jump to Syllable 0 of Word at Jump Address
20 . o . 001 SSSx			Jump to Syllable 1 of Word at Jump Address
20 . o . 010 SSSx			Jump to Syllable 2 of Word at Jump Address
20 . o . 011 SSSx			Jump to Syllable 3 of Word at Jump Address
20 . o . 100 SSSx			Jump to Syllable 4 of Word at Jump Address
20 . o . 101 SSSx			Jump to Syllable 5 of Word at Jump Address
20 . o . 110 SSSx			Jump to Syllable 6 of Word at Jump Address
20 . o . 111 SSSx			Jump to Syllable 7 of Word at Jump Address
20 . i . 000 SSSx		ADDRESS	Add 1 to Base Index Register on Entry to Subroutine
20 . i . 001 SSSx		ADDRESS, 4	Add 4 to Base Index Register on Entry to Subroutine
20 . i . 010 SSSx		ADDRESS, 8	Add 8 to Base Index Register on Entry to Subroutine
20 . i . 011 SSSx		ADDRESS, 12	Add 12 to Base Index Register on Entry to Subroutine
20 . i . 100 SSSx		ADDRESS, 16	Add 16 to Base Index Register on Entry to Subroutine
20 . i . 101 SSSx		ADDRESS, 20	Add 20 to Base Index Register on Entry to Subroutine
20 . i . 110 SSSx		ADDRESS, 24	Add 24 to Base Index Register on Entry to Subroutine

TABLE C-2 (Cont'd)
D851 ASSEMBLY LANGUAGE INSTRUCTIONS

OCTAL CODES AND VARIANT SYLLABLES	MNEMONIC	ADDRESS FIELD	OPERATION
20 .i. iii SSSx		ADDRESS, n	Generate Literal "n" in Stack; Add to BXR on Entry to Subroutine
20 .i. iii SSSx		ADDRESS, *	Add Quantity Already in Stack to BXR on Entry to Subroutine
22 .o.ooo SSSx		ADDRESS	Use Word Fetched from Memory Module as Operand in Current Instruction
22 .i.i. SSSx			Not Defined
22 oo. . . SSSx		ADDRESS	Compute Memory Module Address as Self-Relative
22 oi. . . SSSx		ADDRESS, R	Compute Memory Module Relative to Base Address Register
22 io . . . SSSx		ADDRESS, A	Compute Memory Module Address as Absolute
22 ii. . . SSSx		ADDRESS, X	Compute Memory Module Address Relative to Base Index Register
25	SST		Selective Set
25	LAD		Logical Add
26	SCL		Selective Clear
26	LMP		Logical Multiply
27	SCM		Selective Complement
27	HAD		Half-Add
27	BBC		Bit-by-Bit Compare
36 Sx 40	SR	COUNT	Shift Right by Amount Given
22 .o.ooo SSSx 40	SR	*ADDRESS	Shift Right Indirect
12 Sx 40	SR	*ADDRESS, L	Shift Right
61 04 70	STB	CLEAR	Clear All Tag Bits
61 04 60	STB	CCB	Clear Control Tag Bits
61 04 11	STB	USED	Set Tag Bit to Indicate Used Word
61 04 10	STB	UNUSED	Set Tag Bit to Indicate Unused Word
61 04 60	STB	NQP	Set Tag Bits to Indicate Tag-Bit No Operation
61 04 62	STB	INT	Set Tag Bits to Cause Interrupt
61 04 64	STB	JUMP	Set Tag Bits to Cause Jump
61 04 66	STB	EMPTY	Set Tag Bits to Indicate Empty in Local Data Buffer
61 05 70	JTB	CLEAR	Jump on All Tag Bits Being Zero
61 05 60	JTB	CCB	Jump on All Control Tag Bits Being Zero
61 05 11	JTB	USED	Jump on Tag Bit Indicating Used Word
61 05 10	JTB	UNUSED	Jump on Tag Bit Indicating Unused Word
61 05 60	JTB	NQP	Jump on Tag Bits Indicating Tag Bit No Operation
61 05 62	JTB	INT	Jump on Tag Bits Indicating Interrupt
61 05 64	JTB	JUMP	Jump on Tag Bits Indicating Jump
61 05 66	JTB	EMPTY	Jump on Tag Bits Indicating Empty in Local Data Buffer
04 S 62 ooo . . .	X--	NUMBER	Modify Specified Index Without Testing
22 .o.ooo SSSx 64	JGR	ADDRESS	Jump on Result of Test for Greater with Word Fetched from Memory Module
12 Sx 64	JGR	ADDRESS, L	Jump on Result of Test for Greater Word Fetched from Local Data Buffer
22 .o.ooo SSSx 73 00	JSI	ADDRESS	Jump on Result of Sign-Test of Word in Memory Module
12 Sx 73 00	JSI	ADDRESS, L	Jump on Result of Sign-Test of Word in Local Data Buffer
20 .oo . . . SSSx 74	JUMP	ADDRESS	Jump Unconditionally to Specified Address

APPENDIX D

ALPHABETICAL LISTING OF INSTRUCTION REPERTOIRE

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
ADD	Add	6
ALPH	Alphameric Mode	59
AMA	Add Memory and AR	32
AND	AND	13
B0000		17
B0001	AND Expression	13
B0010	Implication Expression	16
B0011	B True	17
B0100	Reverse Implication	18
B0101	A True	18
B0110	Exclusive OR Expression	14
B0111	OR Expression	13
B1000	NOR	18
B1001	Material Equivalence Operations	18
B1010	Not A	19
B1011	Reverse Implication Not	19
B1100	B Not	19
B1101	Implication Not	20
B1110	NAND	20

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
B1111		20
BBC	Bit-By-Bit Compare	14
BLØCK	Fill Current Four-Word Memory Block with NØP's	60
CCB	Clear Condition Bit	57
CCM	Communicate with Computer Module	55
CHB	Complement Bit	21
CHS	Change Sign	23
CLB	Clear Bit	21
CLR	Clear to Zero	15
CØM	Complement	15
CYCD	Cycle Stack Down	37
CYCU	Cycle Stack Up	37
DIV	Divide	7
DLM	Dump LDB into Memory	30
DRM	Dump Registers into Memory	30
DUP	Duplicate Operand in Stack	35
DUPD	Duplicate Double Length	35
ENM	Enter Normal Operational Mode	54
ESP	Enter Executive and Scheduling Program	54
EXT	Extract	16
FAS	Fetch Absolute Address from AR to Stack	29
FBML	Fetch Block from Memory to LDB	27
FCML	Fetch Character Stream to LDB	27
FILL	Set to Full Scale	15
FIX	Fixed-Point Mode	59
FLA	Fetch LDB Word to AR	32
FLR	Float Remainder	7
FLS	Fetch from LDB to Stack	28
FLT	Floating-Point Mode	58
FMA	Fetch from Memory to AR	32
FML	Fetch from Memory to LDB	26
FMS	Fetch from Memory to Stack	26

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
FRS	Fetch Register to Stack	31
HAD	Half-Add	14
II \emptyset	Interrupt Input/Output Program	56
IMP	Implication	16
INB	Insert Bit	22
INF	Insert Field	16
INS	Insert Sign	23
INT	Integer Mode	57
I \emptyset	Initiate Input/Output Program	56
JBT	Jump on Result of Bit Test in Top of Stack	52
JCB	Jump on State of Condition Bit	49
JEQ	Jump on Result of Test for Equality	51
JFS	Jump on Result of Test for Full Scale	51
JGA	Jump on Result of Test for Absolute-Value Greater	50
JGR	Jump on Result of Test for Greater	49
JLA	Jump on Result of Absolute-Value Less	50
JLS	Jump on Result of Test for Less	50
JSI	Jump on Result of Sign Test	52
JTB	Jump on Tag Bit(s)	42
JUMP	Jump Unconditionally	52
JUS	Justify	11
JXEL	Jump on Result of Index Equal to Limit Test	43
JXES	Jump on Result of Index Equal to Stack Test	43
JXEZ	Jump on Result of Index Equal to Zero Test	42
JXGL	Jump on Result of Index Greater Than Limit Test	43
JXGS	Jump on Result of Index Greater Than Stack Test	44
JXLL	Jump on Result of Index Less Than Limit Test	44
JXLS	Jump on Result of Index Less Than Stack Test	45
JX __ AI	Add Increment to Index After Testing	47
JX __ A \emptyset	Add One to Index After Testing	47
JX --AS	Add Stack to Index After Testing	48
JX __RL	Refill Index Content from Limit Field After Testing	48

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
JX __RS	Replace Index by Top of Stack After Testing	47
JS __SØ	Subtract One from Index After Testing	48
JX __SS	Subtract Top of Stack from Index After Testing	49
JZE	Jump on Result of Test for Zero	51
LAD	Logical Add	13
LIT	Enter Literal Word in Stack	25
LML	Load Data Block from Memory into LDB	30
LMP	Logical Multiply	13
LMR	Load Memory into Registers	31
LØG	Logical Mode	59
MEM	Initiate Memory Module Program	56
MUL	Multiply	7
NEG	Set Negative	22
NØP	No Operation	54
NØRM	Normalize	10
NØT	Logical Inversion	15
ØR	OR	13
ØRX	Exclusive OR	14
PØS	Set Positive	22
QUAD	Quadruplicate Stack Operand	36
RET	Return from Subroutine	53
RETI	Return to Interrupted Program at Point of Interrupt	53
REV	Reverse Operands in Stack	36
REVD	Double-Length Reverse	36
RL	Rotate Left	11
RND	Round	8
RR	Rotate Right	10
RTS	Rearrange Top of Stack	34
SAC	Shift and Count	10
SB	Set Bit	21
SCL	Selective Clear	13
SCM	Selective Complement	14

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
SFCN	Execute Special Function	55
SIP	Significant-Point Mode	58
SJF	Set Up Jump If False	40
SJSF	Set Up Jump to Subroutine If False	41
SJST	Set Up Jump to Subroutine If True	41
SJT	Set Up Jump If True	41
SL	Shift Left	9
SLIT	Enter Short Literal in Stack	25
SLP	Set LDB Pointer	29
SQR	Square Root	8
SPP	Specified-Point Mode	58
SR	Shift Right	9
SSH	Streaming Shift	12
SSL	Store from Stack to LDB	28
SSM	Store from Stack to Memory	29
SSR	Store from Stack to Register	31
SST	Selective Set	13
STB	Set or Clear Tag Bit	23
STEP	Step Stack Up	37
STØP	Stop Processor	54
SUB	Subtract	6
TRIP	Triplicate Operand in Stack	35
UNJ	Unjustify	11
WØRD	Fill Current Instruction Word with NØP's	61
X	Index	38
XA	Index Augmented	38
XAI	Add Increment to Index	47
XAO	Add One to Index	47
XAS	Add Stack to Index	48
XM	Index and Modify	39
XMA	Index Augmented and Modified	39
XRL	Refill Index Content from Limit Field	48

<u>Mnemonic</u>	<u>Operation</u>	<u>Page</u>
XRS	Replace Index by Top of Stack	47
XS	Index by Top of Stack	39
X \emptyset	Subtract One from Index	48
XSS	Subtract Top of Stack from Index	49



Burroughs Corporation

Detroit 32, Michigan

NEW DIMENSIONS / in computation for military systems